cnet Centre National d'Etudes des Télécommunications - Grenoble - France

CSELT Centro Studi e Laboratori Telecomunicazioni - Torino - Italy

fi Forschungsinstitut der Deutschen Bundespost beim FTZ - Darmstadt - Germany

"CAD-VLSI for TELECOMMUNICATIONS"

(CVT - Project)

KARL III INSTANT

Univ. Kaiserslautern

August 1984

This document must not be distributed to parties other than participants to or reviewers of the CVT Project

Ref.:
Council (EEC) Regulation N° 3744/81 of 7th December 1981, Official Journal of the European Communities N° L 376 of 30 December 1981

K A R L - III

I N S T A N T

Version no. 1

June 1984

Kaiserslautern University, Fachbereich Informatik (Computer Science Department), Kaiserslautern, June 1984; Address: Postfach 3049, D - 675 Kaiserslautern, Fed. Rep. of Germany. phone: ..49 (631) 205 . 2606; telex: 04 . 5627 UNIKL D

c o n t e n t s

1. introduction
1.1 structure of the KARL-III software package
1.2 KARL-III data path formatting
1.2.1 data path format conventions
1.2.2 catenation and abutment constructors
1.2.3 replicators
1.3 parameters
1.4 personality matrixes

2. KARL data operators and standard functions
2.1 data operators
2.2 standard functions
2.2.1 operative standard functions
2.2.2 wiring standard functions
2.2.3 RAM and ROM standard functions

3. SCIL (Simulation control and input language)
3.1 SCIL syntax meta notation
3.2 SCIL simulation command instant
3.3 SCIL I/O command instant

4. Terminology, values and codes of bits and words
4.1 values and types of bits
4.2 word types and terminology
4.3 modelling of bus circuits and connections

5. KARL - II grammar

KARL - III
INSTANT

Version no. 1

June 1984

Kaiserslautern University, Fachbereich Informatik (Computer Science Department), Kaiserslautern, June 1984; Address: Postfach 3049, D - 675 Kaiserslautern, Fed. Rep. of Germany. phone: ..49 (631) 205 . 2606; telex: 04 . 5627 UNIKL D

1. Introduction

This KARL-III instant describes two languages: KARL-III source language (sections 1.2 and chapters 2, 4, and 5) for hardware description, and the SCIL language for simulator activation, stimuli supply, and logging of responses. Fig. 1 gives a survey of the structure of the software package. The executable form, called RTcode, is described in detail somewhere else /WE881/. More details on KARL are found in /LIE81/ and /LIE82/.

