Report 82-02
Stanford -- KSL

Scientific DataLink

The Partitioning of Concerns in Digital System Design. Mark J. Stefik, Alan G. Bell, Daniel G. Bobrow, Harold Brown, Lynn Conway, et al., Feb 1982

card 1 of 1

Heuristic Programming Project
Report No. HPP-82-2

February 1982

The Partitioning of Concerns in Digital System Design

MARK STEFIK, ALAN BELL, DANIEL BOBROW,
HAROLD BROWN, LYNN CONWAY, CHRISTOPHER TONG

HEURISTIC PROGRAMMING PROJECT
DEPARTMENT OF COMPUTER SCIENCE
STANFORD UNIVERSITY
STANFORD,CALIFORNIA 94305 AND
VLSI SYSTEM DESIGN AREA, XEROX PARC

The Stanford University component of this research is funded by the Defense Advanced Research Projects Agency contract MDA-903-80-007.

ABSTRACT

This paper* proposes the use of explicit abstraction levels to organize decision making in digital design. These levels partition the concerns that a designer must consider at any time. They provide terms and composition rules for the composition of structural descriptions within a level. This allows multiple opportunities for mapping behavior into structure.

A version of this paper was presented at the Conference on Advanced Research in VLSI, Massachusetts Institute of Technology, Cambridge, Massachusetts, January 25-27, 1982. Another version appeared as Xerox PARC report VLSI-81-3, December 1981.

*The Stanford University component of this research is funded by the Defense Advanced Research Projects Agency contract MDA-903-80-C-007.

BLANK PAGE

Introduction

The Mead and Conway textbook[1] presents a methodology for designing digital integrated systems which exploits the properties of charge-storage devices. As in most texts concerning design, the methodology is communicated via examples. It is clear from these examples that designers work within multiple design spaces ranging from abstract system descriptions to circuit layouts. At the layout level, the methodology is formalized in an explicit set of rules for composing primitive terms, "colored rectangles" representing material on a chip. If users follow these *Lambda* composition rules, their designs are guaranteed to have adequate physical spacing on a chip. However, the intermediate levels in the design process are not usually recorded in a formal notation and are only informally shared in the community of designers.

This paper proposes a number of formalized intermediate description levels, each of which allows a designer to deal with particular concerns. Each level provides a vocabulary of terms and a set of simple composition rules. This enables a designer to do *composition within a description level.* The rules make possible a principled approach for creating composite structures and ultimately digital systems. Designs expressed at multiple description levels interact through explicit constraints. We are developing transformation rules to take designs at one of these levels and express them at other levels.

To aid in testing these ideas, we are developing an expert design system, Palladio, which will assist a user to design with multiple levels of description. An active part of the system will suggest alternative implementations of a higher level design as well as possible optimizations. Palladio is being built in the knowledge engineering paradigm. *Knowledge engineering* is a branch of research in artificial intelligence that is concerned with the creation of knowledge-based *expert* systems[2,3]. The systems are called knowledge-based because their performance is a consequence of symbolic reasoning from explicitly represented knowledge, that is, facts and heuristics about the task domain. They are called expert when they perform tasks that require substantial expertise.

In this paper we provide an overview of our use of multiple description levels in a proposed design system. We discuss how our approach differs from silicon compilers and from traditional register transfer (RT) hardware description languages. We argue that our approach provides more leverage for systematic exploration of possible designs.

Getting Leverage from Abstraction

The principle of *divide and conquer* is an essential part of mastering otherwise overwhelming tasks. As Simon and many others have observed:

> To design . . . a complex structure, one powerful technique is to discover viable ways of decomposing it into semi-independent components corresponding to its many functional parts. The design of each component can then be carried out with some degree of independence of the design of others . . . [page 148][4]

That so many design systems have facilities for supporting this kind of hierarchical design suggests that the point is widely appreciated. But exploitation of a *component hierarchy* is only one of several opportunities for dividing up the design process. Design using abstraction levels is a complementary way to do it. The metaphor in force here is that design is search. Solutions exist in a potentially large space of possible designs. Design is a process of generating alternatives and testing them against requirements and constraints. Abstract solutions are descriptions that stand for an equivalence class of detailed solutions.

