Higher Level Simulation and Hardware Description Languages

Reiner W. Hartenstein
Kaiserslautern University
Postfach 3049, D-675 Kaiserslautern, F.R.G.
phone: (+49 - 631) 205 - 2606, or: (+49 - 7251) 3575

Abstract

This paper gives an introduction to Computer Hardware Description Languages (CHDLs) and their application in early phases of the VLSI design process. It first gives a survey on objectives of its use in simulation. Then it briefly introduces a subset of a modern register transfer language (RT language). Finally it gives a survey on various CHDL-based CAD tools and its linkage to physical design, as well as its integration into CAD environments.

Contents

1. INTRODUCTION: WHY CHDLs ?

The next higher abstraction level above gate level is called register transfer level (RT level), since its primitives are registers, register arrays, and data transfer paths, such as e. g. operators, buses, multiplexers, and others. Hardware descriptive notations at RT level (and sometimes above) are called *Computer Hardware Description Languages (CHDLs).* (For a classification see [HaHa87].) Some of them [HLW86] have a mnemonics which is similar to that of the Pascal programming language. CHDLs may be used to feed higher level simulators, to feed silicon compilers (at least future ones), for automatic generation of *go/no go* test patterns [HaWo85], for hardware specification, and for documentation, for more concise teaching the principles of digital hardware, use as a design calculus, and for many other applications. For a survey on CHDL applications see [Har87].

All modern CHDLs are hierarchical and thus have structural description capabilities, so that this should not be used as a classification criterium. We may distinguish two major classes of CHDLs: non-procedural languages and algorithmic languages. Non-procedural languages may be used only for description of input/output behaviour of systems and modules. Algorithmic CHDLs, have additional programming features to describe sequences of mircoinstructions and other

sequences. So the same language may be used e. g. to describe and simulate the data paths of a microcomputer and the microprograms running on it. The question only is, which solution has more advantages: using an algorithmic CHDL, or, using a non-procedural CHDL system, which is interfaced to a separate microprogram compiler.

Substantial Reduction of Complexity. What are the benefits in using such CHDLs, compared to traditional abstraction levels, such as e. g. the gate level ? The most important benefit is the reduction of notational complexity. Gate level notations, such as e. g. Boolean equations, do not yield a substantial reduction of complexity, compared to circuit diagrams. The average number of transistors per gate is the quotient of complexity reduction: this is only about 3 to 5, in CMOS and some other circuit techniques only about 4 to 8. In using CHDLs this quotient may by much higher, sometimes up to several hundreds.

One reason for reduced complexity is the fact, that CHDLs use to bundle a bit vectors of bits into words, like in high level programming languages. In describing a 32 bit data path, for instance, its data values are kept in a single word to be processed at once within the simulator and other tools. A second reason is the fact, that at RT level more powerful operators are available, such as e. g. multiplication, an equivalent up to hundreds of gates, and many others. A third reason for reduced complexity is found in the flexibility of RT language use. Modern CHDLs feature capabilities to describe a particular hardware in different levels of abstraction. That's why in using the same language a design process may start with a very high level specification of very low complexity, and after several steps of refinement it may end up with a more complex and more detailed description of a solution concept.

Training. So it is sure, that the complexity problem of VLSI design can be solved only by using CHDLs. Later in this paper a number of additional benefits will be illustrated, such as support of design for testability, early test pattern development, structured VLSI design, experimenting with alternative architectures before starting logic design, and many others. Why does a majority in industry hesitate to introduce CHDLs? Some quite interesting tools are available. However, most Universities do not teach using CHDLs, which would be an effort taking about less than half of the time needed to introduce Pascal. Another problem is the lack of methodology. At gate level a very elaborate and formal design methodology has grown, mainly within the last 30 years, due to contributions of thousands of scientists throughout the world. At register transfer level, however, most contributions are more of narrative character and of analytical nature, rather, than being a design methodology. A generally and widely accepted formal notation - comparable to Boolean algebra at gate level - has not yet been established in most application areas at RT level. So design tends to be more a trial and error procedure, or, to use one of a few popular concepts, such as e. g. systolic arrays and others.

2. EARLY PHASES OF THE DESIGN PROCESS

CHDLs are an important opportunity to avoid expensive redesigns needed to correct errors, such as e. g. bad testability, bad topology and bad structure of the circuit, too much area comsumption, or, bad (VLSI-) architecture, missing the requirements, and others. Many of such errors could be avoided, if design concepts would be decided at a very early phase of the design process, definitely before the costly logic design procedure has been started.

2.1 Specification and Design Problem Capture

One important role of CHDL use is design problem capture. To be sure to meet the requirements the design problem has to be pinned down the correct way. A concise notation has to be used to express the design problem. A description of a design problem using such a notation is called a specification. An advantage over narrative specifications ist, that an automatic syntax check may give valuable diagnostics on a specification and its implementability. Unreasonable specifications may be rejected by the diagnostic of a CHDL compiler - before the design process is started. So a lot of expensive redesigns could be avoided. Of course, a language has to be tailored to this specification diagnostics: it has to be some sort of guide line. So it has to be restrictive in a sense, that not every nonsense can be expressed by it.

Specification Verification by Simulation. To be sure to capture the design problem correctly the specification has to be checked against the requirements. If a CHDL implementation including a simulator is available, the requirements could be simulated, after the specification has been accepted by the tool. So the CHDL system may serve for design problem capture. Such a CHDL system could be also used as a communication medium between customer and design center, or, if within the same company: between product planning division and design division. The customer uses the CHDL system, such as e. g. a KARL compiler and simulator, for design problem capture. Reacting to simulation results the customer successively debugs the specifications. Finally the verified and debugged specifications (for instance, a KARL description of the design problem) are handed over to the design center.

2.2 Experimenting with alternative Architectures

Bugs in specifications are not the only possible reasons for missing the requirements. Sometimes the principles of a design concept are critical with respect to real-time performance, to design cost, or other important aspects. Often several possible solutions have to be considered and analyzed, so that experimenting with alternative architectures is needed. Of course such experiments should be carried out at the highest possible level of abstraction to avoid incomprehensibility of descriptions and high labour cost because of high complexity. For more detailed discussion see [GHW85]. For a survey on automated optimization support see[Wod86].

2.3 Design for Testability

Testing VLSI circuits currently is a major desaster area in industry, since the technology of testing and test pattern generation is far behind the possibilities of manufacturing technology and design capabilities. For mass production very often the time needed for testing is too long. Desirable would be around a second or less. For automatic test pattern development often an excessive amount of CPU time is needed. A very critical aspect is the fact, that for a given set of test patterns often the test coverage is much too low. This means, that the percentage of circuit faults, which will be detected by using a given set of test patterns, is far below 100%. This issue is critical, since it severely affects product quality. In some applications, such as where malfunction of circuits could be a danger to human life (process control, aerospace, some modern automotive, medical applications etc.), or, could make the entire mission fail (aerospace applications etc.) this quality aspect is one of the most important objectives at all.

Very early Test Pattern Developoment. In many cases the design is the reason, why a circuit's fault coverage is low. In such a case the best possible test patterns could not achieve high fault coverage. That's because of properties of the design important inner subcircuits cannot be reached by a sufficient percentage of stimuli. Nor a sufficiently high percentage of its responses could be observed from outside the circuit. Only an expensive redesign of the circuit, which takes testability aspects into account, could solve such a problem. The product development schedule could slip for months or more.

All this illustrates that design for testability is an important ingredient of the VLSI design process. Not only testability per se, but also the length of the test needed is a very important objective in design for testability. The best solution is to carry out test patern development in very early phases of the design process, at least before logic design has been started. The most desirable time would be, when the specifications are ready. Instead of being part of the logical design, and thus being highly expensive, testability would be a subject of early design planning and partitioning definition. The designer could fully concentrate on the testability architecture of a circuit and could experiment with alternative architectures. However, this would require, that the test patterns are available at such an early time, so that testability data and test length data of different version architectures are available. Otherwise the designer would not know, which alternative to decide, and, whether the design for testability efforts have to be continued or not.

Functional Testing. All this is feasible, since fortunately for production testing of integrated circuits only a *go / no go* test (sometimes called a *functional test*) is needed. That's because integrated circuits are not repaired, so that fault locating is not required. A functional test can also be developed without any structural knowledge about the circuit. (A test also exhibiting fault location diagnostics would be called a *structural test.*) That's why a functional test can already be developed from the functional description of a circuit, i. e. from its specification.

Integrating Simulation and Test Development. Although the area of functional test pattern generators currently is rather immature, it is useful to have it available along with the circuit specificaiton for designing for testability. For instance, the output of the KARATE test pattern generator (has been implemented in Kaiserslautern [HaWo85]) uses the same language SCIL [HHa86], which is also accepted by the KARL simulator. KARATE is the first algorithm with hierarchy capability, which uses symbolic techniques and accepts any high level, low level or mixed level multiple fault models [HeW88, AHW88, ARW88].