1.1 Structure of the KARL-III software package

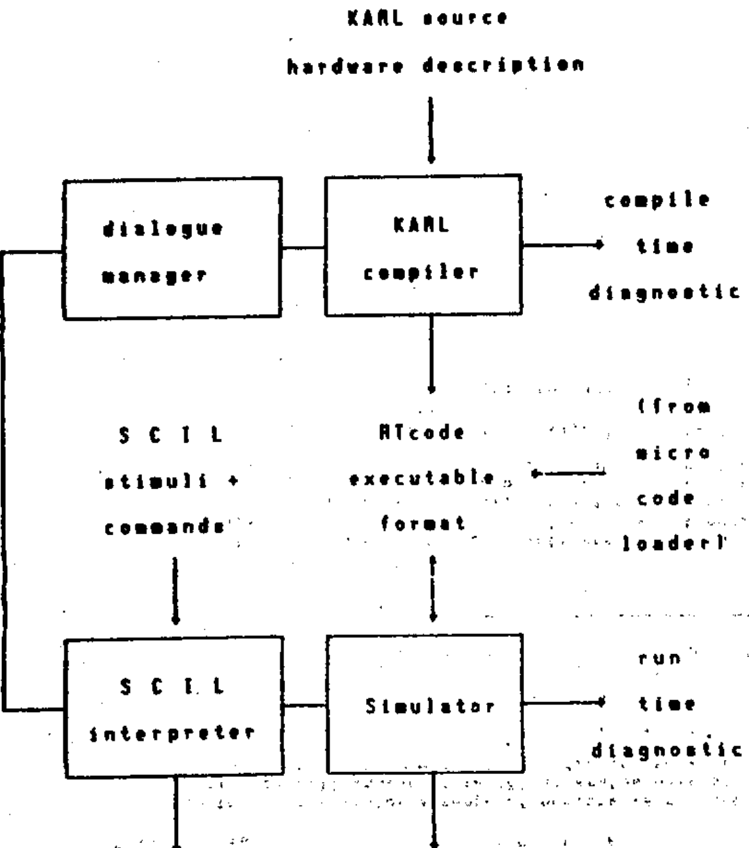


Fig. 1. Structure of the KARL-III software package

1.2 KARL-II data path formatting conventions and operators

1.2.1

DATA PATH FORMAT CONVENTIONS

Normally a data path does not change width: All input and output paths of an operator, function, or, connection have the same width, if not indicated otherwise; exceptions:

- relational operators and test functions have a one bit wide output

- other exceptions: see multiply operators and code conversion functions

"indicated otherwise" means the use of special language primitives to describe change of path width:

- data catenation: "." for words and " | " for arrays

- function slice abutment ":"

- selection by means of [...] , or by means of if or case

1.2.2

CATENATION AND ABUTMENT CONSTRUCTORS

For catenation of words the following constructors may be used:

. word catenation! example: AWORD . BWORD

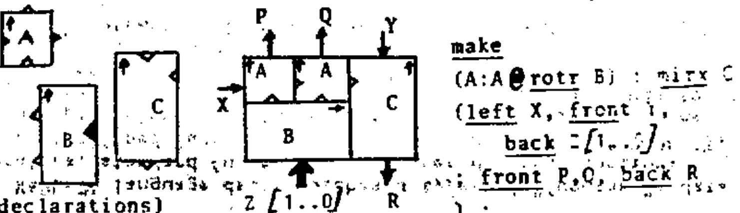
| array catenation! example: XARRAY | YARRAY

For rows of slices, or, columns of layers the abutment constructor may be used:

: slice abutment! example: S : T

@ layer abutment! example A @ B

The ports of abutting sides are automatically connected if matching.



1.2.3 replicators

A replicator '&' may be appended to catenators '!' or '.' , to abutment constructors ':' or '@', or, to a comma ',' . In this context a decimal number n behind the '&' specifies the number of times the operand is used. The operator is used n-1 times. Example using slice replicator ':&' :

F :&4 is equivalent to F : F : F : F

By appending a range to the replicator a range of numbers is be generated to be used for numbering of cell instances (a '&' separates name and number) or for modification of names by numerical appendix (no separation). Examples:

F :&4 [3..0] is equivalent to F&3 : F&2 : F&1 : F&0

X ,&5 [1..5] is equivalent to X1, X2, X3, X4, X5

Range legal with ':&', '@&' ('&' generated), '.&', '!&', ',&' (no '&' gen.)

1.3 parameters

KARL-III has as feature to assign values to path width specification and other specs at instantiation time instead of declaration time. At declaration time parameters or parameter expressions are used, the constant values of which are computed at instantiation time. So only one description of a cell is needed in a library, if several versions with different path width are used.

parameter expressions (pe) with: pe ::= ({identifier | constvalue} // {&+ | &- | &* | &/})+

no priorities: the operators are evaluated from left to right

Parameter expressions may be used behind replicators (see 1.2.3), and behind '!&', '..&', '&&', and '&_' ('&' are deleted after param evaluation). Examples:

REG [&UPP ..&LOW]	evaluated to	REG [7..0]
XYNAME&_APP	evaluated to	XYNAME_27
NAME&&APDX	evaluated to	PDNAME12

where UPP = 7, LOW := 0, APP = 27, and, APDX = 12.

1.4 personality matrixes (also see instant grammar, last page)

Personality matrixes may be processed and converted into boolean expressions for the following structures: PLAs, PALs (only a PLA AND plane), POLs (only PLA OR plane), and Kolte arrays (useful for Petri net descriptions and similar concepts). Values of matrix entries are for PLA AND plane, PAL and POL:

0 negated input 1 input - not connected x,X don't care

For the OR plane of a PLA only the values '1' and '-' are legal. A Kolte array is derived from a PLA by cutting off the path of the term vector (so that a term input vector and a term vector output is accessible) and then folding both planes into each other. Kolte array values have the following meanings:

0	negated condition	s,S	source place	-	not
1	condition	d,D	destination place		connected

The line subscript is the number of a transition. Each place and each condition input is represented by a column of the Kolte matrix.

