The Elementary

Structural Description

Language

The Elementary Structural Description Language - ESDL - is a notation to facilitate the description of digital systems in a manner which is amenable to further automatic processing by a computer. Although ESDL is a textual notation and uses only ASCII characters, it has been designed to be as close as possible, semantically, to the languages of hand sketched logic diagrams and system block diagrams. Consequently it is very easy for a designer to sit at a computer terminal and directly translate the 'back of an envelope' denotation of his initial ideas to a form that can easily be processed mechanically. No special input devices are required - any character oriented computer terminal will do - and a significant amount of design documentation can be produced automatically from an ESDL description.

ESDL was first defined by P McLellan in the summer of 1977, and was extended and re-defined by L D Smith in Aug 1978. L D Smith was responsible for the definition of I-code.

L D Smith
Sept 1979

Contents

Appendix 1

Appendix 2

Appendix 3

Appendix 4

Appendix 5

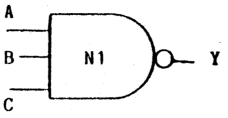
Appendix 6

Alphabetical Index

Introduction

ESDL is a textual notation for describing the structural features of digital systems. It is semantically close to the language of hand sketched logic diagrams and the following examples show that it is very easy to generate an ESDL description, by inspection, from such sketches:

Sketch

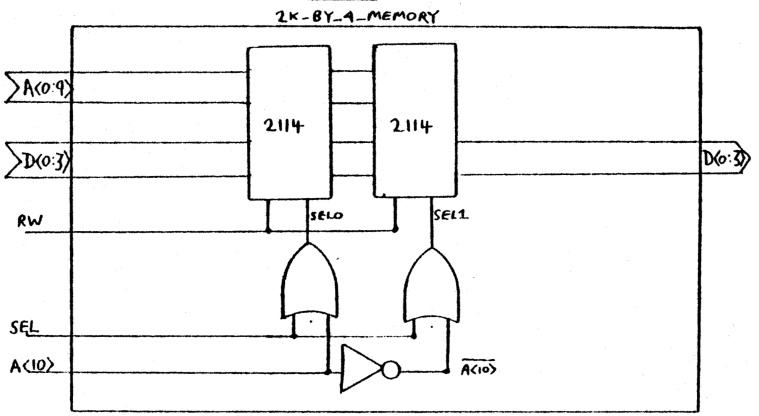


ESDL

```
N1:NAND(A,B,C)->Y
```

Note that all signals must be named, in order that they may be referenced, and that the gate is referenced by the name of its type. In this example the gate has also been given a label to distinguish it from other gates of the same type. This is not usual practice, but illustrates the ease with which an ESDL description can be made to incorporate the information content of the original sketch. The example below illustrates how new building blocks can be defined from existing black-box specifications:

Sketch



ESDL

```
$ SEL is asserted when low, and selects the chip
$ RW is 0 to read and 1 to write
SPEC 2114(ADDR<0:9>,DATA<0:3>,RW,SEL)->DATA<0:3>

UNIT 2K_BY4__MEMORY(A<0:10>,D<0:3>,RW,SEL)->D<0:3>
   2114(A<0:9>,D<0:3>,RW,SEL0)->D<0:3>
   2114(A<0:9>,D<0:3>,RW,SEL1)->D<0:3>
   OR(SEL,A<10>)->SEL0      $ Asserted when low
   INV(A<10>)->A10'         $ 0 when A<10> is 1
   OR(SEL,A10' )->SEL1      $ Asserted when low
END
```

The new building block 2K_BY4_MEMORY is defined from two 2114 1K X 4-bit memory chips and the logic circuitry necessary to select one chip or the

other according to the value of the most significant address bit. The ESDL description commences with a syntactic specification of the 2114 chip in which DATA is declared to be an array of 4 input-output signals indexed by 0 through 3. DATA<i> is declared to be in input-output signal by virtue of appearing in both the input specification and the output specification of 2114 (on both sides of the arrow). The SPEC further serves to remind the reader of the significance of each of the signals to or from the 2114 chip, since each signal is given a name suggestive of its meaning. The ESDL compiler attributes no meaning to a signal other than by virtue of its position in the signal list, so it is important that each instance of 2114 has its various signals in the correct positions. This is easy to check visually when a SPEC is provided.

It should also be noted that the UNIT (representing a building block) delimits the scope of names of signals. This is a programming language property that logic diagrams and block diagrams do not possess.

In the example below a 4 bit Johnson counter is described in terms of D-type flip-flops, which are in turn described in terms of primitive gates. This is not a very sensible thing to do but it illustrates the use of hierarchical descriptions in ESDL and the obvious scoping rules for names.

```
GENERIC SPEC NAND(?,?,?)->? DELAY 5:10:3:6
   $ declare all 3-input NAND gates to have
   $ min/max turn-on/off delays of 5,10,3, and 6
   $ (but don't force all NAND gates to have 3 inputs)

UNIT 4_BIT_JOHNSON_COUNTER(CLOCK,CLEAR)->D<0:3>

   UNIT DTFF(CK,D,P,CL)->Q,Q'
      $ a D-type flip-flop in terms of gates ...
      $ ... the UNIT header is automatically a SPEC
      NAND(J,P,K')->J'
      NAND(CK,J',CL)->J       $ set for output stage
      NAND(CK,K',J)->K        $ reset for output stage
      NAND(D,K,CL)->K'
      $ the output S/R flip-flop
      NAND(J,Q',P)->Q
      NAND(K,Q,CL)->Q'
   END

   $ Now the 4 D-type flip-flops and inverter
   $ which form the Johnson counter

   DTFF(CLOCK,D3', .1,CLEAR)->D<0>,?  $ note the use of ? to
   DTFF(CLOCK,D<0>,.1,CLEAR)->D<1>,?  $ denote deliberately
   DTFF(CLOCK,D<1>,.1,CLEAR)->D<2>,?  $ omitted connections
   DTFF(CLOCK,D<2>,.1,CLEAR)->D<3),?
   NOT(D<3>)->D3'
END
```

Physical or constructional details can be included in an ESDL description just as easily. The example on the next page of a Texas series 7400 chip shows how pin numbering (or naming) and chip carrier information can be included in a natural manner.

```
$ a CHIP is much like a UNIT ...
$ except that it has more physical details
$ such as power (.VCC) and ground (.GND) connections
$ and PIN numbers associated with each signal ...

CHIP T7410(A1,B1,C1,A2,B2,C2,A3,B3,C3,.VCC,.GND)
        ->Y1,Y2,Y3
   $ there is a pin number for each input or
   $ output in order ...
   PINS 1,2,13,3,4,5,9,10,11,14,7,12,6,8

   $ ... the CHIP is mounted in a DIL14 package ...
   ON   DIL14

   $ and contains three three-input NAND gates
   NAND(A1,B1,C1)->Y1
   NAND(A2,B2,C2)->Y2
   NAND(A3,B3,C3)->Y3
END
```

In a similar manner it is possible to define the physical details of packages, and pre-slotted boards. This will be discussed later after a fuller description of the language (see appendices 4-6).

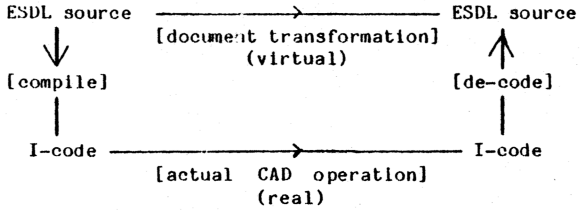
The ESDL compiler

Before an ESDL description can be processed by a computer it must be translated to a form that is easier for a machine to read than is the user-oriented source text. This task is accomplished by the ESDL compiler which also checks that the description really is a description. A bonus of this syntax checking and translation step is that many trivial errors of transcription can be detected before the description undergoes further processing. Of course only syntactic errors can be detected, but, surprisingly, this accounts for over half of the errors that are made in transcribing from the primary informal denotation of an idea to ESDL. Errors such as mistyping a signal name (which causes an unused signal name to be introduced into the description), numbering too many (or too few) pins, and instancing a UNIT in a manner which does not match its SPECification, are easily detected and located by the ESDL compiler.

When a description is found to be syntactically correct it is translated into a more compact intermediate format henceforth referred to as I-code (see appendix 3 for a definition of I-code). I-code is another textual notation, with a very simple syntax, and is always guaranteed (by any process which generates it) to be syntactically correct. Consequently, any applications program which reads I-code need do no syntax checking or syntax-error diagnosis. This is a considerable simplification of these programs.