The design process can be characterized as a dialectic between goals and possibilities. Designers explore parts of the design space as driven by their current goals and they sharpen their goals as they learn what is possible. They decide how far the overall design should be completed before designs for particular subsystems should be developed. When designers work bottom-up on particular subsystems, they gain information about *what is possible* in isolated parts of the design space. When designers work top-down, they decompose designs to reflect subgoals. Sometimes a reformulation of subgoals yields a simplification of interfaces between subsystems.

This dialectic reflects the absence of a complete synthetic theory of design. Designers must begin without knowing exactly what they want or what is possible. In each design task the information gained in the search for solutions yields an informal theory, which accumulates with the experiences of the designers. In our knowledge engineering paradigm, we hope to provide a language by which designers can capture this emerging theory in Palladio.

Abstraction levels provide leverage for top-down processes by enabling a designer to deal with critical issues early and across the breadth of a design. This is a step towards a principled approach for dealing with critical considerations early in the design process and defering others. For example, in the abstraction levels in this paper computational issues are considered critical. Using these levels designers can work out certain storage and communication decisions before worrying about power considerations. At each level, designers make decisions about structures for implementing devices with the desired behavior.

Abstract descriptions have fewer specifications than complete solutions and are generated more quickly. We can explore more territory because we travel faster. Familiarity with useful kinds of tradeoff can guide the generation process. Following tradeoffs we can move from one possible description to another having different costs along some design dimension (e.g., communication vs. redundant computation). We explore more effectively because we know where to look. Another benefit of designing with abstractions is early pruning. For this there must be a cost metric that can be applied to prune particular descriptions. Using the metric to eliminate a description, we avoid pursuing the members of the equivalence class of detailed solutions.

Abstraction levels and constraints have been used before in expert systems, although this may be their first application in an expert system for the design of digital systems. Multiple levels of abstraction have been used for hypothesis generation and evaluation in several expert signal interpretation programs[5,6] and constraints have been used to represent subproblem interactions in a hierarchical planning program[7].

Examples of Description Levels

Figure 1 summarizes four experimental description levels that we are developing. Each description level has a set of *terms* that are composed to form systems and a set of *composition rules* that define legal combinations of the terms. The concerns of each level are characterized by specific classes of *bugs* that can be avoided when the composition rules are followed. Each level has a behavioral meaning as well as a structural meaning so that descriptions are also *runnable*.

Description Level	Concerns	Terms	Composition Rules	Bugs Avoided
Linked Module Abstraction LMA	Event Sequencing	Modules Forks Joins Buffers	Token Conservation Fork/Join Rules	Deadlock Data not Ready
Clocked Registers and Logic CRL	Clocking 2 Phase	Stages Register Transfer Transfer Functions	Connection of Stages	Mixed Clock Bugs Unclocked Feedback
Clocked Primitive Switches CPS	Digital Behavior	Pull-Ups Pull-downs Pass Transistors	Connection of switch networks Ratio Rules	Charge Sharing Switching Levels
Layout	Physical Dimensions	Colored Rectangles	Lambda Rules	Spacing Errors

Fig. 1 Description Levels for Palladio

The *layout description level* is used in many digital design systems. The terms at this level of description are regions (e.g., rectangles) of metal, polysilicon, and diffusion in nMOS. The composition rules at this level are a set of *Lambda rules* governing the sizes and spacing of the regions. The rules are intended to guarantee that there will be no spacing errors that would prevent correct operation of a fabricated system.

The following sections give simple examples of structural descriptions and rules at each of the more abstract levels that we are developing. The first level is substantially technology independent. The next two levels are specialized for two-phase clocking systems implemented in nMOS. These levels are intended to cover important design concerns and to admit some early decision making in design that is not prematurely constraining. Our purpose in discussing these levels is to provide concrete examples of multiple-level descriptions; the examples below provide a sampling of our current thinking.

The Linked Module Abstraction (LMA) Level

The *linked module abstraction* (LMA) level[8] is concerned with the sequencing of computational events in a digital system. It describes the paths along which data can flow, the sequential and parallel activation of computations, and the distribution of registers. Our formulation provides a simple closed covering of ideas from many sources including Petri nets[9] and the design of speed-independent modules[10,11].