2. KARL data path formatting conventions and operators

The KARL-III language distinguishes 3 types of "functors": data operators, standard functions, and, user-defined functions. The user-defined functions (see expression grammar of KARL-III) are not parameterized with respect to data path width in the current version of the KARL-III package (the KARL user has to write another version of a module, in order to have it available in a different data path width. However, operators and standard functions automatically adopt the width of the data paths it is connected to.

2.1.1 KARL data operators and standard functions

DATA OPERATORS AND STANDARD FUNCTIONS

All operators are dyadic and are used in an infix notation. example:

A + B

All standard functions are monadic and are used in functional notation. e. g.: not (X)

In addition to an operand (e. g. X) a standard function may have an integer parameter. example: fold&4 (INPUT) , where INPUT is the operand, 4 is the parameter, fold is the the standard function's name, & is separator.

2.1.2 data operators

DATA OPERATORS

All operators in KARL are dyadic. Both inputs have same width; exceptions: / and mod (see there). Output width is equal input width; exceptions: * (double width output) and relational operators (1-bit output).

dyadic logical operators:		and or nand nor exor coin
relational operators:		=, =/, <, >, =<, >=
adding operators:		+, -
multiply operators:	*	multiply (double width output)
	/	divide (one input has double width)
	mod	remainder (one input double width)

2.2 standard functions

Most standard functions have an integer function parameter, which is appended to the function name by a "&" connective (without any blanks around the "&"). All standard fuctions are monadic and its operand is inside parentheses behind function name and parameter (if any).

2.2.1 operative standard functions

OPERATIVE STANDARD FUNCTIONS

All are monadic functions; input and output have same path width; exceptions: test functions (1-bit output) and code conversion functions (see there), and sideward paths (if any).

CODE CONVERSION FUNCTIONS (changing path width):

decode	input interpreted as a binary coded integer number i output is singulary code (1-of-n code) of i
encode	if singulary input then inverse of decode function, else ???... output
decount	input interpreted as a binary coded integer number i output is unary code of i (i ones right adjusted)
encount	inverse of decount function, if legal input, else ???..
unary	synonym of decount
count	counts and returns number of "1"s in arbitrary input

TEST FUNCTIONS (1 bit wide output):

odd	check, if odd parity
even	check, if even parity
sing	check, if singulary code
unary	check, if unary code
msb	check most significant bit
lsb	check least significant bit
equ(&i)	check, if equal to constant i (default i is 0)

MISCELLANEOUS FUNCTIONS (no change of path width):

inc(&i)	increment by constant value i (default i is 1)
dec(&i)	decrement by constant value i (default i is 1)
pril	priority from left (singulary outp: if non-empty input)
prir	priority from right (singulary outp if non-empty input)
not	inverts all bits

2.2.2 wiring standard functions

WIRING STANDARD FUNCTIONS

all are monadic functions, and, input and output have same width; exceptions: sideward paths

SHIFT FUNCTIONS: (default i is 1)

(x)shy(&i) ciy(&i)	shift by i bits, y ::= l/r, x ::= c/d/n/e. circular shift by i bits, y ::= l/r, where r means "shift to the right" and l means "shift to the left"
cir, cil	circular shift
shr, shl	serial shift in from side path (not impl'ted)
cshr, chrl	"constructive": i one-bits shifted in
dshr, dshl	"destructive": i zero-bits shifted in
nshr, nshl	"non-destructive": i unused bits directly connected to source side
eshr eshl	"extended sign": similar to dshr, however, most significt. bit directly from source side of the path
push pop	shift to the left by i words ("word shift") shift to the right by i words ("word shift")
npush npop	"non-destructive": i unused words directly from source side of the path (not impl'ted)

SHUFFLE FUNCTIONS: (default i is 2)

fold(&i)	"word fold": rearrange destination bit sequence such, that i is destination step width (example with i=4 in 12 bit path: destination bits 11,7,3,10,6,2,9,5,1,8,4,0 connected to source bits 11,10,9,...2,1,0) i=0 is illegal
merge(&1)	"array fold": see fold, however, sequence of words within an array is folded i=0 is illegal

MIRROR FUNCTIONS: (no parameter i)

reflect	reverses sequence of bits within a word
reverse	reverses sequence of words within an array