The most important property of I-code is that it has the same information content as the ESDL source text from which it was generated. Consequently it is always possible to regenerate an equivalent ESDL source text from I-code. In fact it is not necessary that the I-code was generated from ESDL source text; it could have been generated by a program which accepts I-code as input and produces I-code as output. It is this ability to 'invert' I-code that allows us to think of computer aided design operations as transformations from an ESDL document to another ESDL document, rather than operations on intermediate representations. A

computer aided design system which operates on ESDL descriptions can then be thought of as a document flow system in which design documents flow through various processing stages, gaining information as computational work is performed upon them. This inversion property is most elegantly summarised by the following isomorphism diagram:



If CAD processes are always constrained to produce I-code as output (except as final output which is not fed back into the design system), then it is never possible to create an object that cannot be described in ESDL (by virtue of the fact that any I-code description has an equivalent ESDL description associated with it). Thus ESDL describes the universe of such a computer aided design system

A particular example of I-code to I-code transformation, the flattening of hierarchical descriptions through the replacement of UNIT instances by the appropriate UNIT bodies, will be discussed in detail later, after a fuller description of the language. This process of flattening hierarchies (almost macro expansion) groups naturally with the processes of compilation and inversion introduced above, as all three are intimately connected with the manipulation of ESDL source documents.

The ESDL compiler also produces a listing of the input text on which the position of errors is marked by a ¦ character. As far as possible the line structure of the listing follows that of the source but occasionally source lines have to be split over several listing lines. In this case the continuation is marked with a + character after the line number. Parts of the source text which are skipped during recovery from syntax errors are marked by a $ character after the line number. Error messages are output on the listing as close as possible to the site of the error, however, certain classes of warning message can only be produced at the end of a UNIT (such as the unused signal name warning - see appendix 1 for details).

The textual layout of ESDL

ESDL is a textual notation, using only ASCII characters, designed for the description and definition of black boxes with inputs and outputs. The notation is completely free-format (there is no concept of 'statement') and spaces and newlines can be used anywhere, except in the middle of a syntactic token, to improve readability. Similarly, comments can appear between any pair of syntactic tokens. ESDL can be parsed with single token look-ahead (though not with single character look-ahead) and the language is context sensitive in a trivial way (there is an ambiguity between Tags, Strings, and Numbers, the character sequence 1234 being a valid member of all three classes). A formal definition of ESDL and the compiler's syntax-error recovery strategy can be found in appendix 1.

Reserved words

Certain words have a pre-defined significance in ESDL, and these are listed below. Since these words are read from a pre-definition file whenever the compiler is invoked, there is freedom to 'tune' ESDL to a particular application by altering these words. Consequently the list below is implementation dependent and liable to extension. In a similar vein the ESDL compiler usually ignores the distinction between upper and lower case letters (translating both to upper case) except when the letter occurs in a string (when no translation occurs). However, the compiler can be made to preserve the distinction between the cases and thus it would be possible to establish a convention that keywords (reserved words) were to appear in one case and all other identifiers in the other (one of the 'stropping' conventions available in ALGOL68). This has not been done in any of the implementations to which this document refers, and henceforth only upper case will be used in example ESDL descriptions. The compiler can also be made to convert lower case letters within strings to upper case by using COPTION STRCONVERT (see later - Compiler control statements). The reserved words are:

AT	BOARD	CHIP	COPTION	DEFINE
DELAY	END	FINISH	GENERATE	GENERIC
LISTOFF	LISTON	NOGENERATE		ON
OPTION	PACK	PACKAGE	PINS	PLACE
SIZE	SPEC	SUBPACK	UNIT	VALUE
WIRE				

The syntactic tokens of ESDL

Syntactic tokens are the atomic objects recognised by the ESDL compiler. These comprise comments, identifiers, numbers, strings, reserved words, and a number of special characters not included in these classes such as ( ) , < > -> = + - * / and ?.

Comments

Comments begin with the character $ and are terminated by the next $ or newline character to occur. A comment may appear between any pair of syntactic tokens.

Identifiers (Tags)

Identifiers, or tags, consist of letters, numbers, and the following characters: ! # % & ' [ ] . \ and _. Tags beginning with '.' have a special significance (to be explained later - see the section entitled 'The scope of names in an ESDL description'). The characters '#', '_', '[' and ']' are used by the program which flattens hierarchies in order to generate unique names (see later) and consequently the use of these characters in identifiers should be avoided until their standard use is well understood. There is no constraint that a tag must start with a letter and 1234 is a valid tag (so is !#'&%[_] - however not a very sensible one).
Conventionally an apostrophe at the end of a signal name denotes an inverse of the named signal, thus, for example, Q and Q' are often used to name the complementary outputs of a flip-flop.

Numbers

A number in ESDL is an unsigned integer in a specified base and takes the form <base>_<number-in-base>. The base and underscore character can be omitted in which case base 10 is assumed (as would be expected). A number is any sequence of digits whose value is less than the base (which is usually 10). For bases greater than 10 the letters A, B, C, ... and a, b, c, ... are considered to have the values 10, 11, 12, ... For example, 10, 16_FC, 8_20_0a0 are all valid denotations for the decimal numbers 10, 252, and 160 respectively.

Strings

A string in ESDL is either any sequence of characters (except newline) enclosed by string quotes (") in which the quote character itself is represented by a double occurrence (""), or it is any sequence of characters not beginning with a string quote (") and not containing the characters space, newline, comma, or right parenthesis. Any of these four characters terminates (and is not included in) an unquoted string. Strings may not have a length exceeding 255 characters. Within strings lower case letters are not converted to upper case unless the compiler option COPTION STRCONVERT is specified (see later - Compiler control statements). All occurrences of the character '^' are replaced by '|', and there is no denotation for the newline character within a string. This is because '^' has a special significance in I-code. The newline character has a special insignificance in I-code, and cannot be permitted to occur in strings, numbers, or tags. A quoted string can span two source lines under the conditions that the first non-space character to follow the end of the first part of the string is a newline and the first non-space character on the continuation line is a string quote ("). For example:

```
"This is rather a long string and is continued"
      " on another line of the source text."
```

The structure of an ESDL description

In this section the syntax of a number of ESDL constructs will be presented in a modified Backus-Naur notation. The brackets { and } will be used as meta-brackets to denote that an item is optional. An asterisk after a closing meta-bracket denotes that the object inside can appear any number of times (including zero times). Non-terminal symbols have names consisting of lower case letters and terminal symbols appear as they would in an ESDL description (see above for list of reserved words and special characters). Identifiers, numbers, and strings are denoted by the names Tag, Number, and String (with a capital letter at the beginning). Alternative right hand sides of the definition of a non-terminal symbol appear on separate consecutive lines, and the left and right hand sides of a definition are separated by the meta symbol ::= ('is defined to be').

An ESDL description consists of any number of units, each of which defines a black box with inputs and outputs, followed by the reserved word FINISH. Formally:

```
ESDL_description ::= {unit}* FINISH
```

Units

Each unit is either a SPECification, in which case it comprises only a unit header, or it has a (possibly null) body which forms the definition of the unit. In both cases the unit header forms a SPEC for instances of the unit, and can be preceded by the reserved word GENERIC to indicate that this is only one of a class of definitions for units of this name (for example AND gates may be defined with 2 inputs, 3 inputs, 4 inputs, 8 inputs, etc). Formally:

```
unit          ::= {GENERIC} rest-of-unit
                  ;
rest-of-unit  ::= SPEC unit-header
                  ucbp unit-header unit-body END
ucbp          ::= UNIT
                  CHIP
                  BOARD
                  PACK
```

The ';' symbol is rarely (if ever) required by users of ESDL, but may be required by those who prepare pre-definition files. It is used to force compiler control keywords (see later) to have immediate effect in circumstances in which the effect of toggling a compiler switch would be delayed.

Unit headers

A unit header consists of an optional label, the name of the unit, a list of input signals, a list of output signals, and optionally some extra information such as delay values, chip position, etc. Formally:

```
unit-header  ::= {Tag : } Tag rest-of-head {extra}*
rest-of-head ::= ( input-signal-list ) {-> rhs}
                 -> rhs
rhs          ::= ( output-signal-list )
                   output-signal-list
signal-list  ::= signal-names {, signal-names}*
signal-names ::= Tag {< expn {: expn} >}
```

Signal names

Note that a signal name can be subscripted by an integer expression (which must be evaluatable at compile time). This encourages the sensible naming of 'arrays' of signals such as are found in bus structures. In fact, two integer expressions separated by a colon can be specified and this is shorthand for a range of subscripted signals, for example:

BUS<7:0> may be written as shorthand for ...
BUS<7>,BUS<6>,BUS<5>,BUS<4>,BUS<3>,BUS<2>,BUS<1>,BUS<0>

The direction in which the subscripts in the range run is determined by inspection in the obvious manner. It should be noted that the longhand form of BUS<7:0> is the canonical form and that is is this form that is generated when I-code is decoded by the DCODE program.