Problems yet to be solved, are the following ones. For large circuits an exhaustive simulation is not possible, unless an accelerator is available which runs the simulator, or, a physical model extension is used. So the user will have to select subsets of the test patterns in a clever way to run the simulator. (This would be supported by the good readability of the SCIL language, and the KARL simulator's dialogue mode capability.) Currently there is also no way to an optimization of functional test patterns with respect to those errors which have an extremely low probability for technological reasons. However, it is not known, how many of such errors could be expected. Also the automatic testability analysis area is rather immature. Some of the more widely known analyzers having been published yield quite obscure results. About a more recent one [SM85, SMP86] this paper's authors have only limited information.

2.4 Structured VLSI Design

The term of *structured VLSI design* has been coined by the Mead-and-Conway scene. It stands for a method to implement algorithms directly onto the planar surface of silicon in a way, which attempts that most cells of the design are connected by abutment. This means, that by means of port matching between neighbour cells no routing area between these cells is needed. This in many application problems is a highly efficient way to save chip area, since routing areas tend to eat up very much more chip area (sometimes up to about 95% of the chip) than active cells. The best way to use this method is it, to try to plan the chip in a way, that most of it is made up by arrays of abuttable cells. The success of such a solution highly depends on the cleverness in planning the shape and the topology of *key cells,* being efficiently abuttable. Often a successful key cell design is possible only, when a clever partitioning and placement strategy has been used in chip floor planning (also see [LeNe87]). All this means, that layout considerations are needed at very early phases of the design process, about when the specification is formulated. An important and repidly growing branch of structured VLSI design is the area of systolic architecture synthesis. Currently at Kaiserslautern a CHDL-based systolic synthesis system is being implemented [Lem88, Lem89].

Innovative Power of VLSI Design. Structured VLSI Design as a design style has an innovative power. The success of structured VLSI design efforts depends on selection of the best possible task realization algorithm for a VLSI solution. Sometimes the smart memory approach is a good solution (this is shown in tutorial-like explaining the design of a simple sorter chip example in [BBad85]). Also this illustrates the benefit of very early chip planning. To provide means for design plan verification at this early phase, a simulator input language (a CHDL) is needed, which can express such partitioning and topological features already at specification level. Such language features will be shown later.

3. INTRODUCING A CHDL AND ITS USE

To get a more illustrative presentation a particular CHDL will introduced. It is the KARL-III non-procedural language which is the most familiar one to the author [Har77, NN85, HLW86, HHa86], having been successfully used for quite a number of designs, also complex ones having been fabricated [(e. g. BaV86, CFLP88, Far86, MoM87] and others). KARL-III is a multi-level language which includes the **RT level, gate level,** and the **switch level.** This has good reasons: sometimes small pieces of a description cannot be expressed at RT level, so that this 'remainder' can only be presented in using gate level primitives. Another important reason is the bus, being an important architectural resource at high levels of system description: it is a switch level concept [Ha77]. This multi-level paradigm has more advantages: using the same language top-down design a specification can be successively refined to a more detailed concept.

Structural primitives

group	primitives
modules (user-defined)	**cell** declaration topology: front, back, left, right, ports: in, out, bi **function** declaration
abutting chip floor planning	**make** expr.: @, :, mirx, miry, rotr, rotl, rotu
inter-connect	simple: .= := bus, terminal formatting: •, | (catenate) [..] (subscript) *also see wiring functions*
user-defined cells	

a)

KARL functional primitives

level	combinational	w. memory
RT level	**arithmetic:** +, -, *, /, mod **relational:** >, <, =, =<, >=, <> **multiplexers:** if..... case.....	register, array reg. RAM ROM constant
gate level	**logical:** not, and, nand, or, nor, exor, coin	delays
switch level	**bus drivers:** oeo, oco, enables **buses:** upbus, downbus tribus	dynamic memory
clocking: at, on, wile		

b)

Fig. 3.1 . Survey on KARL-3 Language Primitives

KARL-III is a multi-paradigm language. It is not only multi-level, but also strongly typed (for diagnosability), it features structural description as well as functional description, it also features topological description including a cell abutment expression sublanguage (for floor plan capability). We believe, that all this is a nearly optimum mix to fulfil the requirements of design problem capture, early design for testability, and, structured VLSI design. Nevertheless, KARL is not a baroque language, like Ada, for instance.

Comprehensibility. Although being multi-level, KARL is substantially ***more easy*** to learn than Pascal. For instruction and documentation we believe, that the simultaneous use of two versions of a RT language is quite useful for better comprehensibility. This fact has been recognized quite earlier, so that in fact at gate

3.1 Language Primitives

The language, its power, and its flexibility is determined by the repertory of its primitives. We may distinguish structural primitives from functional ones. Structural primitives within KARL-III are uniform throughout all abstraction levels: they are all the same, no matter whether being used at RT level, gate level, or, at switch level, and even at circuit, and symbolic layout levels [Wel86, Wel87]. KARL-III structural primitives are much more easy to use than those of KARL-II (which is no more supported).

Utilities (standard functions):

operative standard functions		wiring standard functions	
	comment	word format	array format
code conversion:		**shift functions:**	
decode		shr shl	push pop
encode		dshr dshl	dpush dpop
decount		cshr cshl	cpush cpop
encount		nshr nshl	npush npop
test functions:		eshr eshl	epush epop
equ, odd, even		clrshr clrshl	clrpush clrpop
testunary,		**shuffle functions:**	
testsingulary		fold	merge
miscellaneous:		**butterfly functions:**	
inc	*increment*	fly	butter
dec	*decrement*	**mirror functions:**	
pril	*priority left*	reflect	reverse
prir	*priority right*	**field select functions**	
		msb, lsb	msw, lsw

Fig. 3.2 Survey on KARL-III Language Utilities

Structural primitives (fig. 3.1 a) may be subdivided into module definition features, and, into notations to specify interconnect. KARL III provides two different module declaration facilities, the func declaration for user-defined function modules, and the cell declaration also including topological features, which are explained in section 3.2 (for a survey see fig. 3.1 a). There are two kinds of interconnect descriptions: implicite descriptions within expressions and explicite descriptions in terminal, bus, assignment statements, as well as by means of actual parameters (connections to cell ports) in cell instantiations (for details see [HLE86, NN85]).

Functional primitives may subdivided into RT level, gate level, and switch level primitives (see fig. 3.1). RT level primitives of KARL-III are: arithmetic and relational operators, multiplexers, and all lements with permanent memory (register, RAM, ROM, etc.). This supply is extended by RT level utilities (standard functions), such as e. g. decode, encode, priority functions etc. The group of wiring operators may also considered to be RT level operators, are subject of sect.3.3. Gate level primitives of KARL-III include the usual logical operators, and two kinds of delay elements which may be used to model propargation delays at all levels. At switch level a simple but flexible technology-independant bus modelling scheme has been developed [Ha77, NN85, HaEIS86], providing an upbus, downbus, and, tribus (for three-state bus) declaration for the 3 basic bus types. Three technology-independant bus driver primitives oco ('open collector output'), oeo ('open emitter output'), and enables (having a separate control input) provide a method for modelling bus systems, and to model the circuit principles of of MOL (matrix-oriented logic) using personality matrix specifications (see section 4.0).

3.2 Cell Modules with Floor Plan Capability

The concepts underlying the KARL cell definition and instantiation features efficiently supports structured VLSI design and the integration of KARL-based CAD tools into physical design. Relative to its declaration orientation a KARL cell distinguishes four different sides (left, right, front, and, back) of port location (fig.