To save storage space used for RT code a second notation for the description of RAMs and ROMs has been introduced as new standard functions. The values (e. g. microcode) may be read from a file into microcode under dialogue manager guidance. The format is illustrated by the following synopses:

rom[&i] (address) returns stored value

ram[&i] (address . write-enable ; input-name)

Currently data path width is 32 bits. Use "[...]" or "." at input and output to model memory word sizes other than 32

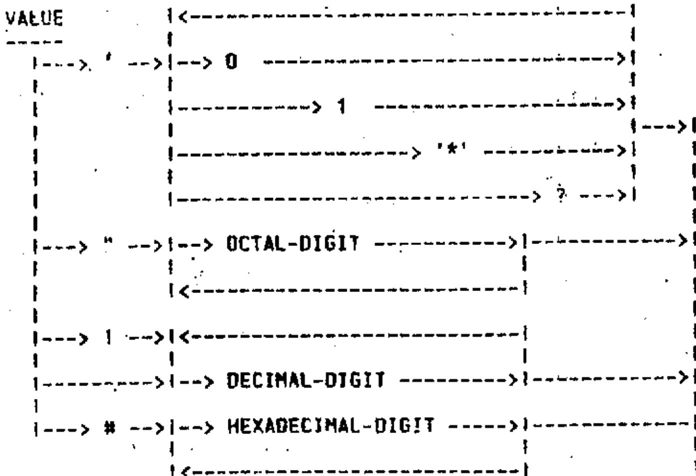
3. SCIL (simulation control and input language)

3.1 SCIL syntax meta notationsa

The following characters are used for syntax meta notation only (do not use for personalizing from syntax rules):

::=   [ ]   //   /   +   *

```
examples: [ .... // .... ]+ means: ..... 1 or more times,
          and separated by .... if more than once
          [ .... // .... ]* see above, however, 0 or more times
          [ .... ]   means: .... is optional (0 or 1 times)
          [ .... ]1  means: .... appears exactly once
          .....  /  .....   means:  .....  or ........
```



also see lexical grammar of KARL-II

S C I L SIMULATOR COMMAND INSTANT

EXECUTION COMMAND:

STEP   execute one micro step

NAME SYNTAX: (i is instant no.)

name ::= [ident[&i] // .]+

ASSIGNMENTS:

executed by next STEP command

```
INIT  value                all switches, registers, and delays are set to this value
ASSIGN [ name value // , ]+    assignment of values to words
XFERTO [ name1 name2 // , ]+   assign to (1) the value of (2) (not implemented)
SEQ [ name i [ value ]+ // , ]+ -> (ASSIGN // i times STEP )+ (not implemented)
SET name ( [ bitnumber [ .. bitnumber ] // , ]+ )        set specif. bits to 1
RESET name  (  [ bitnumber [ .. bitnumber ] // , ]+ )    set these bits to 0
INVERT name  (  [ bitnumber [ .. bitnumber ] // , ]+ )   invert specified bits
LOCK  [ name // , ]+          freeze values til encounter of RELEASE command
KEEP  [ name value // , ]+    overriding freeze til encounter of RELEASE
RELEASE [ name ]+   or:   REL [ name ]+              see KEEP and LOCK
```

BLOCK DEFINITIONS

never use DEF of ENDDEF within loops !

```
LOOP n [ command ]+  ENDLOOP                  loop n times the command sequence
ENDLOOP see LOOP
REPEAT [ non-DEF-command ]+ UNTIL condition           (test at end of loop)
UNTIL  see REPEAT
WHILE condition [ command ]+ ENDWHILE                 (test in front of loop)
ENDWHILE  see WHILE
DEF name [ command ]+ ENDDEF   define command macro to be evoked later by name
ENDDEF                         see DEF
DELETE               delete all commands back til most recently executed one
ESCAPE               see "DELETE" (changed to "DELETE")
SKIP                 jump to first command behind end of outermost block
```

CONDITIONAL COMMANDS

IS is different from "=" !

```
IF condition THEN [ command ]+ [ ELSE [ command ]+ ]  ENDIF    self-explanatory
condition  ::=  name  [ IS / ISNOT ]1  value    see syntax diagram of 'value'
ISNOT                                           result bit is inverted IS result
IS                      result bit is 1 only for operand bit pairs: (0, 0),
                        (1, 1), (?, ?), (*, *), otherwise result bit is 0.
```