Examples of unit headers

Some examples of valid unit headers follow and should be compared with the BNF descriptions given above. Note the use of '?' to denote a signal in a SPEC (there is no sense in naming a signal if there is no unit body to reference it and it is not an input-output).

```
SPEC NAND(?,?,?,?)->?       $ 4-input NAND gate
BOARD EURO1:EIA_INTERFACE(DATA<0:6>,RESET,GO)
                          ->DONE,ERROR
PACK CERAMIC:DIL16(.VCC,.GND)
GENERIC SPEC AND(?,?)->?    $ 2-input AND gate
GENERIC SPEC AND(?,?,?)->? $ 3-input AND gate
```

Input-outputs

As was mentioned earlier it is possible to specify that a signal is an input-output. This was used in the example 2K_BY4_MEMORY where the signals DATA<0:3> were declared to be input-outputs (clearly, the memory chip either reads from, or writes to, DATA, according to whether it it enabled to read or enabled to write). A signal is declared to be an input-output by virtue of appearing in both the input and output lists of a SPEC or unit header. For example:

```
   SPEC 2114(ADDR<0:9>,DATA<0:3>,RW,SEL)->DATA<0:3>
   SPEC TRISTATE(I,IO,.VCC,.GND)->IO,O
or SPEC TRISTATE(?,IO,?,?)->IO,?
   UNIT FRED(A<1>,A<2>,A<3>)->A<4>,A<3>,A<2>
      $ A<2> and A<3> are input-outputs
      $ A<1> is an input, and A<4> is an output
```

Extra information attatched to unit headers

To every unit header, whether it constitutes the black box input output specification of a unit or stands for an instance of a unit, it is possible to attatch certain extra information such as pin names, delay values etc. There are three classes of such extra information. The first is the PINS construct, which specifies a pin name to be associated with each signal name in the unit header (note that the same pin must be specified twice in the case of input-outputs). The second is the OPTION construct which specifies an integer expression which is interpreted by applications programs, which read the I-code generated by the ESDL compiler, as a series of flags (currently only one flag is used). The third class consists of a number of reserved words each of which can be followed by a String (see above for the definition of a String) which is interpreted by applications programs which read I-code. This class is extensible, and (as previously stated) can be tuned to a particular task in any specific implementation of ESDL. However, the following descriptions and examples apply to all implementations which are referenced by this manual, and it is unlikely that other implementations will differ radically from these. Formally:

```
extra      ::= OPTION  oexpn
               PINS    pin-list
               AT      parm-string
               DELAY   parm-string
               ON      parm-string
               PACKAGE parm-string
               PLACE   parm-string
```

```
                    SIZE    parm-string
                    SUBPACK parm-string
                    VALUE   parm-string
oexpn          ::= ( expn )
                     expn
pin-list       ::= ( s-list )
                     s-list
s-list         ::= String [, String]*
parm-string    ::= ( String )
                     String
```

For example:

```
NOT(A)->A'    ON T7404   DELAY (8:12:7:10)
T7474(P4,.H2,RESET,CLOCK,DU,.H2,RESET,CLOCK,.VCC,.GND)
   ->DU,?,TO4,?
   PINS 2,4,1,3,12,10,13,11,14,7,,5,6,9,8
   ON   DIL14
   AT   A3
FRED(A)->OUT    OPTION 1   $ no expansion by FLATTEN
SPEC INOUT(?,?,IO)->IO,?
   PINS  (A1,A2,A3,A3,A4)  $ note repetition of A3
```

Defaults for extra information

Note that the value of any particular 'parameter' (for lack of a better word) applied to a unit instance automatically overrides the value which is passed, by default, from the SPEC or unit definition to all instances of that unit. For example:

```
GENERIC SPEC NAND(?,?,?)->?      DELAY 6:12:7:10
NAND(A,B,C)->Y  $ with standard delays
NAND(P,Q,R)->Z  $ a fast gate $  DELAY 4:6:3:5
```

The body of a unit defintion

The body of an ESDL unit definition consists of any number of unit definitions, unit instances, and WIRE constructs. A unit instance is syntactically identical to a unit-header (defined above) and denotes that the named unit is to be instanced with the given signals in the given positions in the input and output signal lists. Several examples of this have been given already in the introduction. Formally:

```
unit-body     ::= [body-stmnt]*
body-stmnt    ::= WIRE wire-stmnt
                  unit
                  instance
instance      ::= unit-header
wire-stmnt    ::= [pstag] rest-of-wire
rest-of-wire  ::= ( list-of-tags ) [-> rhs-of-wire]
                  -> rhs-of-wire
rhs-of-wire   ::= ( list-of-tags )
                    list-of-tags
list-of-tags  ::= pstag [, pstag]*
pstag         ::= Tag [<expn>]
```

The WIRE construct

The WIRE construct is used to create a single electrical net from the signal names listed. No significance attatches to the different syntactic positions that signal names can occupy in the WIRE construct - this is merely to allow uniformity of notation with unit headers and instances. For example:

```
WIRE CLOCK->CLK       $ make CLOCK and CLK the same net
WIRE (.1,.H,.HIGH)    $ different names for logical 1
WIRE .VCC->.GND       $ Whoops! A catastrophe.
```

The scope of names in an ESDL description

Unit definitions can be nested to any depth, and their names obey the obvious scope rules. Units must be defined before they are referenced and become out of scope as soon as the properly nested END is encountered. All signal names introduced in the body of a unit have scope local to that unit definition unless the signal name begins with a '.', in which case the name is considered to have global scope. This feature is useful for dealing with obviously global connections such as power, ground, constant values, master clear, clock signals, etc. The global signals .VCC, .GND, and .1 have already been encountered in the examples above. These examples should be perused further if there remains any uncertainty about the semantics of global signal names.

A simple macro facility

ESDL implements a simple parameterless macro facility whereby a tag can be defined to have a textual value. Whenever the ESDL compiler reads such a tag it automatically substitutes the textual value (which itself may contain references to tags with textual values). A simple, and useful, application of this facility is to give a symbolic name to a numeric quantity. In an example above the OPTION reserved word was seen followed by the numeric quantity 1 to denote that the unit was not to be expanded by the hierarchy flattening program. It is much cleaner and more comprehensible to express this as

```
OPTION NOEXPAND
```

where NOEXPAND is defined to have the textual value 1. This facility can also be used as shorthand, or to provide alternative expressions for ESDL constructs. For example

```
DRIVER(INPUT,P) PRODUCES OUTPUT
```

where PRODUCES has the textual value "->" and P has the textual value ".VCC,.GND" is equivalent to the statement

```
DRIVER(INPUT,.VCC,.GND)->OUTPUT
```

Defining tags to have a textual value

The macro facility introduced above requires that at tag be given a textual value. This is accomplished with the DEFINE statement. Formally:

define-stmnt ::= DEFINE Tag = String {, Tag = String}*

For example:

DEFINE P=".VCC,.GND", PRODUCES = ->

(Why does the string .VCC,.GND have to be enclosed in string quotes when the string -> need not be?)

A define statement can occur anywhere that a unit would be valid (see earlier), which is to say anywhere between unit definitions, unit instances, and wire constructs. No macro substitution takes place within a define statement so it is possible to re-define a defined tag. To be meaningful a defined tag must comprise an integral number of syntactic tokens, and an instance of a defined tag (macro call) will only be recognised if it stands as a complete syntactic token. For example, if THIS is defined to be "43" and THAT is defined to be "7" then THIS.THAT is not recognised as 43.7 (but as the Tag THIS.THAT), however THIS*THAT is recognised as 43*7 because THIS, THAT, and * are all distinct syntactic tokens.

If more sophistacted macro-processing is required then it is recommended that a general purpose macro-processor be used, such as GPM [A general purpose macrogenerator, C Strachey, Computer Journal 8/3, 1965, pp225-241] or the pre-processor due to McLellan [undocumented] which is available on several Departmental machines (the program is called PRE and is available on VAX and LEGOS).

Compiler control statements

There are a number of compiler control reserved words which are used to control the production (or otherwise) of a listing file, to determine whether or not macros (described in the above section) are expanded in the listing, and to set certain internal compiler flags. These keywords can occur between any pair of syntactic tokens, take effect almost immediately, and are listed below:

LISTON. Turn on the production of a compiler listing. This is the default established at the end of the pre-definition file.

LISTOFF. Turn off the production of a compiler listing.

NOGENERATE. Do not list the expansion of macros. This is the default established at the end of the compiler pre-definition file.

GENERATE. List the expanded form of all macros.

COPTION <numeric-expression>. Set the compiler's internal flags to the value of the expression (which must evaluate to an integer). Initially the flags are all set to 0. The compiler pre-definition file establishes mnemonics for the values required to set these flags in a convenient manner, and these are explained below:

DUMP1 (=1) Dump the keywords part of the stack on the diagnostic dump stream (this facility is not normally available to users).

DUMP2 (=2) Dump the stack after the END of each unit definition (this facility is not normally available to users).

DUMP3 (=4) Dump the stack at the end of the description (this facility is not normally available to users).

FORGET (=8) Forget the SPEC of a unit as soon as the END of its definition is encountered. This can save a considerable amount of workspace if the ESDL description consists of a large number of small units which are independent (such as the definition of a collection of chips) and allows larger descriptions to be compiled.