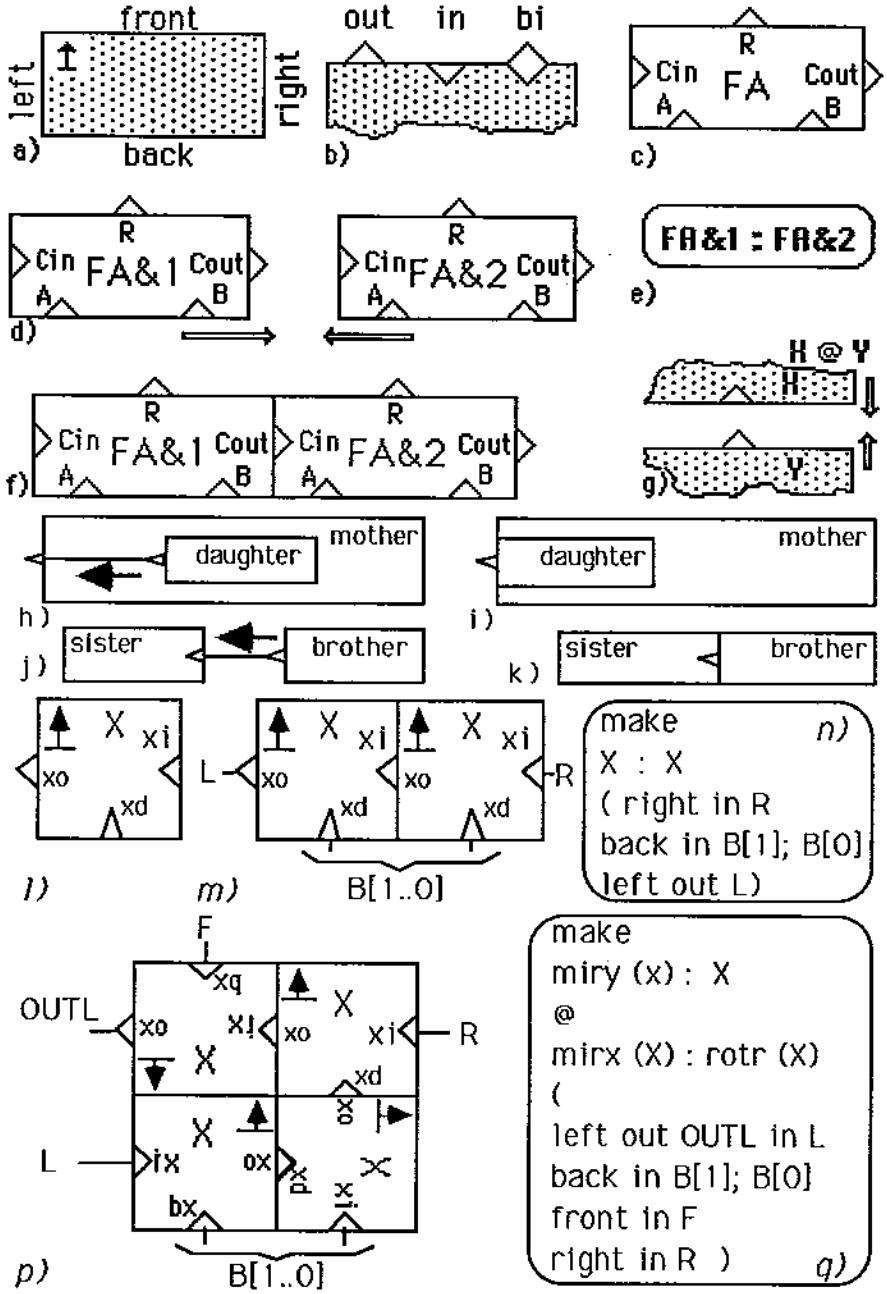


Fig. 3.3.

3.3 a), as well as 3 different port types in, out, and bi (fig. 3.3 b). These declaration attributes are used for automatic interconnect generation for cell abutments. Fig. 3.3 c shows the ABL diagram of the external view of a full adder cell example named FA. Fig. 3.3 d shows two instances FA&1 and FA&2 of this cell, due to be abutted ('&' is the separator between cell name and instant number). Fig. 3.3 e shows the abutment expression which is the notation for this instruction, where ':' stands for horizontal abutment, i. e. for abutment of slices. Fig. 3.3 f shows the result of the abutment operation. Fig. 3.3 g illustrates vertical abutment of two cells X and Y, described by the abutment expression X @ Y, using the '@' connective.

Abutment Expressions. Figures 3.3 h through k illustrate the alternatives between routing connection (h and j) and abutment (with automatic interconnect , generated by the KARL compiler, see figs.i and k). Figures j/k illustrate interconnect between sister and brother cell (same level within the hierarchy). Firuges h/i illustrate daughter-to-mother interconnect (the daughter cell is an internal component within the mother cell). Also complex abutment expressions may be formulated for the synthesis of complex supercells including several levels of cell hierarchy, as well as rotate and mirror transforms on single cells and compound cells. Figures 3.3, l through q show two examples of compound cells described by non-trivial abutment expressions. The cell X in fig. l is used as a component. Fig. m shows the ABL diagram of a two cell abutment example, and fig. n shows its abutment expression. The parameter list within parentheses describes the interconnect of the compound cell with nodes (L, R, B) of its environment. Fig. p (ABL diagram) and q (its equivalent abutment expression) describe the instantiation of a four cell compound also using rotate and mirror transforms (transforms are relative to the declaration orientation of the cell X, shown in fig. l). For more details about this abutment algebra and its chip floor planning application also see [HLW86, NN86].

3.3 Description of Wiring Patterns

Any arbitrary wiring can be expressed in KARL by user-defined descriptions, and, by user-defined routing cells. However, let us look at those wiring patterns which are predefined by KARL language primitives. In KARL-III there are three classes of wiring descriptions: 1) those changing path width ('•' and '|' for juxtaposition of paths to create a wider path, as well as (2) uses of subscripting to split up a path; compare section 3.1 and fig. 3.1), and those (3) which preserve data path width. The latter ones may be split up into subgroups: (3a) direct connections which do not affect the sequence of bits, user-defined routing boxes, and, (3b) *wiring operators* which rearrange the sequence of bits algorithmically, such as shift, shuffle, reflect, and butterfly operators.

Wiring Standard Functions. The implementation of wiring operators in KARL uses the KARL standard function format. Fig. 3.4 illustrates a few examples, 16 bits wide: a) cirshr&3 a circular shift right by 3 bits, b) fold&2, the *'perfect shuffle'*,

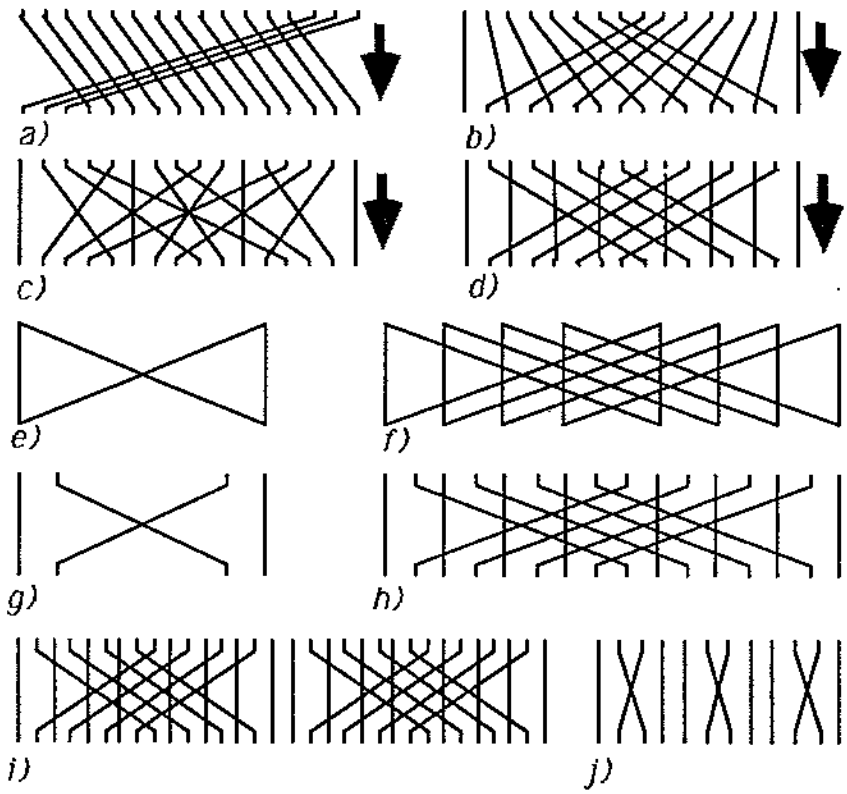
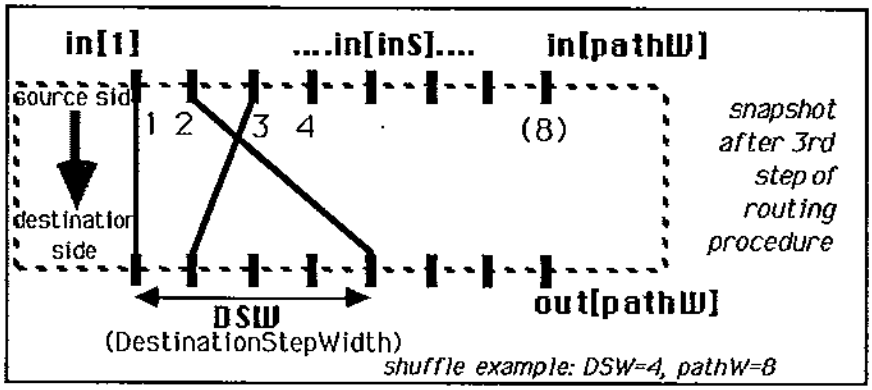


Fig. 3.4.

c) fold&4, a shuffle with destination step width of 4, d) a butterfly example. The constant number behind the '&' separator is the function parameter. For shift operators the wiring pattern is well known. The algorithm generating the shuffle wiring pattern is rather easy to be found for a particular path width (8) and parameter (4) (see fig. 3.5/ By the

KARL compiler the wiring patterns are automatically adapted to both parameters. Illegal parameter combinations (possible with shuffle and butterfly patterns) are detected and diagnosed by the compiler. Fig. 3.5 illustrates the meaning of the destination step width parameter in shuffle patterns.