MULTIPLE OCCURRANCE OF IDENTIFIERS:

instance selection
also see above name syntax

```
IN name [ command ]+  ENDIN    all local names referenced from now til ENDIN are
                               inside the function module named after IN
ENDIN                 see IN      (IN .... ENDIN clauses may be nested)
```

3.3 SCIL I/O comand instant

S C I L I/O COMMAND INSTANT

PRINT COMMANDS:

PRINT name	most recent value printed together with name
PRNTACT	values of all those variables are printed, which changed during most recent STEP executi
PRNTRTC	output of RT code description
PRNTUNIT	see PRNTRTC, however, also most recent values sho
SENSE [name value // .]+	sense output, print "out error" if unequal to val (not implemente

PLOT COMMANDS:

PLOTALL [name ([bitnumber [.. bitnumber] // .]+) : size // .]+
plot-like print of all specified bit values continued after each STEP execution until encounter of ENDPLOT command

PLOTNEW see PLOTALL, however plot is suppressed after each STEP, wher all specified values remain unchanged

ENDPLOT see PLOTALL

INTERACTIVE FILE MANAGEMENT COMMANDS:

STDIN, STDOUT simulator input resp. output switched over to terminal
NEWIN, NEWOUT system enters dialogue to assign new file for input resp. outpu
OLDIN, OLDOUT switch back to input resp. output used before NEWIN or NEWOUT

INTERRUPTION AND TERMINATION OF SIMULATION RUN:

ENDSIM	termination of simulation run
RETURN	interrupt to enter KARL system to dialogue mode

MISCELLANEOUS COMMANDS:

SETTIME	start counting CPU time used
GETTIME	print CPU timer elapsed since SETTIME activatio
HELP	dump of command analyser state

COMMENTS:

COMMENT " ' " <text string> " ' " 'blank' comment in SCIL listing
WRITE see COMMENT, however, comment inserted in simulator output

4. terminology, values and codes of bits and words

4.1 values and types of bits

input bits other than 0, or, 1

Operators and functions normally expect neither * nor ? input bits, otherwise ???..... output ("undefined") might be produced. To be sure see truth table of operator or function. Some cases are quite simple, such as for instance the <u>and</u> operator with "0" response to the input pairs:

(0, 0), (0, 1), (0, *), and, (0, ?).

? value's normal meaning is: "no value has been assigned". In case of delay range declaration it also may mean: "it is undetermined, wether a transition has already occurred"

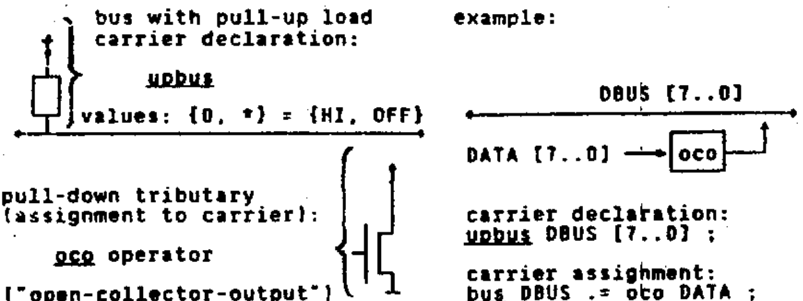
bit type	legal values of bit code
(default)	0, 1
input to upbus	0, *
input to downbus	*, 1
input to tribus	0, 1, *

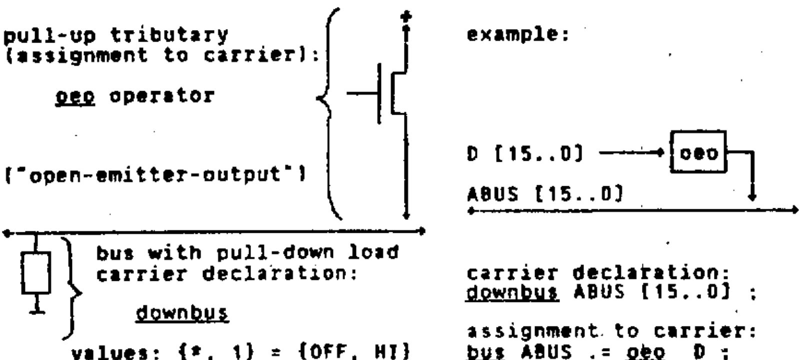
bit value	meaning
0	LOW, false
1	HIGH, true
*	OFF, high impedance state, don't care
?	undef., no value assigned, probability of transition