STRCONVERT (=16) Convert all lower case letters within strings to upper case. This is useful when a string containing lower case letters defines a name that is subsequently matched (perhaps by an applications program) to a tag defined elsewhere (tags are converted to upper case by default ... see earlier, Reserved words).

PUTSPECS (=32) Output all SPECs to the I-code. This feature is for diagnostic purposes only and is of no use to users.

NOSIGNALS (=64) Allow unit instances to reference no signal names (normally this condition is faulted). This is useful when defining pre-slotted boards when the only information to be included in a description is the name of a slot and its position.

Examples of compiler control statements

The example statements

```
LISTOFF
COPTION FORGET+NOSIGNALS
```

cause the compiler listing to be turned off, the SPECification of unit headers to be forgotten as soon as the END of the definition is encountered, and cause unit instances with no references to signal names not to be faulted.

The compiler listing

As well as translating an ESDL source document to I-code the ESDL compiler produces an annotated listing. The compiler listing can be disabled and enabled by means of the compiler control statements LISTOFF and LISTON (see above - Compiler control statements), however lines containing errors are always listed together with the error messsages pertaining thereto. If the compiler control option GENERATE has been specified then all macro invocations are listed in their expanded form. As far as possible the line structure of the listing follows that of the source document. Sometimes it is necessary to break a source line over two

or more listing lines (such as when a macro expansion causes a source line to lengthen, or when a error is flagged and the error message breaks the line) and when this occurs the continuation line is given the same line number as the first part of the line followed by a + character. When recovery from syntax errors causes syntactic tokens to be skipped the skipped tokens are identified by a $ character after the line number and the proportion of tokens skipped is printed at the bottom of the compiler listing (and also on the report stream - usually the terminal from which the ESDL compiler has been invoked). Error messages are largely self explanatory and a list of them, with brief explanations, can be found in appendix 1.

Flattening a hierarchical description

As was mentioned earlier it is possible to flatten, or expand, a hierarchical ESDL description mechanically, and replace references to units by the appropriate unit bodies. This is accomplished by a program, called FLATTEN, which takes hierarchical I-code as input and produces monolithic I-code as output.

A moment's thought will convince the reader that there are certain problems associated with this process. Look back at the definition of the Johnson counter. Within the definition of the D-type flip-flop there are four signal names (J, J', K, and K') which are entirely local to DTFF. Consequently when the four references to DTFF are replaced by DTFF bodies there will be four signals called J which are intended to be distinct! FLATTEN solves this problem by adding a prefix to an instance of a local signal name which makes it distinct from all other instances of it. The form of this prefix is:

```
unit-name (# n) ([ m ]) _
```

where unit-name is the name of the unit definition to which the signal name is local. This is further qualified by # followed by an integer if there is more than one unit of this name whose name is defined in some scope containing the scope of the signal name. The integer takes the value 1 for the first such unit to be defined, 2 for the second, and so forth (where the ordering is the order in which the definitions occur in the source text). If there is only one unit defined with the given name (as in the Johnson counter example) then no qualification with #n takes place.

The prefix is further qualified by [m] where m is an integer taking the value 1 for the first instance of the unit, 2 for the second, and so on (where the ordering is the order in which the instances occur in the source text). The prefix is separated from the signal name by the underline character. In the case of the Johnson counter example (see above) the internal signal name J gives rise to the signal names DTFF[1]_J, DTFF[2]_J, DTFF[3]_J, and DTFF[4]_J within the flattened description of the Johnson counter. To emphasise the manner in which this prefixing operates, the description of the Johnson counter is reproduced below, followed by the flattened description. This is produced by applying the program DCODE to the output of the program FLATTEN. Note that FLATTEN leaves a history of the hierarchy it destroys in the form of a series of mechanically generated comments which note the start and end of the body of an expanded instance.

The hierarchical description

```
UNIT JCOUNT(CLOCK,CLEAR)->D<0:3>
   UNIT DTFF(CK,D,P,CL)->Q,Q'
      NAND(J,P,K')->J'
      NAND(CK,J',CL)->J
      NAND(CK,K',J)->K
      NAND(D,K,CL)->K'
      NAND(J,Q',P)->Q
      NAND(K,Q,CL)->Q'
   END

   DTFF(CLOCK,D3',.1,CLEAR)->D<0>,?
   DTFF(CLOCK,D<0>,.1,CLEAR)->D<1>,?
   DTFF(CLOCK,D<1>,.1,CLEAR)->D<2>,?
   DTFF(CLOCK,D<2>,.1,CLEAR)->D<3>,?
   NOT(D<3>)->D3'
END
```

The flattened description

```
UNIT JCOUNT(CLOCK,CLEAR)->D<0>,D<1>,D<2>,D<3>
$ DTFF
   NAND(DTFF[1]_J,.1,DTFF[1]_K')->DTFF[1]_J'
   NAND(CLOCK,DTFF[1]_J',CLEAR)->DTFF[1]_J
   NAND(CLOCK,DTFF[1]_K',DTFF[1]_J)->DTFF[1]_K
   NAND(D3',DTFF[1]_K,CLEAR)->DTFF[1]_K'
   NAND(DTFF[1]_J,DTFF[1]_Q',.1)->D<0>
   NAND(DTFF[1]_K,D<0>,CLEAR)->DTFF[1]_Q'
$ End of DTFF
$ DTFF
   NAND(DTFF[2]_J,.1,DTFF[2]_K')->DTFF[2]_J'
   NAND(CLOCK,DTFF[2]_J',CLEAR)->DTFF[2]_J
   NAND(CLOCK,DTFF[2]_K',DTFF[2]_J)->DTFF[2]_K
   NAND(D<0>,DTFF[2]_K,CLEAR)->DTFF[2]_K'
   NAND(DTFF[2]_J,DTFF[2]_Q',.1)->D<1>
   NAND(DTFF[2]_K,D<1>,CLEAR)->DTFF[2]_Q'
$ End of DTFF
$ DTFF
   NAND(DTFF[3]_J,.1,DTFF[3]_K')->DTFF[3]_J'
   NAND(CLOCK,DTFF[3]_J',CLEAR)->DTFF[3]_J
   NAND(CLOCK,DTFF[3]_K',DTFF[3]_J)->DTFF[3]_K
   NAND(D<1>,DTFF[3]_K,CLEAR)->DTFF[3]_K'
   NAND(DTFF[3]_J,DTFF[3]_Q',.1)->D<2>
   NAND(DTFF[3]_K,D<2>,CLEAR)->DTFF[3]_Q'
$ End of DTFF
$ DTFF
   NAND(DTFF[4]_J,.1,DTFF[4]_K')->DTFF[4]_J'
   NAND(CLOCK,DTFF[4]_J',CLEAR)->DTFF[4]_J
   NAND(CLOCK,DTFF[4]_K',DTFF[4]_J)->DTFF[4]_K
   NAND(D<2>,DTFF[4]_K,CLEAR)->DTFF[4]_K'
   NAND(DTFF[4]_J,DTFF[4]_Q',.1)->D<3>
   NAND(DTFF[4]_K,D<3>,CLEAR)->DTFF[4]_Q'
$ End of DTFF
   NOT(D<3>)->D3'
END
```

Note particularly how the signal name Q' becomes multiply instanced.

Although Q' is not local to DTFF, no signal name is specified in the Q' position of any of the instances' signal lists. This causes Q' to be treated as being local to the unit body that replaces each instance. Contrast this with the Q position of the signal lists which is variously occupied by D<0>, D<1>, D<2>, and D<3>.

Linking together separately compiled units

A further function performed by FLATTEN is that of linking together separately compiled units. This feature allows libraries of standard units to be constructed and used. For example, it is not usual to expand flip-flops to gate level (as was done in the Johnson counter example): flip-flops are available as complete entities, packaged several to a chip, at even the lowest levels of integration. However, in order to use a gate level simulator that has no inbuilt model of flip-flop it is necessary to make this expansion. A convenient way to reconcile this conflict (to expand or not to expand) is to create a library of flip-flop descriptions. Design documents can then be written in terms of flip-flops (which is convenient for constructional purposes) and expanded, by a mechanical process, when gate level simulation is required.

The precise operation of the linking process is as follows. The first I-code stream (representing the compiled design document) is read, and references to units (due to instances) are resolved if this is possible. There will always be some references that cannot be resolved: lowest level units which have no definition and references to library units. At this stage the second I-code stream (representing a library of independent unit definitions) is read. Any unit whose name matches an unresolved reference is retained and all other units are ignored. The resulting structure is then flattened in the usual manner.

Note that a library unit can be deliberately ignored if OPTION NOEXPAND (see above) is specified for its instances. This can even be applied selectively in order to expand some, but not all, instances.

Invoking ESDL on Departmental computing systems

ESDL, DCODE, and FLATTEN are three of a suite of computer aided design and construction programs known collectively as DL1. DL1 has been implemented on the LEGOS systems and the DEC VAX 11/780 belonging to the Department of Computer Science.