Butterfly Wiring Patterns. Butterfly patterns have many applications, such as for example in digital signal processing, micro processor and micro computer interconnect networks and many other areas [Bat76, GoL73, MGN79, TYF74, TYF81, etc.]. Fig. 3.4. e illustrates the butterfly shape. The precise wiring pattern of the elementary butterfly, however, is that shown in fig. 3.4. g. For wider data paths the butterfly pattern usually is shaped by the superposision of several elemantary butterflies, such as illustrated by the 4-layer example in fig. 3.4 f. This figure is only an illustration of the pattern generation principle. The precise wiring pattern of it, however, is shown by fig. 3.4. h. The butterfly function parameter indicates the number of segments. Figures 3.4 d through h only show examples with the parameter value i = 1 (butterfly default parameter value within KARL). Figures 3.4 i, or, j, show examples with i = 2, or, with i = 3, respectively.



SHUFFLE procedure parameters:
Destination Step Width DSW (explicit)
Datapath Width pathW (automatically adapted)

legal parameter ratio: iff
Rem (pathW/DSW)=0
and DSW ≤ pathW / 2

Variables used in fig. 3.5

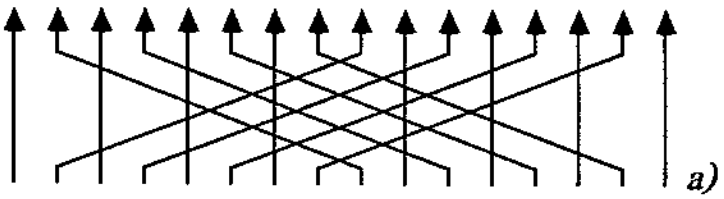
name	type	meaning
pathW	integer	DATApathWidth
DSW		DestinationStepWidth
in		input
out		output
inS		inputSubscript
outS		outputSubscript
LFS		LeftmostFreeSubscript
isLFS	Boole	currentSUBSCRIPTisLFS

Fig. 3.5.

Within the KARL System the wiring operators illustrated above are provided in two versions: a *word format* version rearranging the bit sequence, and, an *array format* version reordering the word sequence in an array. For a survey see fig. 3.2. The array format versions of these wiring operators are useful for concise description of interconnect patterns in switch boxes like bunyan networks etc. (e. g. see [LaMa86, BaV86]), for parallel signal processing circuits, such as e. g. for fast fourier transform. Shuffle operators are also useful for mixed mode (word format) RT level and (single-bit format) logic simulation in using KARL, where the shuffle pattern may be used as an adapting interface between both kinds of the hardware description. Fig. 3.6 shows an example of a recursive definition of a butterfly wiring pattern, using KARL as a calculus. For more details you may request [HLW86, NN86].

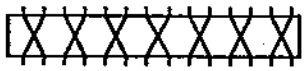
3.4 Step-wise refinement using a CHDL

Refinement capability is an important language property to achieve its use as a design language. So you do not need to change the language in top-down planning from a purely functional specification to a more detailed description which then may be used to enter logic design and physical design. So you may use the same language and the same tools to analyze and to synthesize a conceptual description being a structural / topological / functional notation which carries along all the clever architectural ideas for testability, for structured design, etc. over to the silicon implementation team.

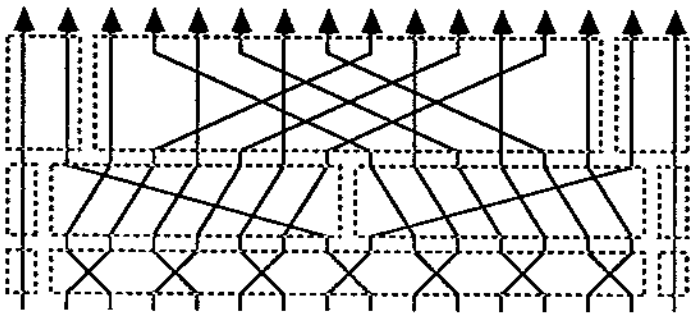


reflect parameter:
segment count

width = 16 ->
(width-2)/2 = 7



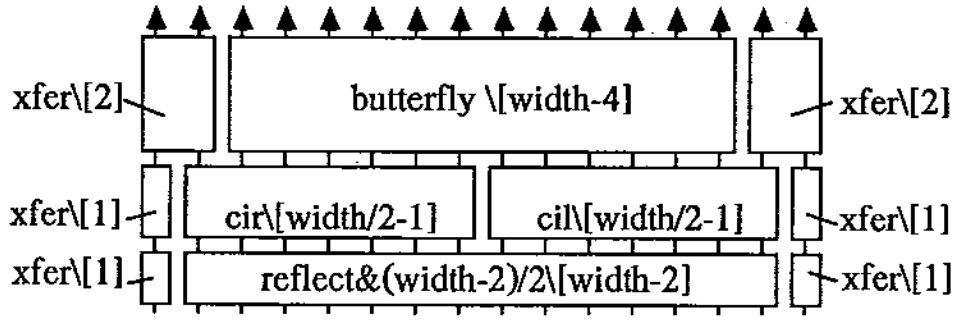
width function:
returns total path
width inherited
from source



e)

```
ButterflyOut .=
xfer\[2] : butterfly\[width-4] : xfer\[2]
( xfer\[1] : cir\[width/2-1] : cil\[width/2-1] : xfer\[1]
(xfer\[1] : reflect&(width-2)/2\[width-2] : xfer\[1]
( <operand>
)))
```

c)



d)

Fig. 3.6

Fig. 3.7 illustrates stepwise refinement in KARL and ABL by an example circuit, a simple 2-way multiplexer. It always shows two notations: the textual (KARL) notation at the left side, and the ABL notation (such as produced by the ABLED editor [GHW85, Far86]) at the right side. It shows the circuit in different degrees of detailedness, in different abstraction levels: a) at RT level, b) at gate level, c) at switch level. So this also illustrates, that the KARL descriptions a) thru c) are technology-independant. By the way: the simulator gives the same response for all three of them. So this is required to have a consistent implementation of the language KARL.

Figures 3.7 d) and e) are no more KARL descriptions, but circuit diagrams of different implementations: d) in CMOS technology, e) in NMOS technology. Figures d) and e) have been produced bei another graphic editor MLED [Wel86]. This is a mixed-level editor including all levels from RT down to layout. Bei menue guidance any mixed-mode representation can be arranged, such as e. g. showing one cell at ciruit level, another one at layout level, a third one at RT level (here using ABL) etc.