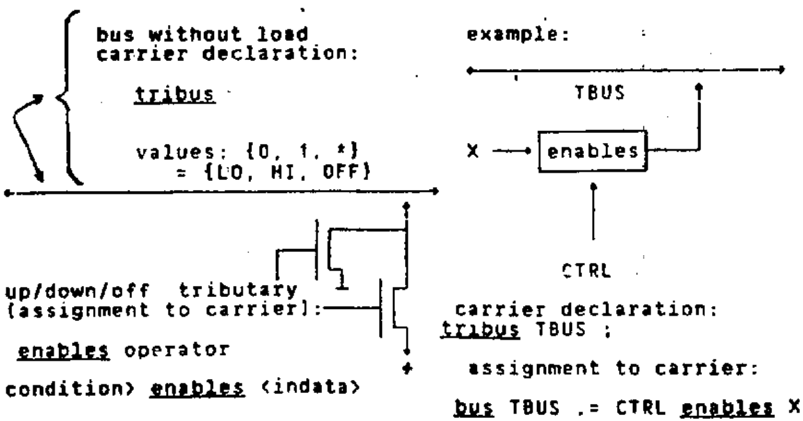
4.2 word types and terminology

word class	word type	bit pattern of words
numerical	binary	any
(only "0" and "1" bits permitted)	singulary	exactly one "1"
	multiple singulary	superposition of several singularies (AND of several singulary number codes)
	unary	all "1"s of word in a single contiguous block right side, all "0"s of word in a single contiguous block at left side.
"empty" words:	zero-word	all bits are "0"
	one-word	all bits are "1"
	star-word	all bits are "*"
	undef-word	all bits are "?"

4.3 modelling of bus circuits and connections







Literature

/1/ R.W. Hartenstein: Fundamentals of Structured Hardware Design, North Holland/A merican Elsevier, Amsterdam/New York, 1977

/2/ P. Haselmeier: The KARL language Compiler, report no.77-14, Istitutto di Elettronica, Politecnico di Milano, Milano, 1977.

/3/ P.A. Anderson, M. Nygaard: A study of different languages and tools for description and simulation of digital systems, report no. 6/78 The University of Trondheim, Div. of Computer Science, Trondheim Norway 1978

/4/ U. Hedengran: A Parser for KARLZ, international report, Dept.of Applied Electronics, The Royal Institute of Technology, Stockholm, Schweden 1980

/5/ R. W. Hartenstein: Basis of Structured Design Methologies: Data Paths and Finites State Machines, in (eds.:P.Jespers, C.Sequin) Design Methologies for VLSI, Noordhoff&Stijthoff, Alphen van Rijn, The Netherlands, 1981

/6/ R. W. Hartenstein: KARL-II eine Sprache zur Spezifikation beim Entwurf kundenspezifischer Digitalbausteine, Universität Kaiserslautern, Angewandte Informatik 12/82

/7/ R. W. Hartenstein; P. Liell: RTCode Instant Universität Kaiserslautern, FB-Informatik, 1983

/8/ P. Liell: Test Pattern Generation for Data Path using Iterative Arrays of Cells; Ph.D.dissertation (in German) Universität Kaiserslautern, FB-Informatik, 1983

/9/ H. Mirkes: Simulation of Petri nets using the KARL Language (in German Language) Universität Kaiserslautern, FB-Informatik, 1983

/10/ R. W. Hartenstein, K. Lemmert: KARL-III reference manual Universität Kaiserslautern, FB-Informatik, 1984

/11/ B. Borrmann; R. W. Hartenstein: superKARL-III Specification Universität Kaiserslautern, FB-Informatik, June 1984

/12/ N.N. : KARL-III Instant Universität Kaiserslautern, FB-Informatik, June 1984

/13/ N.N. : KARL-II hardware descriptions and simulation sessions: a collection of examples, in preparation: Universität Kaiserslautern, FB-Informatik, mid. 1984