On LEGOS

On the LEGOS systems access can be gained to DL1 by issuing the command LIB CAD. Thereafter, programs of the suite are run in the usual manner. Specifically:

```
ESDL Input1,Input2,Pre-def/I-code,Listing
```

Input1 and Input2 are ESDL source text files and are read in that order. Input2 can be omitted, and will not be read if Input1 is terminated by the reserved word FINISH. Pre-def is the compiler pre-definition file, and, if omitted, defaults to the system pre-definition file (listed in appendix 2). I-code is the output file for the compiled document, and Listing (if specified) will receive a listing of the source text annotated with line

numbers, error messages, and warnings.

```
FLATTEN Input,Library/Output
```

Input is a compiled ESDL description (to be flattened). Library is a library of unit defintions and can be omitted. Output receives the I-code representing the flattened description.

```
DCODE Input/Output
```

Input is any I-code file. Output receives the de-coded input, namely an equivalent ESDL description. If omitted, Output defaults to the temporary file $DCODE.

Examples:

```
ESDL JCOUNT:SRC/JCOUNT:E,JCOUNT:LIST
FLATTEN JCOUNT:E/JCOUNT:F
DCODE   JCOUNT:F/JCOUNT:FSRC
```

On VAX 11/780

On the VAX 11/780 (under the VMS operating system) access is gained to the DL1 system by issuing the command DL1SETUP. Thereafter, programs of the suite are invoked in much the same manner as on LEGOS (see immediately above section for details). The major difference between VAX and LEGOS is the file naming structure. On VAX all file names consist of two parts: a name and an extension (up to three characters in length) separated by a dot. Programs of the DL1 suite use a number of standard extensions which are provided by default if no file name extension is specified when the program is invoked. Consequently it is often possible to invoke a program of the DL1 suite with only a single (input) file name as parameter. The extension for this file name is added by the program and any other filenames required are generated by adding appropriate extensions to the specified file name. In the defaults given below %11 refers to the name of the file specified for the first input stream. Note that the defaults can always be overridden by specifying a full file name explicitly.

Examples:

```
ESDL FRED
ESDL JKMS/,.TT
ESDL JKMS/.N
Defaults: .SRC,.SRC,SIO:[ESDL]ESDL.PRM/%I1.EIC,%I1.LIS

FLATTEN FRED
FLATTEN JKMS/FLIPFLOP
FLATTEN CIRCUIT,JKMS.FIC/SIMCCT.LIB
Defaults: .EIC,.LIB/%I1.FIC

DCODE FRED.EIC
DCODE JKMS.FIC/JKMS.LIS
Defaults: /.TT
```

Appendix 1

A definition of ESDL and its syntax-error recovery strategy

In this appendix a semi-formal definition of ESDL is presented. Embedded in this description is a definition of the syntax-error recovery strategy which is implemented in the ESDL compiler.

The syntax definition is presented in a rather unusual form which requires some explanation. In order to clearly define the error recovery and error message generation mechanisms it is necessary that the denotation for an attempt to recognise a non-terminal symbol allows three outcomes rather than two as in more conventional notations such as BNF. These outcomes are success (denoted by T), failure (denoted by F), and error (denoted by E).

In the syntax graphs that follow, non-terminal symbols are denoted by lower case words. Terminal symbols are denoted by upper case words or special characters, as in the preceding sections of this manual. The first non-terminal in a syntax graph is the name of the whole graph (the left hand side in BNF terms). The flow of control proceeds to the right and downwards, unless otherwise indicated by arrowheads. Conceptually, recognition proceeds by attempting to match the first symbol of the syntax graph to the input text. If this match succeeds then the horizontal branch is taken, otherwise the vertical (downwards) branch is taken and the next (lower down) symbol tried. Eventually an outcome is reached and this is the 'value' of the parse at that point. Note that the recognition of a terminal symbol has only two outcomes; success or failure. If the recognition succeeds then the horizontal branch marked by the terminal symbol is taken. An attempt to recognise a non-terminal symbol can have three outcomes and every non-terminal symbol in a syntax graph is followed by a number of horizontal branches labeled with these outcomes (in diamond shapes). The branch which matches the outcome is taken and outcomes which can never occur do not appear in the graph. Whenever a circle is encountered, the recognition of the non-terminal symbol represented by the syntax graph is terminated with the outcome with which the circle is labeled. Terminal symbols which must be recognised then re-recognised (look-ahead, in effect) are enclosed in rectangles and error messages are enclosed in a box with a right pointing arrow head. Error messages are all of the form '*' followed by 'E' for error or 'W' for warning and the number of the message. These messages are listed below. A continuation of a syntax graph is denoted by a single capital letter enclosed in a right pointing arrowhead and the continuation begins with this same capital letter, rather than the more usual lower case non-terminal name.

Error recovery is performed by the pseudo non-terminal symbol 'recover' which reads syntactic tokens until a token belonging to one of the two recovery classes is recognised. All tokens thus skipped are marked on the compiler listing with a '$' symbol at the beginning of the line. The recovery classes are listed at the end of the appendix. Essentially, a group1 token (denoted below by GP1) is anything that denotes the start of a unit (such as UNIT, SPEC, etc) and a group2 token (denoted below by GP2) anything that might denote a sensible re-starting place within a unit definition (such as UNIT, WIRE, a Tag - denoting the start of a unit instance- etc). Consequently syntax-error recovery is crude, but effective. More sophisticated error recovery is difficult in a language with no statement boundaries and less textual redundancy than is usual in a conventional programming language such as PASCAL (indeed, error recovery is only possible because of redundancy).

Recovery groups

The first recovery group contains all tokens that might begin a unit definition, namely DEFINE, GENERIC, SPEC, UNIT, CHIP, BOARD, and PACK. The second recovery group contains all tokens that could reasonably begin a 'statement' within a unit definition (of course, there are no statements, which is the problem, so this is a guess at where the parse might be picked up successfully). This group contains END, ';', newline followed by Tag, and WIRE.

Error messages from the ESDL compiler

The warnings, errors, and disasters listed below are referenced from the syntax graph definition of ESDL by error number. Most of the messages are self explanatory, but a few words of explanation are given for those that are not.

Warnings

W1: Unexpected end of input
Most probably the FINISH statement has been omitted from the end of the input. If a message of the form "n/m input ignored" also occurs then this message indicates that the compiler has become badly lost.

W2: missing ')'

W3: missing '>'

W4: missing '='

The following warnings are issued at the end of a unit (when the END statement is encountered).

* unused? FRED
The signal name FRED is occurs only once in the unit (thus it cannot connect two or more subunits). This is either an error or the specification of a deliberately unused pin. In the latter case use ? instead of an otherwise unused Tag.
* no fan-in? FRED
The signal FRED connects only inputs (thus the net FRED has no fan-in).

* no fan-out? FRED
The signal FRED connects only outputs (thus the net FRED has no fan-out).

Errors

E1: not recognised
This indicates that a non-specific syntax error has occurred. Usually the message is output after the compiler becomes lost or when an incorrect attempt at error recovery is made.

E2: missing tag

E3: missing '->'

E4: invalid expression

E5: missing '('

E6: missing ')'
Normaly an omitted ')' only causes a warning. Sometimes there is insufficient redundancy in the text to allow a sensible attempt at recovery to be made. In these cases and omitted ')' causes an error.

E7: missing, or invalid, expression

E8: invalid options

E9: missing '"'
A quoted string has not been closed before the end of the line (erronious attempt to enclose the newline character within a string).

E10: type conflict
An attempt has been made to use a Tag for two entirely conflicting purposes, for example as a signal name and as a macro name (in a DEFINE statement).

E11: too many pins

E12: too few pins

E13: string too long
The string exceeds 255 characters in length.

E14: missing END
This message may also be indicative of other errors that have caused the compiler to become lost.