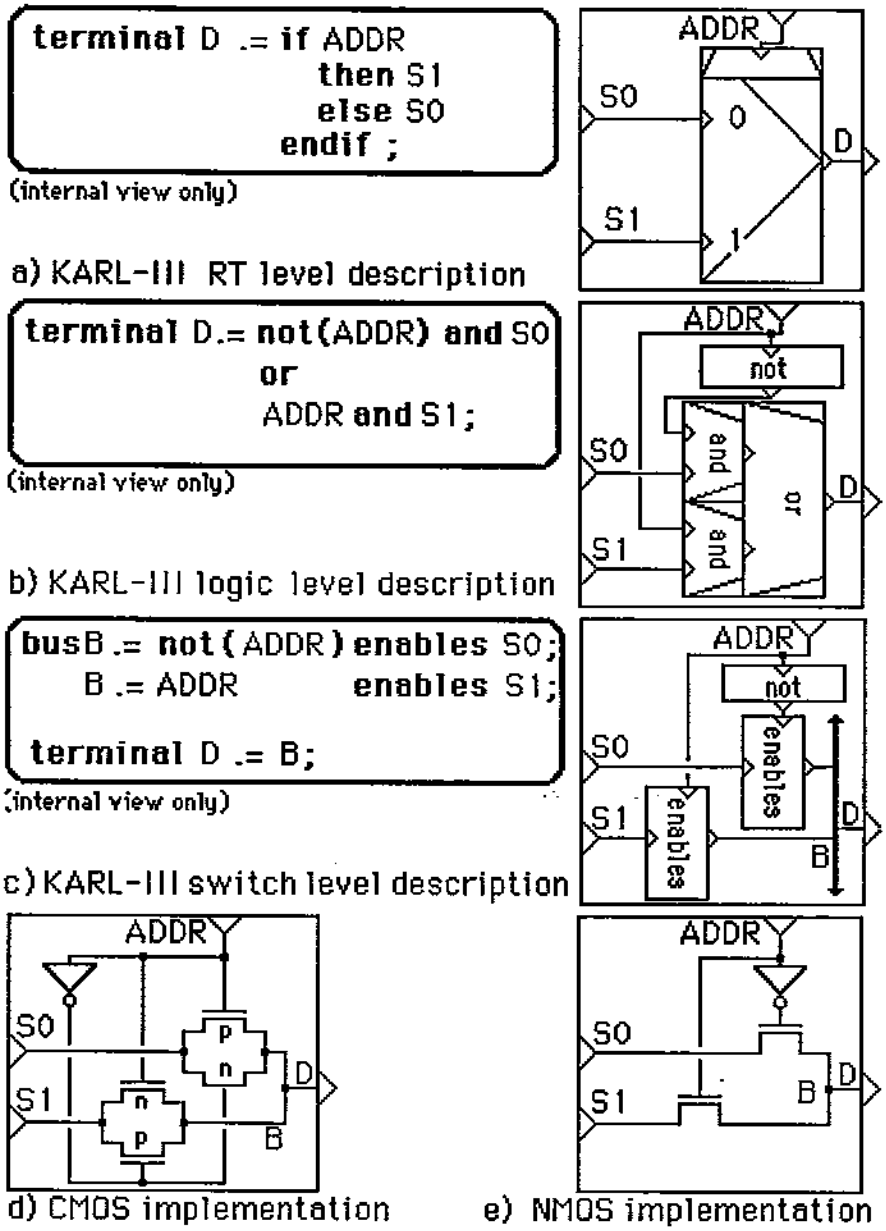


Fig. 3.7

3.5 CHDL Use as a Design Calculus

Section 3.4 has illustrated the consistency of KARL and its capability for step-wise refinement, so that top-down design can be carried out without changing the language. This is a requirement if it is desired to use the language as a design calculus in order to reach design goals by means of a sequence of algebraic manipulations. However, such a language can only be a medium to express such algebraic rules, however, it cannot be this algebra itself. This medium, however, is powerful, suitable for effective exploration of many areas of application by experimenting with alternative architectures and structures. Fig. 3.6 shows an example of a recursive definition of a butterfly wiring pattern, using KARL as a calculus. In [NN85, Lem86] an example is described, where a regularly structured integer multiplier layout has been developed from the algorithm description by a sequence of successive algabraic manipulations. In [Ha77, Lem 86] approaches to a general algebraic schematics development are illustrated in using binary-to-BCD, BCD-to-binary code converter, and universal shifter examples.

4. CHDL-BASED DESIGN ENVIRONMENTS

The KARL core system (fig. 4.1 a) as well as the KARL environment are modular systems. The KARL core system uses three different languages: the hardware description language *KARL* as a description source, the simulator activation and test description language *SCIL* [HHa86], and the executable intermediate form called

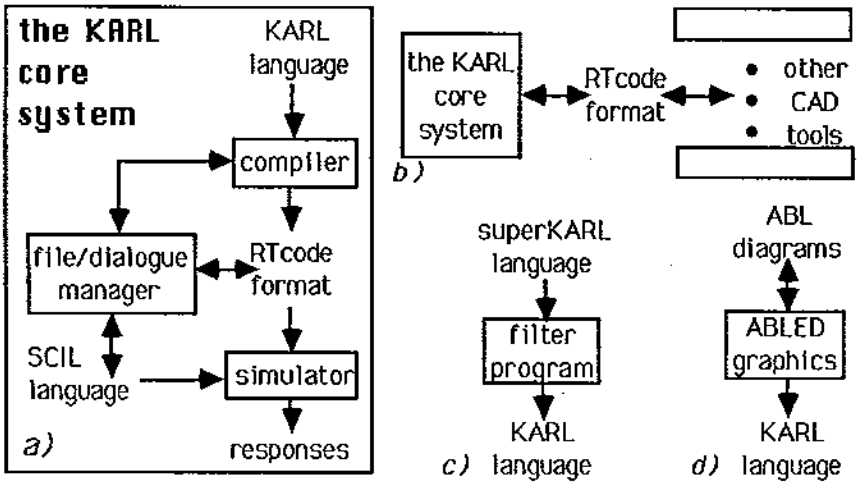


Fig. 4.1.

RTcode. RTcode [HaMa86] has turned quasi-standard interface to a number of tools within and outside the CVT project [NN86] (fig. 4.1 b). For instance, three different simulators have been implemented, which accept RTcode: the original KARL simulator [Web81, HHa86], the fault simulator of the CVT CAT system [SMP86, SM85], and an event-driven fast simulator having been implemented at CSELT [Per86].

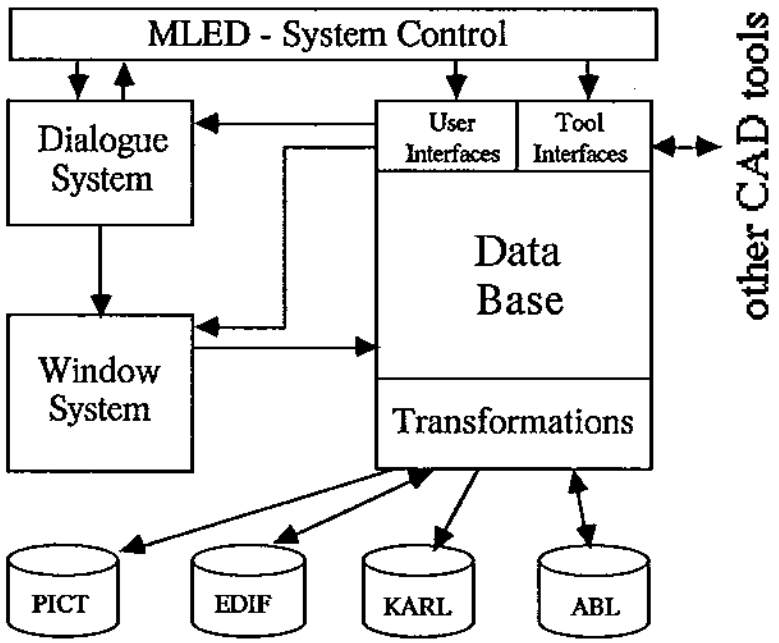


Fig. 4.2

Figures 4.1 c) and d) illustrate the implementation of other languages by means of precompilers or interpreters generating KARL source descriptions. The language *superKARL* [GHHO86] is a KARL extension featuring parametric cell descriptions using array size and path width parameters [HaHa86]. Fig. 4.6 illustrates the generation of cell arrays under control of superKARL array size parameters and data path width parameters: on-dimensional cell arrays (a), bit node

arrays (e), and word node arrays (f); two-dimensional cell arrays with linear growth(c), exponential growth (d), node arrays with exponential growth (g), as well as recursively definde two- dimenstional cell arrays (b). superKARL also features a rule-driven, and thus technology-adaptable algorithm for translation of personality matrixes into KARL functional descriptions for a wide variety of matrix-oriented logic (MOL) circuit techniques, such as e. g. PLAs, folded PLAs, Weinberger arrays, folded Weinberger arrays, Lopez/Law dense gate matrix layout, KOLTE arrays, and others [GHHO85]. There are also other CAD tools having interfaces to the KARL system [NN86], not described here because of lack of space.