/14/ G. Girardi: ABL editor: user manual (draft) Torino, March 1984

KARL-III Instant

declaration sequence:

- cell
- func
- constant
- rom
- switch
- clock
- node
- lightnode
- register
- ram
- delayer
- downbus
- tribus
- upbus

```
cell iden '(' [cellparlist ] ';'[cellparlist ]^2_0 ')' ';' body {';' / ';'} .

func iden rangespec '(' [ funcparlist ] ')' ';' body {';' / ';'} .

constant ( iden rangespec '.=' constval // ';' )+ ';' .

clock ( iden '[' dec_no ';' range ']' ';')+ .

delayer iden rangespec by dec_no [ to dec_no ] ';' .
```

DECLARATION PART

```
body ::= {external / decl_part begin stmt_part end} .

{rom / switch / node / lightnode / register / ram} ( ( iden // ',' )+ rangespec ';' )+ .

rangespec ::= [ '[' range [ ';' range ] ']' ]  .
range     ::= constval [ '..' constval ]  .

{downbus / tribus / upbus} ( iden rangespec ';' )+ .

act_funcpar ::= ( iden // ',' )+ .

...list ::= ( ...element // ',' )+ .

{act / cell}par ::=[{left / right / back / front}] iden rangespec .

iden ::= {'$' / letter} [ [ {'$' / letter / digit / '_'} +] {'$' / letter / digit} ] .
```

32 characters are analyzed

scope of variables:
single hierarchical level
within current func or cell

```
{terminal / light / delay} ( iden '.=' expr ';' )+ .

subscript ::= '[' [ [ {expr / range} ] ';' ] {expr / range} ']' .

bus iden [ subscript ] '.=' ( {oco / oco / expr enables} expr ';' )+ .
```

CONNECTIONS

---

1 iden, '('expr')', func_expr
2 '!'
3 '.'
4 '*' '/' mod
5 '+' '-'
6 '=' '#' '>' '<' '>' '<'
7 and nand
8 or nor
9 exor coin

priorities (1 is highest)

```
expr ::= { if expr then expr else expr endif
         | case expr of ( valuelist ':' expr )+ [else expr] endcase
         | unconditional_expr (see priority table of operators) } .

func_expr ::= iden['&'dec_no] '(' [expr // ',']* ')' .
```

EXPRESSIONS

```
make ( abut_exp '(' actparlist ')' ';' )+

abut_exp ::= ( ( slice // ':' )+ // '@' )+ .

slice ::= { [{mirx / miry / rotl / rotr / rotu}] '(' abut_exp ')'
          | iden [ '&' dec_no ] } .
```

':' > '@'

A @ B A : B

ABUTMENT

```
{at / on} expr do assgmt_list {endat / endon}

assgmt_list ::= ( iden '[' expr ']' ':=' expr )+ .
```

REGISTERS

```
if expr then stmt_list [ else stmt_list] endif
case expr of ( valuelist ':' stmt_list )+ [else stmt_list] endcase
wile expr do ( assgmt [otherwise {at / on} expr take expr ';' ] )+ endwile
```

```
kolte frame term ';' orin ';' {split matrix_pair / kolte_matrix} endkolte

pla frame term ';' matrix_pair endpla

{pal / pol} frame matrix {endpal / endpol}

inspec  }
outspec } ::= iden '[' range ']' .
term    }
orin    }

p_del ::= { ';' / ';'eoln / eoln } .

matrix ::= ([and_line] p_del )+ .

split_line ::= and_line ' ' or_line .

matrix_pair ::= ([split_line] p_del )+ .

kolte_matrix ::= ([kolte-Line] p_del )+ .

frame ::= cell_iden ':' inspec ';' outspec ';' .
```

line		
split		
and	or	kolte
0	-,x,X	0
1	-,x,X	1
-,x,X	-,x,X	-
0	1	&,D
1	1	#,S
	','	','

PLA etc.

parameter declaration

```
parameters ( iden // ',' )+ endparam .
paramvalues ( ( iden '=' constval ) // ',' )+ endvalues .
```

parameter expressions (pe)

```
...{:& / @& / !& / .& / ,&} pe ['['pe'..' pe ']'] ...

...{[& / ..& / && / &_} pe ...

pe ::= ( {iden / constval} // {&+ / &- / &* / &/} )+ .
```