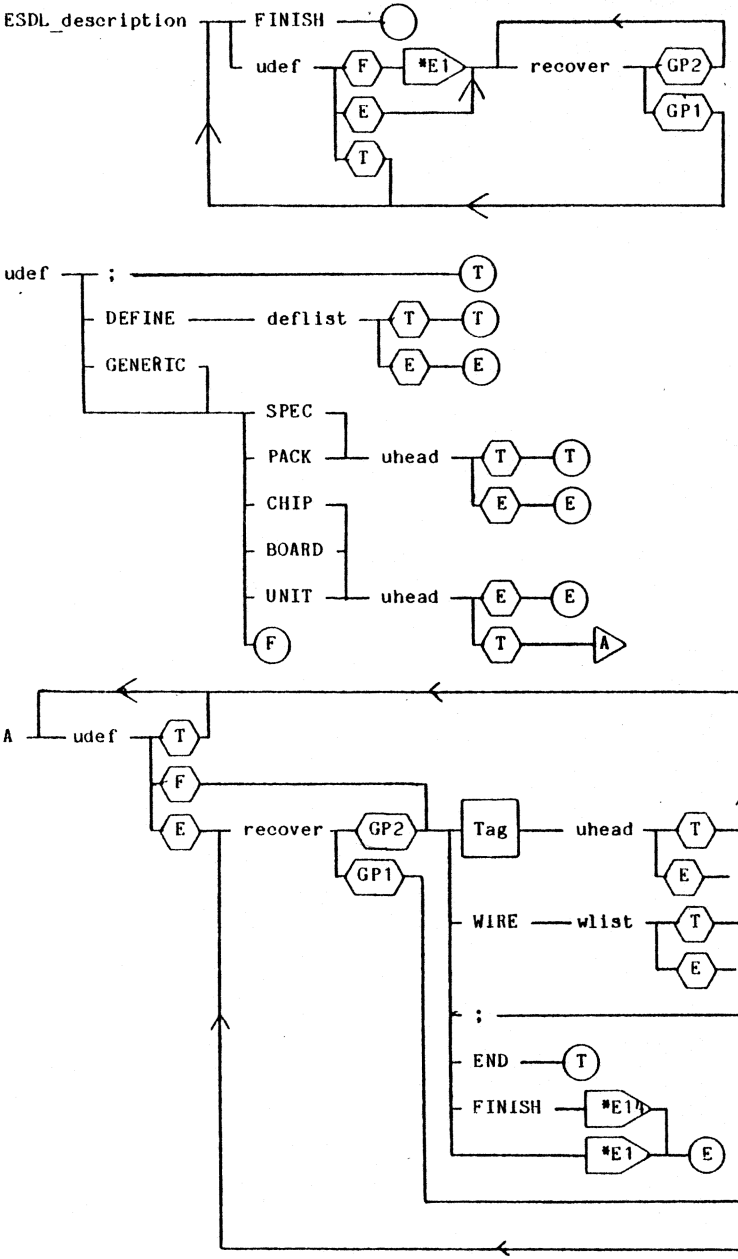
Disasters

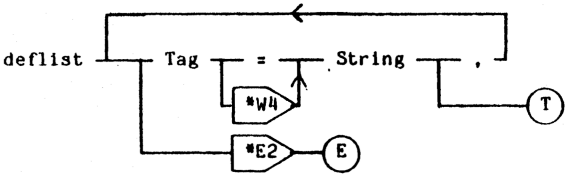
A disaster is an occurrence that causes processing to be terminated immediately.

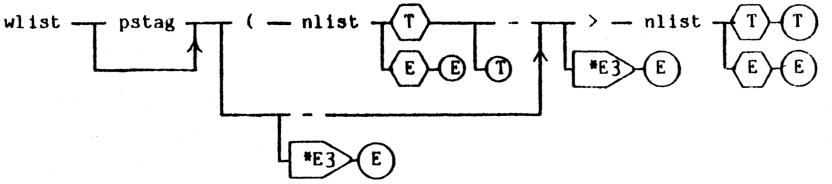
D1: workspace full
The implementation dependent workspace limit has been exceeded. Either have the limit increased (by the maintainer of the system) or move to an implementation which allows a larger limit.

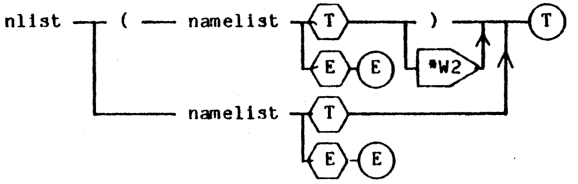
D2: too many errors
The compiler has become hopelessly lost. Fix some of the errors and re-submit the compilation.

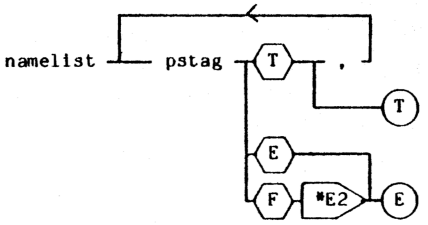
D3: too many levels of DEFINE
Macro definitions have been nested beyond the implementation dependent limit (currently 9 on all implementations). See the system maintainer to have the limit increased. Note that the existence of a limit provides some protection against certain classes of internal compiler error.

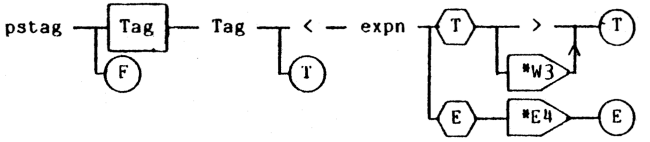

ESDL_description
FINISH
udef
F
*E1
recover
GP2
GP1
E
T
udef
;
T
DEFINE
deflist
T
T
E
E
GENERIC
SPEC
PACK
uhead
T
T
E
E
CHIP
BOARD
UNIT
uhead
E
E
T
A
F
A
udef
T
F
E
recover
GP2
GP1
Tag
uhead
T
E
WIRE
wlist
T
E
;
END
T
FINISH
*E1
*E1
E

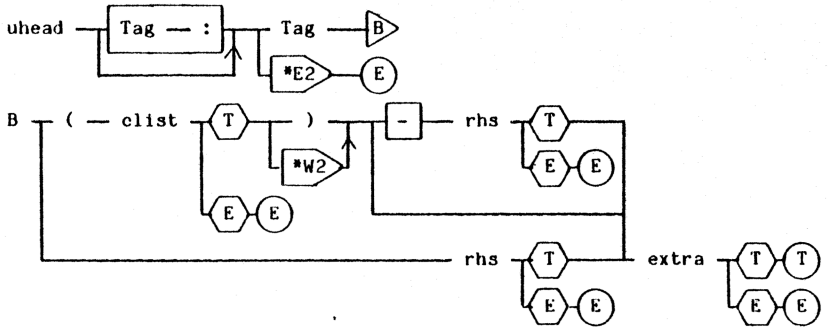

deflist
Tag
=
String
,
*W4
T
*E2
E

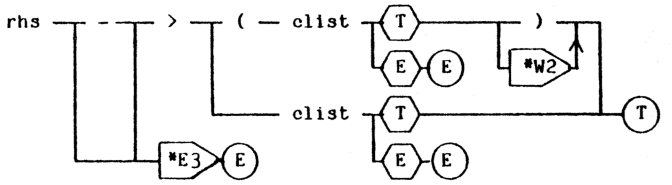

wlist
pstag
(
nlist
T
E
E
T
-
>
nlist
T
T
E
E
*E3
E
-
*E3
E

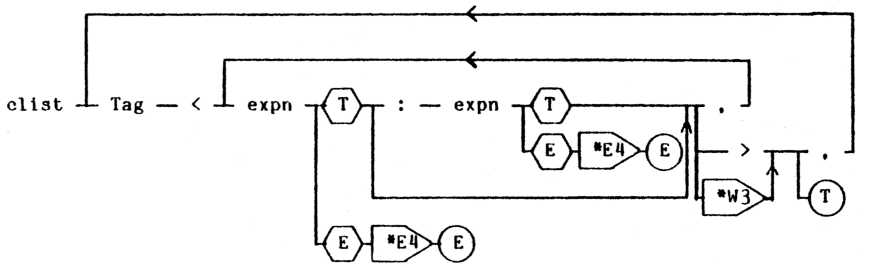

nlist
(
namelist
T
E
E
)
T
*W2
namelist
T
E
E

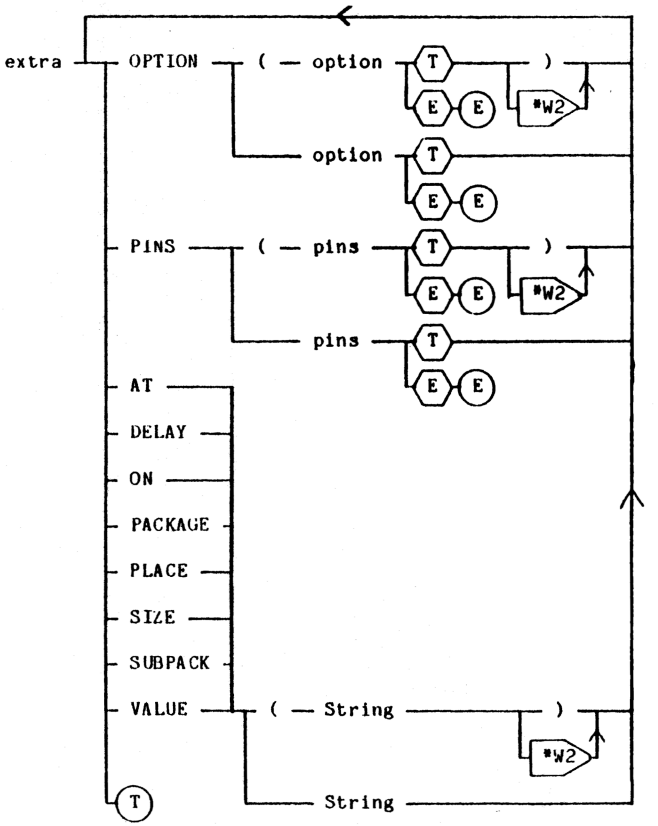

namelist
pstag
T
,
T
E
F
*E2
E

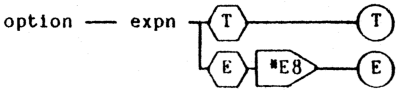

pstag
Tag
F
Tag
<
T
—
expn
T
>
T
*W3
E
*E4
E

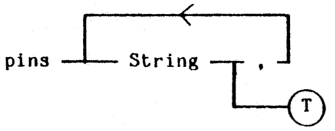

uhead
Tag — :
Tag
B
*E2
E
B
(— clist
T
)
*W2
E
E
-
rhs
T
E
E
rhs
T
E
E
extra
T
T
E
E