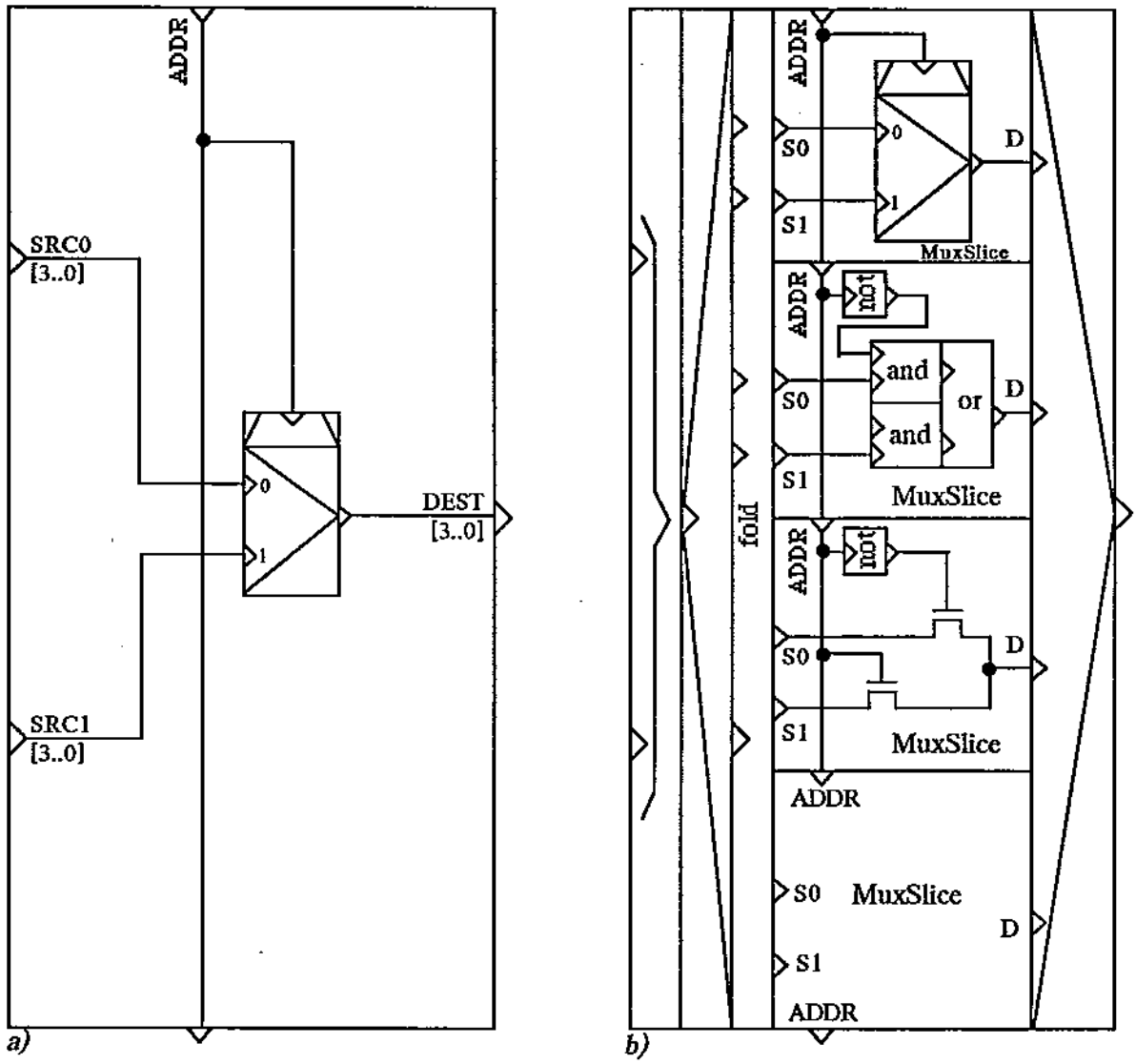


Fig. 4.3.

4.1 Using an Interactive Graphic CHDL Editor

This section briefly illustrates a new interactive graphic RT level editor ABLED, from a user's point of view. ABLED has been developed within the CVT project bei CSELT (Torino, Italy) and Kaiserslautern University [GHW85]. It is an interactive graphic interface to the KARL system (see fig. 4.1 d). Its typical diagram symbols are illustrated by the right side of figures 3.7 a thru c. The arrows to indicate ports are placed on the edge of cell boundary boxes to allow the DOMINO notation [Ha77] to show symbolic abutment (of abstract boxes) in architectural diagrams, and, to show physical abutment (when boxes reflect the shape of real cells) in case of a partitioning derived from a chip floor plan. This DOMINO feature is especially useful in MLED [Wel86] (for configuration see fig. 4.2), which combines the features of ABLED and those of editors for layout, circuit diagrams, and logic diagrams (also see section 3.4). The symbols are automatically created by picking via menue. The editor also includes an on-line graphical syntax check,

which immediately diagnoses illegal matings. This accelerates working with KARL considerably, since most of the diagnostics is already interactively available, before the KARL compiler has been called to parse the derivative of the ABL data structure (fig. 4.1 d). ABLED (and MLED) uses a tightly guiding menue technique, so that working with KARL is much more easy to the user via ABLED, than directly at the KARL textual interface. For illustration fig. 4.3 shows two alternative representations of more complex diagram example having been plotted by MLED. A detailed description of ABLED, from a users point of view, gives [GHW85]. For MLED see [Wel86, Wel87].

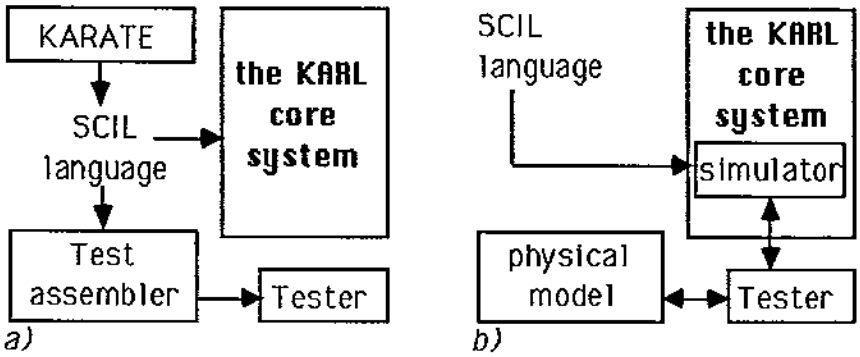


Fig. 4.4

4.2 CHDL-based CAD Tools

A number of KARL-related CAD tools [NN86] have been developed within research projects: within the CVT project having been funded by the Commission of the European Communities, by the multi-university E.I.S. project being funded by the German Federal Minister of Research and Technology, and also outside these projects. Those programs are tools for automatic synthesis (silicon compilers using KARL source input [ArMa86, EvP85]), for interactive synthesis (microprogram transformation [RaGr86], interfacing to RTcode of the KARL core system, compare fig. 4.1 b) for RT level verification [GSch85, SchG86], for test pattern development and testability analysis (the CVT CAT Environment [SM85, SMP86], the KARATE program [HaWo85]). Fig. 4.4 a.) shows how test development and simulation are integrated by using the same language SCIL (Simulator Command and I/O Language [HHs86]), which at the same time is the test description language used as an output language by the test generator KARATE, and, the stimuli and command language used as an input to the KARL simulator inside the KARL core system (compare fig. 4.4 b). A test assembler program is used to assemble a device-specific test program for the test device in use (fig. 4.4 a). Only this assembler has to be changed, when an other test device will be used. Fig. 4.8 shows a SCIL program example: for testing the circuit shown in fig. 4.7 (left side: ABL diagram, right side: its textual version).

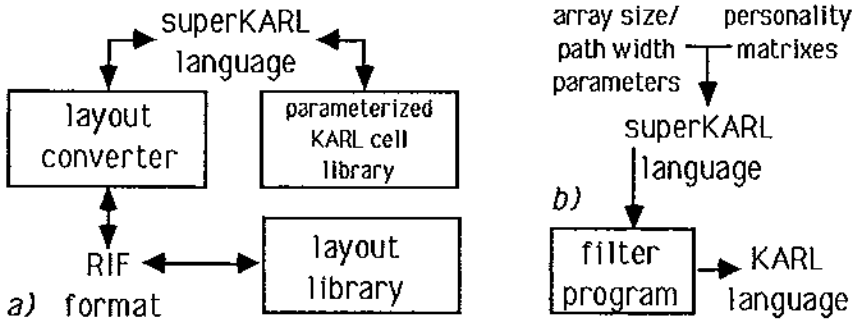


Fig. 4.5

Fig. 4.4 b shows, how also a physical model extension feature (PMX [HaHi86]) is integrated into the same set-up. (A PMX allows the simulator to communicate with real hardware *'physical model'*, such as e. g. an already existing prototype hardware module of the system to be developed. This requires by far less CPU time, than simulating the whole system.) Since the tester anyway is directly coupled to the computer which is hosting the KARL core system, as well as the test program assembler, the physical model may be just plugged into the adapter of the testing device. The only difference is, that the tester is talking to another piece of software: to the simulator.

4.3 Interfacing to other CAD Tools

This section briefly illustrates interfacing to layout level by means of a few example configurations. The most direct connection from CHDL down to layout is the silicon compiler, having been mentioned above [LaMa86, EvP85].

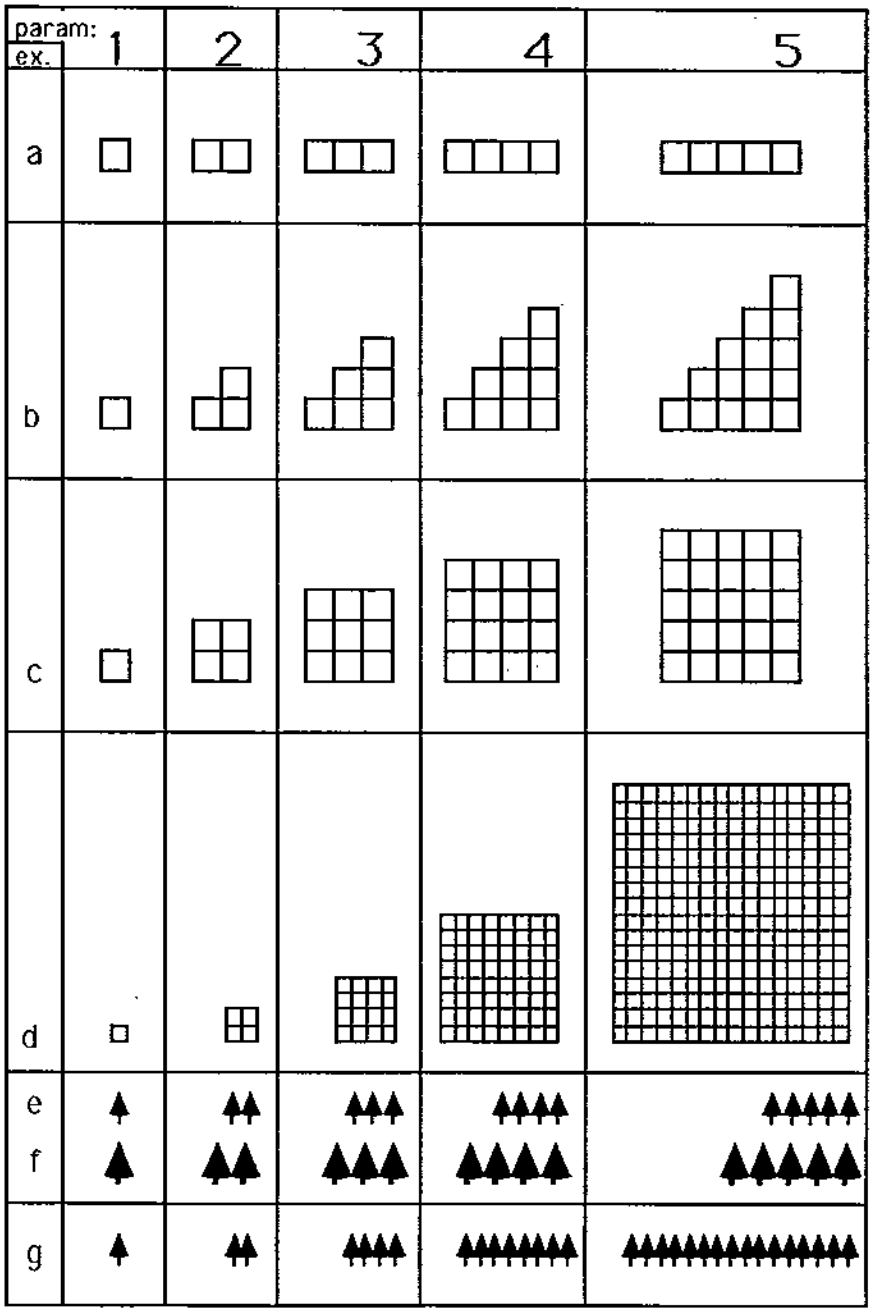
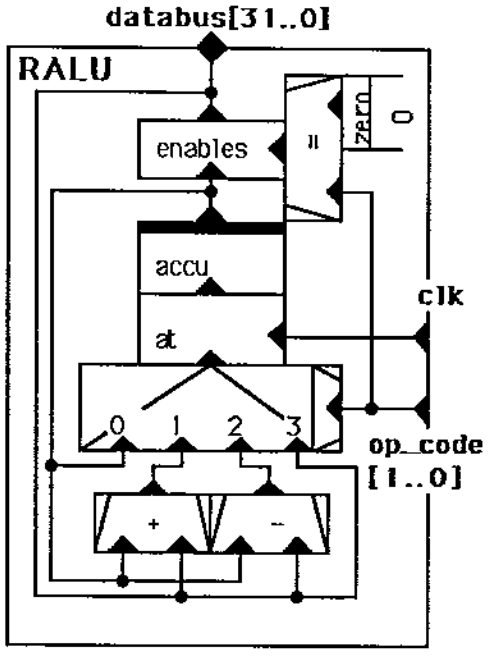


Fig. 4.6

KARL Extraction from Layout. The other way around is behavioural extraction from layout. The REX system [Neb86] directly generates KARL descriptions from Layout (so the interface is simple). It is rule-driven, and thus technology-adaptable. REX is useful, but wasting CPU time for large circuit design verification, however, would be more efficient for verification of critical cells and library cells. So such an extractor could be very useful to verify the basic circuit library needed for a much more efficient methodologies based on personality matrix notations of MOL (matrix-oriented logic; compare section 4.)




```
cell RALU right in op_code[1..0]
back: bi databus[31..0] right in clk);

constant zero[1..0].= 0;
register accu[31..0];
tribus   databus[31..0];

begin
  bus databus.=
      (op_code = zero) enables accu;
      {value of register accu is written
       on the bus if op_code = 0}

  at clk do accu:= cases op_code of
                   0: accu
                   1: accu + databus
                   2: accu - databus
                   3: databus
                 endcase;
  endat;
end.
```

Fig. 4.7

KARL Extraction from Personality Matrixes. The MOL extractor [GHHO85] accepts description sources at higher symbolic layout levels, using personality matrix formats, or similar notations. At this abstraction level many CAD tools are much more efficient than those based on lgeometric ayout sources. Examples: extractors (generate KARL description from a personality matrix) synthesizers and topological optimizers [HaEIS86] layout architectural converters (see next paragraph) etc.

Support of Layout Conversion. Let us go a little bit into detail in layout conversion. Its goal is the architectural exptension or compaction of circuit layout with respect to array size and path width. The goal, for example, be the automatic conversion of the layout of a 64 kbit memory into that of a 1Mbit memory. The memory array size has to be extended, and its peripheral logic has to be extended linearly and to be relocated geometrically, where a modified layout format RIF (relative intermediate form) may be helpful (fig. 4.5 a). (Also fan-out changes etc. have to be considered, of course). However, the organizational aspects are of much higher level, than layout. Such a layout converter has to be guided by RT level data. A KARL extension like *superKARL* (also see section 4.) or its forerunner version *hyperKARL* [Borr85] uses such array size and path width parameters. The actual cell hierarchy tree is produced after parameter assignment and translation into KARL (fig. 4.5.b). The layout converter (fig. 4.5 a) has to be partly implemented in a similar way, or, has to be part of, or, interfaced to, the filter program. This again shows the usefulness of interfacing RT level tools to low level CAD tools.

KARL interfaces to other CAD Tools. The following tools, having been interfaced to KARL, or, beeing KARL-based, have not yet been mentioned in this paper. The VERENA verifier, based on theorem proving techniques, compares two different KARL descriptions for equivalence [GSch85, SchG86]. Also the ARIANNA chip planner has interfaces to KARL [ARA84], as well as the following silicon compilers, or, silicon assemblers, repectively [ArMa86, BILM88, BoLa88, EvP85]. The CVT CAT (Computer-Aided Test Development) environment uses 5 different tools, part of it beeing KARL-based [MHM84, SMP86], also see [Mav87, Mel87]. For more literature and other information on KARL and its applications see [NN86, LM86].

5. CONCLUSIONS

We have illustrated the usefulness of using CHDLs and related tools. We shave shown, that contemporary CHDLs are a better front end of the ASIC design process, that CHDLs are very useful in concise design problem capture, in design for testability and, in efficient structured VLSI design in directly casting algorithms onto silicon. We tried to illustrate the substantially improved acceptance of CHDLs by provided an interactive graphic user interface. We also tried to show the

```
declare

variable LOOP_IND [ 31..0 ];

constant READ_DATA := 0;
 ADD_OP := 1;
 SUB_OP := 2;
 LOAD_DATA := 3;

proc CYCLE ();
(* perform a cycle in
   our simulated hardware *)
   clk := 0;
   step;
   clk := 1;
   step;
   clk := 0;
endproc

proc LOAD_IT
( in VALUE [ 31..0 ] );
(* procedure to load a value into
   the register of our example *)
   databus := VALUE;
   op_code := LOAD_DATA;
   CYCLE;
   op_code := READ_DATA;
   CYCLE;
endproc

proc ADD_IT ( in OP_2 [ 31..0 ] );
(* perform the add operation *)
   databus := OP_2;
   op_code := ADD_OP;
   CYCLE ();
   op_code := READ_DATA;
   CYCLE ();
endproc

proc SUB_IT ( OP_2 [ 31..0 ] );
(* perform subtract operation *)
   databus := OP_2;
   op_code := SUB_OP;
   CYCLE ();
   op_code := READ_DATA;
   CYCLE ();
endproc
enddeclare

(* begin the main program *)
   (* initialize the simulated circuit *)
   LOAD_IT ( 0 );
   (* test the subtract operation *)
   databus ?= LOOP_IND;
   LOAD_IT ( 65 );
   for LOOP_IND := 64 downto 0 do
      SUB_IT ( 1 );
      print ( hex data, LOOP_IND );
   endfor

   (* do another test *)
   LOAD_IT ( 4711 );
   for LOOP_IND := 1 to 10 do
      ADD_IT ( LOOP_IND );
      SUB_IT ( LOOP_IND );
      if databus /= 4711 then
         write ( ! test error ! )
      endif
   endfor
endsim
```

Fig. 4.8

substantial acceleration of the design process by using such a graphic interface including an on-line graphic syntax check. We hope to creasonalble onvince the readership of the VLSI community that in an ASIC-oriented and USIC-oriented (User-specific IC) design environment there is no way to cope with complexity of design planning and product planning without using such high level tools.