rhs
-
>
(— clist
T
E
E
)
*W2
clist
T
E
E
T
*E3
E


clist
Tag — <
expn
T
: — expn
T
E
*E4
E
.
>
.
*W3
T
E
*E4
E


extra
OPTION
(
option
T
E
E
)
*W2
option
T
E
E
PINS
(
pins
T
E
E
)
*W2
pins
T
E
E
AT
DELAY
ON
PACKAGE
PLACE
SIZE
SUBPACK
VALUE
(
String
)
*W2
String
T


option
expn
T
T
E
*E8
E


pins
String
,
T

Appendix 2

Pre-definitions for ESDL

The following pre-definition file is read by the ESDL compiler before processing input text. It comprises a list of reserved words (each preceded by a '*' and followed by the numeric value which is used internally by the compiler to represent the token), a list of pre-defined numeric values (such as NOSIGNALS), and a list of SPECs for logic gates.

```
*DEFINE 1
*GENERIC 2
*SPEC 3
*UNIT 4
*CHIP 5
*BOARD 6
*PACK 7
*END 8
*WIRE 129
*FINISH 11
*OPTION 13
*PINS 14
*AT 15
*ON 16
*PACKAGE 17
*SUBPACK 18
*DELAY 19
*VALUE 20
*SIZE 21
*PLACE 22
*COPTION 27
*LISTON 28
*LISTOFF 29
*GENERATE 30
*NOGENERATE 31
DEFINE DUMP1=1,DUMP2=2,DUMP3=4
DEFINE NOEXPAND=1
DEFINE FORGET=8,STRCONVERT=16,PUTSPECS=32,NOSIGNALS=64
GENERIC SPEC NAND(?,?)->?      GENERIC SPEC NOR(?,?)->?
GENERIC SPEC AND(?,?)->?       GENERIC SPEC OR(?,?)->?
GENERIC SPEC NOT(?)->?         GENERIC SPEC INV(?)->?
GENERIC SPEC AMP(?)->?
GENERIC SPEC XOR(?,?)->?       GENERIC SPEC XNOR(?,?)->?
GENERIC SPEC WOR(?,?)->?       GENERIC SPEC WAND(?,?)->?
GENERIC SPEC NAND(?,?,?)->?    GENERIC SPEC AND(?,?,?)->?
GENERIC SPEC OR(?,?,?)->?      GENERIC SPEC NOR(?,?,?)->?
GENERIC SPEC WOR(?,?,?)->?     GENERIC SPEC WAND(?,?,?)->?
GENERIC SPEC NAND(?,?,?,?)->? GENERIC SPEC AND(?,?,?,?)->?
GENERIC SPEC NOR(?,?,?,?)->?  GENERIC SPEC OR(?,?,?,?)->?
GENERIC SPEC NAND(?,?,?,?,?,?,?,?)->?
GENERIC SPEC AND(?,?,?,?,?,?,?,?)->?
GENERIC SPEC NOR(?,?,?,?,?,?,?,?)->?
GENERIC SPEC OR(?,?,?,?,?,?,?,?)->?
;
LISTON
```

Appendix 3

A formal definition of I-code

In this appendix a semi-formal definition of the ESDL intermediate code (I-code) is given. This is the particularly simple form into which ESDL is translated by the ESDL compiler.

Within I-code all numbers are decimal, and all strings are preceded by their lengths. I-code is 'punctuated' by control characters, each of which consists of '^' followed by an upper case letter (recall that '^' was not allowed within a String, and was translated to '|' - this prevents the accidental occurrence of control characters). Newlines can occur freely in I-code and are ignored, however, a newline must not occur in the middle of a number, nor may it separate a '^' from its following letter. An I-code comment (a String preceded by ^K) may occur immediately before any control character (including ^K itself). Strings have a maximum length of 255 characters.

In the definition which follows { and } are used as meta brackets to denote that an item is optional. If the } is followed by a * then it may be repeated any number of times (including 0 times). A } followed by *n denotes that the enclosed item is to be repeated exactly n times. D: denotes that the following item is a decimal number, and S: denotes that the following item is a String. A single significant space is denoted by the character @.

```
I-code          ::= {{^S D:flag} unit}*

unit            ::= ^U D:type unit-header {unit}* {unit-body} ^E

unit-body       ::= ^J D:nsubs {sub-instance}*nsubs {connection-net}*

sub-instance    ::= unit-header

unit-header     ::= ^H D:options @ D:nin @ D:nout @ D:nio @D:nt @
                      S:label S:unit-name {terminal}*nt {parameter}* ^G

terminal        ::= ^T D:tflags @ S:pin-name S:signal-name

parameter       ::= ^P D:parameter-number @ S:parameter-string

connection-net ::= ^N {^A S:net-name D:fan {@ D:subno @ D:tno}*fan}*

comment         ::= ^K S:comment-text

S:string        ::= D:length : {any-char-but-^-and-NL}*length
```

The various decimal numbers which appear in the above definition all have conventional values or interpretations, and these are given below.

flag currently this is unused.

type type takes the value 1 for a SPEC, 2 for a UNIT, 3 for a CHIP, 4 for a BOARD, and 5 for a PACK. A further 8 is added if the unit header is GENERIC (i.e. GENERIC SPEC corresponds to a value of 9).

nsubs is the number of sub-instances in the unit body.

nin, nout, nio, nt are the number of inputs, outputs, input-outputs, and the total number of signal names in the unit header.

options is interpreted as a series of flags. Currently only the least significant bit is interpreted. If this bit is set then it is equivalent to OPTION NOEXPAND on the unit or instance header, and FLATTEN will not expand this instance (or all instances if the bit is set in a unit header).

tflags consists of two fields. The least significant two bits contain a flag: 1 for an input, 2 for an output and 3 for an input-output. The rest of the integer contains the effective terminal number. This number is 1 for the first input to the unit or instance header, and is incremented for each signal in the input list, and for each signal in the output list that is not an input-output. For example the header (A,B,C)->C,B,D has effective terminal numbers 1, 2, 3, 3, 2, and 4.

pno takes the value 1 for AT, 2 for ON, 3 for PACKAGE, 4 for SUBPACK, 5 for DELAY, 6 for VALUE, 7 for SIZE, and 8 for PLACE. Most programs of the DL1 suite currently assume that pno will not exceed 10. Some assume the limit of 8.

fan fan is the number of terminals in a connection net fragment. Note that a connection net can consist of a number of fragments, each with a separate name (caused by WIRE and the use of input-outputs).

subno and tno are the number of the subinstance and the effective terminal number (see 'tflags' above) of a terminal in the connection net fragment. Subinstances are numbered from 1 in the order in which they occur in the ESDL source text. Conventionally subinstance 0 is the unit definition header.