6. LITERATURE

[ABA86] G. Girardi: Collection of Viewgraphs of the 1st ABAKUS (ABL and KARL user group) workshop, Passau, F.R.G., June 1986

[ABA88] W. Grass: Proceedings of the 2nd ABAKUS (ABL and KARL user group) workshop, Innsbruck, Austria, September 1986

[AGA84] G. Arato, O. Gaiotto, P. Antognetti, A. de Gloria; ARIANNA: Floor Planning Tool; CVT/CSELT technical report, Genova /Torino, 1984

[AHrW88] G. Alfs, R. Hartenstein, M. Riedmüller, A. Wodtko: Integration of Simulation, Test Development and Test into a High Level Design Environment; Proceedings IFIP TC-10 Conference on VLSI Computer Architecture, Pisa, Italy, Aug. 1988; Elsevier / North Holland, Amsterdam / New York, 1989

[AHW88] G. Alfs, R. Hartenstein, A. Wodtko: The KARATE System - Integrating functional test development into a CAD environment for VLSI; Proceedings ICCD - Int'l Conf. on Circuit and Computer Design, Port Chester, NY, USA, 1988

[ARW88] G. Alfs, R. Hartenstein, A. Wodtko: The KARL/KARATE System - Automatic Test Pattern Generation Based on RT Level Descriptions; Proceedings ITC - Int'l Test Conf., Washington, D.C., USA, 1988

[ArMa86] G. Arato, R. Manione: PSICO: A System for Automatic Layout Generation; report, CSELT, Torino, 1986

[Bat76] K. E. Batcher: The Flip Network in STARAN; Proc. 1976 Conf. on Parallel Processing; IEEE 1976

[BaV86] G. Balboni, V. Vercellone: Experiences in Using KARL-III in Designing a CMOS Circuit for Packet-switches Networks; in [ABA86]

[BBad85] K. Bastian, R. Hartenstein, W. Nebel: VLSI-Algorithmen: innovative Schaltungstechnik statt Software; VDI/VDE-GME-Tagung "Mikroelektronik in der Automatisierungstechnik", Baden-Baden, 1985, VDI-BerichtNr. 550, VDI-Verlag, Düsseldorf, 1985

[BBG86] A. M. Biraghi, A. Bonomo, G. Girardi: ABLEDitor: User Manual; CSELT, Torino, Italy, July 1986

[BGLM88] A. Bonomo, G. Girardi, A. Lecce, L. Magiulli: GENMON: a specialized ABL editor for design methodology descriptions; in [ABA88]

[BILM88] A. Bonomo et al.: Easily testable data part synthesis in the BACH silicon compiler; in [ABA88]

[BoLa88] A. Bonomo, L. Lavagno: Control part synthesis in the BACH silicon compiler; in [ABA88]

[CFLP88] R. Cecinati et al.: The use of KARL-III in the RIPAC Chip Design; in [ABA88]

[EvP85] E. von Puttkamer: Das SIC-System als Kern eines Silicon-Compilers, report SFB-124, No. 18/85, Fachbereich Informatik, Univ. Kaisaerslautern, 1985

[Far86] D. Farelly: The Practical Application of ABLED and KARL in Designing a Speech Synthesizer; in [ABA86]

[GGG87] G. Girardi, S. Giorcelli, G. Diandonato: The HDL subsystem of integrated CAD systems; in Har87]

[GHHO85] J. Gebhard, R. Hartenstein, R. Hauck, D. Oelke: Behavioural Extraction from Personality Matrixes of Matrix-Oriented Logic (MOL) Circuits; (submitted for publication)

[GHHO86] J. Gebhard, R. Hartenstein, R. Hauck, D. Oelke: The superKARL-III Language Specification; report, Fachbereich Informatik, Kaiserslautern University, Kaiserslautern, F.R.G., 1986

[GHW85] G. Girardi, R. Hartenstein, U. Welters: ABLED: a RT level Schematics Editor and Simulator Interface; Proc. EUROMICRO Symposium Brussels, Belgium, 1985,(ed.: K. Waldschmidt), North Holland Publ. Co., Amsterdam/New York, 1985

[GoL73] R. Goke, J. Lipovski: Banyan Networks for Partitioning Multiprocessor Systems; Proc. 1st Ann. Symp. on Computer Architecture, 1973; IEEE 1973

[GSch85] W. Graß, M. Schielow: VERENA: a Program for Automatic Verification of the Register Transfer Description; IFIP Int'l Symp. on CHDLs and their Applications, Tokyo, Japan, 1085; North Holland Publ. Co., Amsterdam/New York, 1985

[HaEIS86] R. Hartenstein, R. Hauck: Entwurf von Symbolischem Layout mit Werkzeugen der Register-Transfer-Ebene; in: (Hrsg.: H. Heckl, A. Kaesser, K. Koller, K. Woelcken) Entwurf Integrierter Schaltungen, 2. EIS-Workshop im Wissenschaftzentrum, Bonn, 1986

[HaHa86] R. Hartenstein, R. Hauck: Functional Extraction from Personality Matrixes of MOL (Matrix-oriented Logic) Circuits; in [ABA86]

[HHa86] R. Hartenstein, R. Hauck: The KARL System User Guide; CVT report, Kaiserslautern University, Kaiserslautern, F.R.G., 1986

[HaHa87] R. Hartenstein: The classification of HDLs; in [Har87]

[HaHi86] R. Hauck, A. Hirschbiel: A Physical Model Extension for KARL Simulators; in [ABA86]

[HaMa86] R. Hartenstein, A. Mavridis: RTcode Instant for KARL-III, second edition; report, Fachbereich Informatik, Kaiserslautern University, Kaiserslautern, F.R.G., 1986

[Har77] R. Hartenstein: Fundamentals of Structured Hardware Design - A Design Language Approach at Register Transfer Level; North Holland Publ. Co., Amsterdam /New York 1977

[Har87] R. Hartenstein (ed.): Hardware Description Languages; Elsevier Scientific, Amsterdam/New York, 1987

[HLW87] R. Hartenstein, K. Lemmert, A. Wodtko: KARL-III Language Reference Manual, report, Kaiserslautern 1987

[LaMa86] L. Lavagno, R. Manione: Automatic Layout Generation from (KARL) RT level descriptions; in [ABA86]

[Lem86] K. Lemmert: Steps Towards KARL Use as a Design Calculus; in [ABA86]

[Lem88] K. Lemmert: A Systolic Design System using KARL; in [ABA88] - [Lem89] Ph. D. Thesis under preparation

[LeNe87] K. Lemmert, W. Nebel: Conceptual Design based on HDL use; in [Har87]

[Mav87] A. Mavridis: Languages for Simulation and Tester Operation; in [Har87]

[Mel87] M. Melgara: Fault Simulation at Functional Level; in [Har87]

[MHM84] S. Morpurgo, A. Hunger, M. Melgara, S. Segre: RTL Test Generation and Validation for VLSI: an integrated set of Tools for KARL; IFIP CHDL'85, Tokyo, Japan, Sept 1985, North Holland, Amsterdam, 1985

[MGN79] G. Masson, G. C. Gingher, S. Nakamura: A Sampler of Circuit Switching Networks; Computer, June 1979

[MoM87] R. Hartenstein, A. Hirschbiel, M. Weber: MoM - Map-oriented Machine; in (ed.: T. Ambler, P. Agrawal, W. Moore) Hardware Accelerators for Electrical CAD; Adam Hilger Publ., Bristol/Philadelphia, 1987

[Neb86] W. Nebel: KARL Extraction from Layout; in [ABA86], and:, Proceedings IEEE COMP EURO, Hamburg 1986; IEEE, New York, 1986

[NN88] N. N.: KARL-related literature: reference list and order form; ABAKUS, Bau 12, Univ. Kaiserslautern; 1988

[RaGr86] M. Rauscher, W. Graß: Experiences in using KARL in PRIMITIVE (Program for Interactive Microprogram Transformation and its Verification; in [ABA86]

[SchG86] M. Schielow, W. Graß: KARL use in VERENA (Verification of RT Structures); in [ABA86]

[SM85] I. Stamelos, M. Melgara: CVT TIGER: a RT level Test Pattern Generation and Validation Environment; CVT report, CSELT, Torino, Italy, 1985

[SMP86] I. Stamelos, M. Melgara, M. Paolini, S. Morpurgo, C. Segre: A Multi-Level Test Pattern Generation and Validation Environment; International Test Conference, 1986

[TYF74] T Feng: Data Manipulating Functions in Parallel Processors and Their Implementations IEEE-Tr-C-23 (1974),

[TYF81] T. Feng: A Survey of Interconnect Networks; Computer, 1981

[Wel86] U. Welters: Specification of the MLED Multi-Level Editor; internal report, Fachbereich Informatik, Universität Kaiserslautern, Kaiserslautern, F.R.G., 1986

[Wel87] U. Welters: Graphic Hardware Description Languages; in [Har87]

[Wod87] A. Wodtko: RT Languages in Goal-oriented CAD Algorithms; in: [Har87]