Appendix 4

Use of ESDL to describe chips

In this appendix a number of CHIP definitions are given as examples of the use of ESDL. Normally such definitions would be used to create a chip library for use by the ASSIGNment program (see separate documentation).

```
CHIP 8085A(READY,HOLD,INTR,RST55,RST65,RST75,TRAP,NRESETI,SID,X1,
      X2,.VCC,.GND)->AD<0>,AD<1>,AD<2>,AD<3>,AD<4>,AD<5>,AD<6>,
      AD<7>,A<8>,A<9>,A<10>,A<11>,A<12>,A<13>,A<14>,A<15>,ALE,S0,S1,
      NRD,NWR,HLDA,NINTA,RESETO,CLK,IONM,SOD
      PINS(35,39,10,9,8,7,6,36,5,1,2,40,20,12,13,14,15,16,17,18,
      19,21,22,23,24,25,26,27,28,30,29,33,32,31,38,11,3,37,34,4)
      ON "DIL40"
   SPEC U8085A(A,B,C,D,E,F,G,H,?,?,?,?,?,?,?,?,?,?,?,?)->A,B,C,D,E,F,
        G,H,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?,?
   U8085A(AD<0>,AD<1>,AD<2>,AD<3>,AD<4>,AD<5>,AD<6>,AD<7>,READY,
        HOLD,INTR,RST55,RST65,RST75,TRAP,NRESETI,SID,X1,X2)->AD<0>,
        AD<1>,AD<2>,AD<3>,AD<4>,AD<5>,AD<6>,AD<7>,A<8>,A<9>,A<10>,
        A<11>,A<12>,A<13>,A<14>,A<15>,ALE,S0,S1,NRD,NWR,HLDA,NINTA,
        RESETO,CLK,IONM,SOD
END

CHIP 8212(DI<1>,DI<2>,DI<3>,DI<4>,DI<5>,DI<6>,DI<7>,DI<8>,NDS1,
      DS2,MD,STB,NCLR,.VCC,.GND)->DO<1>,DO<2>,DO<3>,DO<4>,DO<5>,
      DO<6>,DO<7>,DO<8>,NINT
      PINS(3,5,7,9,16,18,20,22,1,13,2,11,14,24,12,4,6,8,10,15,17,
      19,21,23)
      ON "DIL24"
   U8212(DI<1>,DI<2>,DI<3>,DI<4>,DI<5>,DI<6>,DI<7>,DI<8>,NDS1,DS2,
        MD,STB,NCLR)->DO<1>,DO<2>,DO<3>,DO<4>,DO<5>,DO<6>,DO<7>,
        DO<8>,NINT
END

CHIP T74126(I1,E1,I2,E2,I3,E3,I4,E4,.VCC,.GND)->O1,O2,O3,O4
      PINS(1,2,4,5,8,9,11,12,7,14,3,6,10,13)
      ON DIL14
   U74126(I1,E1)->O1
   U74126(I2,E2)->O2
   U74126(I3,E3)->O3
   U74126(I4,E4)->O4
END

CHIP T74257(A<0>,A<1>,A<2>,A<3>,B<0>,B<1>,B<2>,B<3>,STROBE,SELECT,
      VCC,.GND)->C<0>,C<1>,C<2>,C<3>
      PINS(2,5,11,14,3,6,10,13,15,1,16,8,4,7,9,12)
      ON DIL16
   U74257(A<0>,A<1>,A<2>,A<3>,B<0>,B<1>,B<2>,B<3>,STROBE,SELECT)->
        C<0>,C<1>,C<2>,C<3>
END

CHIP T8T13(A1<1>,A1<2>,A1<3>,A1<4>,A1<5>,A1<6>,A2<1>,A2<2>,A2<3>,
      A2<4>,A2<5>,A2<6>,.VCC,.GND)->Q1,Q2
      PINS(1,2,3,4,5,6,10,11,12,13,14,15,16,8,7,9)
      ON DIL16
   U8T13(A1<1>,A1<2>,A1<3>,A1<4>,A1<5>,A1<6>)->Q1
   U8T13B(A2<1>,A2<2>,A2<3>,A2<4>,A2<5>,A2<6>)->Q2
END
```

Appendix 5

Use of ESDL to define slotted boards

In this appendix a fragment of an ESDL board definition is presented. Normally such definitions are used to create libraries of standard boards for the PLACEment program (see separate documentation). Note how the signal names of the BOARD header are used to specify edge connector names, and how the PINS statement is used to associate a coordinate pair (representing the edge connector position) with each edge connector name. Slots for packages are named by means of a label on the instance of the package type (e.g. A1:DIL14 ... to represent a DIL14 slot called A1) and the position of the slot is given by means of a coordinate pair (conventionally the coordinates of of the corner of the chip nearest to pin 1, measured on the wire side of the board) as the AT parameter. Any pre-wired signals (such as power and ground planes) are specified as signal names of the slot instance, and associated with the appropriate pins by means of the PINS statement. The size of the whole board is given by the SIZE parameter (conventionally measured on the wire-side of the board using right-handed axes).

```
COPTION NOSIGNALS
BOARD CSD083(
AA1,AB1,AC1,AD1,AE1,AF1,AH1,AJ1,AK1,AL1,AM1,AN1,AP1,AR1,AS1,AT1,AU1,AV1,
AA2,AB2,AC2,AD2,AE2,AF2,AH2,AJ2,AK2,AL2,AM2,AN2,AP2,AR2,AS2,AT2,AU2,AV2,
BA1,BB1,BC1,BD1,BE1,BF1,BH1,BJ1,BK1,BL1,BM1,BN1,BP1,BR1,BS1,BT1,BU1,BV1,
BA2,BB2,BC2,BD2,BE2,BF2,BH2,BJ2,BK2,BL2,BM2,BN2,BP2,BR2,BS2,BT2,BU2,BV2)
      SIZE 850:525
PINS(95:508,95:496,95:483,95:471,95:458,95:446,95:433,95:421,95:408,
95:396,95:383,95:371,95:358,95:346,95:333,95:321,95:308,95:296,
115:508,115:496,115:483,115:471,115:458,115:446,115:433,115:421,115:408,
115:396,115:383,115:371,115:358,115:346,115:333,115:321,115:308,115:296,
95:232,95:220,95:207,95:195,95:182,95:170,95:157,95:145,95:132,
95:120,95:107,95:95,95:82,95:70,95:57,95:45,95:32,95:20,
115:232,115:220,115:207,115:195,115:182,115:170,115:157,115:145,115:132,
115:120,115:107,115:95,115:82,115:70,115:57,115:45,115:32,115:20)

A1:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 155:500
A1:DIL14(.GND,.VCC) PINS(7,14) AT 155:500
A2:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 155:440
A2:DIL14(.GND,.VCC) PINS(7,14) AT 155:440
A3:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 155:380
A3:DIL14(.GND,.VCC) PINS(7,14) AT 155:380
A4:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 155:320
A4:DIL14(.GND,.VCC) PINS(7,14) AT 155:320
A5:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 155:240
A5:DIL14(.GND,.VCC) PINS(7,14) AT 155:240
A6:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 155:180
A6:DIL14(.GND,.VCC) PINS(7,14) AT 155:180
A7:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 155:120
A7:DIL14(.GND,.VCC) PINS(7,14) AT 155:120
A8:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 155:60
A8:DIL14(.GND,.VCC) PINS(7,14) AT 155:60
B1:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 245:500
B1:DIL14(.GND,.VCC) PINS(7,14) AT 245:500
B2:DIL16(.GND,.GND,.VCC) PINS(7,8,16) AT 245:440
```

Appendix 6

The use of ESDL to define packages

In this appendix it is shown how ESDL can be used to define the geometry of physical packages. Normally such descriptions are used to create a library of standard packages which is used by the PLACEment program (see separate documentation). Note how the signal names of the unit definition are used to name the package's pins, and how the PINS statement is used to associate a coordinate pair with each pin (conventionally the coordinates are the offsets of the pins from the corner of the package nearest to pin 1, measured in right handed axes on the wire side of the board, in units of a hundredth of an inch).

```
UNIT DIL4(1,2,3,4)
 PINS(5:0,15:0,15:-30,5:-30)
 SIZE "20:-30"
END

UNIT DIL8(0,2,3,4,5,6,7,8)
 PINS(5:0,15:0,25:0,35:0,35:-30,25:-30,15:-30,5:-30)
 SIZE "40:-30"
END

UNIT DIL14(0,2,3,4,5,6,7,8,9,10,11,12,13,14)
 PINS(5:0,15:0,25:0,35:0,45:0,55:0,65:0,65:-30,
      55:-30,45:-30,35:-30,25:-30,15:-30,5:-30)
 SIZE "70:-30"
END

UNIT DIL16(0,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16)
 PINS(5:0,15:0,25:0,35:0,45:0,55:0,65:0,75:0,
      75:-30,65:-30,55:-30,45:-30,35:-30,25:-30,15:-30,5:-30)
 SIZE "80:-30"
END

UNIT DIL18(0,2,3,4,5,6,7,8,9,
           10,11,12,13,14,15,16,17,18)
 PINS(5:0,15:0,25:0,35:0,45:0,55:0,65:0,75:0,
      85:0,85:-30,75:-30,65:-30,55:-30,45:-30,35:-30,25:-30,
      15:-30,5:-30)
 SIZE "90:-30"
END

UNIT DIL20(0,2,3,4,5,6,7,8,9,10,
           11,12,13,14,15,16,17,18,19,20)
 PINS(5:0,15:0,25:0,35:0,45:0,55:0,65:0,75:0,
      85:0,95:0,95:-30,85:-30,75:-30,65:-30,55:-30,45:-30,
      35:-30,25:-30,15:-30,5:-30)
 SIZE "100:-30"
END

UNIT DIL24(0,2,3,4,5,6,7,8,9,10,11,12,
           13,14,15,16,17,18,19,20,21,22,23,24)
 PINS(5:0,15:0,25:0,35:0,45:0,55:0,65:0,75:0,
      85:0,95:0,105:0,115:0,115:-60,105:-60,95:-60,85:-60,
      75:-60,65:-60,55:-60,45:-60,35:-60,25:-60,15:-60,5:-65)
 SIZE "120:-60"
END
```

Alphabetical Index