Terms. The basic elements of this level are modules. Modules are computational elements that perform *complete* operations. Once a module has been started, it completes what it is doing before it can be restarted. Each module has a number of directional paths: a Go and a Done path for synchronizing communication with other modules, optional Input and Output data paths, and an optional Interrupt path. Each module also has a set of input buffers corresponding to the input data paths. A module is controlled by the absorption and emission of tokens on the Go and Done paths. Besides modules there are several kinds of forks and joins that determine the flow and control of information, calling buffers that mediate the use of shared modules, and subsystems which combine modules to form entities that are not modules (e.g. pipelines).

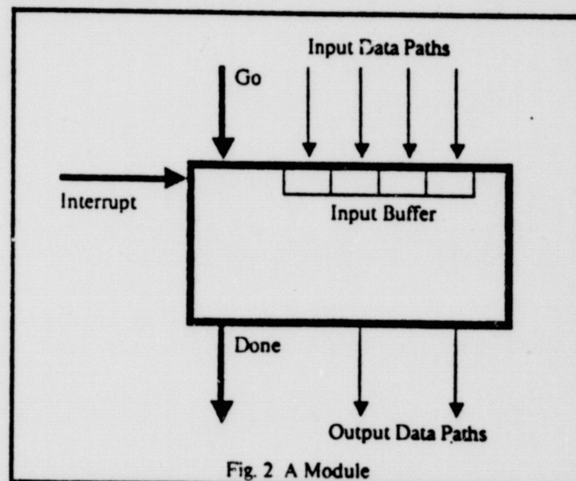

Fig. 2 A Module

<u>Composition Rules.</u> Composition rules at the LMA level govern the use of shared structures and provide conventions for control of information flow. The following are examples of composition rules.

> *Fork Rule.* If the output data paths of a module connect to more than one other module, then its Done path must be connected through a Fork to the Go paths of all of the other modules.
>
> *Join Rule.* If the input data paths of a module are connected to more than one other module, then its Go path must be connected through a Join to the Done paths of all of the other modules.
>
> *Shared Module Rule.* Calls to shared modules must be buffered.

The composition rules of the LMA level prevent the use of data before it is ready (e.g., has not settled) and prevent deadlock from the use of shared modules in the digital system.

<u>Examples from the Design of a Stack.</u> A major goal of the LMA level is to provide an expressive language for describing digital architecture. We believe that practicing designers do not share a common notation and that this makes it unnecessarily difficult to understand designs. This section argues that the LMA language is expressive enough to admit meaningful comparisons and abstract enough to provide leverage for exploring design alternatives.

Stacks are familiar devices for providing *last-in-first-out* access to stored data. There is a *push* command for adding a datum to the top of a stack, and a *pop* command for retrieving it. An error occurs for a push on a full stack or a pop on an empty stack. There are several fundamentally different architectures for implementing this simple device. This section considers five of them briefly as architectural examples.

Pointer Stack. This version corresponds to the usual software implementation. Information is stored in an array of registers. An index register (called the pointer) contains the address of the top of the stack. The push and pop instructions increment and decrement the pointer. The following shows the abbreviated graphical notation and linear notation for this version of the stack:

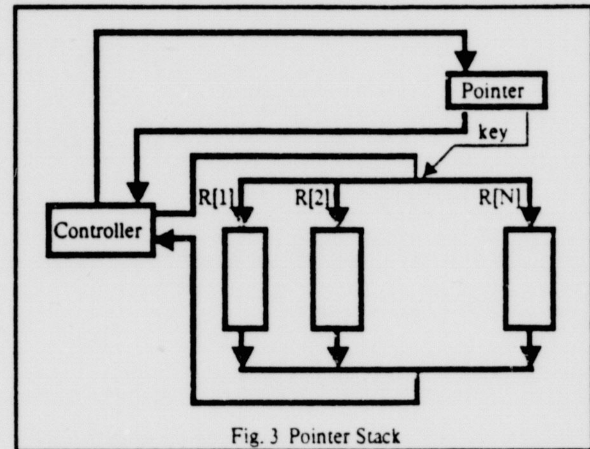

Fig. 3 Pointer Stack

```
Module PointerStack
inputs [command: action, dataIn: item]
outputs    [dataOut: item]
               --These specify typed input and output lines.
parameters     [depth: Integer, item: Type]
               --These are construction parameters.
components     [pointer: Register[addressWidth(depth)]
                       R: Set[1, depth, Register[item]]]
               --This specifies subcomponents of modules from a library.
               --Set(1,n,type) creates a set of n indexable items of type type.
action     [Case command
     [push     [if pointer<depth
                    then {select[key←pointer | R[key]←dataIn];
                    pointer←pointer + 1}]]
               --Select activates an indexed element from a set.
     [pop      [if pointer>1
                       then {pointer←pointer-1;
                            <select key←pointer; dataOut←R[key] |>}]]]
interrupt  [{pointer←1}]
end Module PointerStack
```

Roving Marker Stack. This version uses a *mark bit* associated with each storage cell to indicate the top of the stack. In a push or pop instruction, all of the cells receive the command but only the one with the mark bit set performs the operation and then moves the marker bit by calling a neighboring cell.

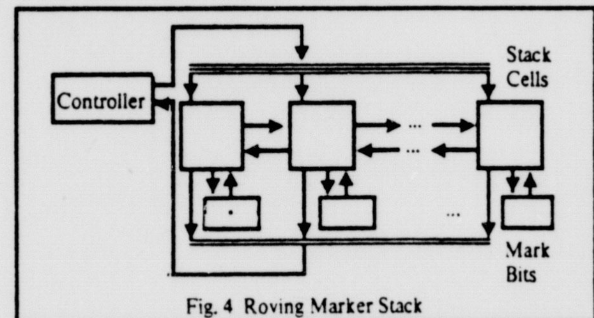

Fig. 4 Roving Marker Stack

Ganged Marker Stack. This is similar to the previous stack except that all of the marker bits are moved, instead of just the top ones. Architecturally, this stack is a mixture of the first two. It combines an array of registers for data storage with a shift register for marker storage.

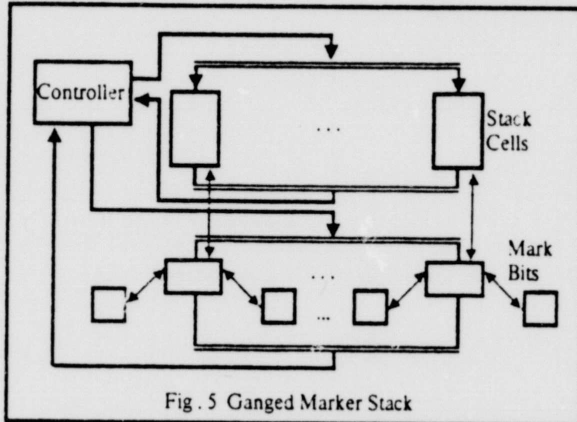

Fig. 5 Ganged Marker Stack

Buffered Stages Stack. In the previous versions, the cell containing the data at the top of the stack varied according to the number of pushes and pops. An indicator was used to keep track of which cell was the current top of the stack. In this implementation, the top of the stack is always the leftmost cell and all the data in the stack move simultaneously. Intermediate stages buffer the data as it moves been cells.

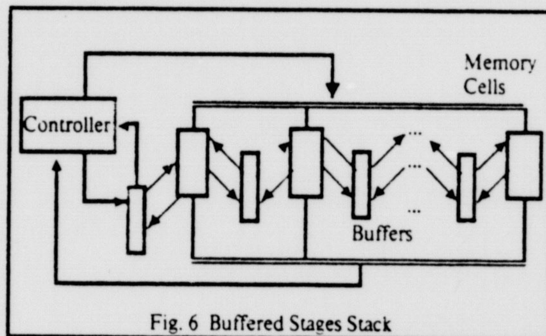

Fig. 6 Buffered Stages Stack

Ripple Stack. Like the buffered stages architecture, this version has a fixed cell for the top of the stack and moves the data on push and pop instructions. In contrast, the controller for this architecture needs only to be connected to the first and last stack cells. A push command from the controller starts at one end of the stack; the required movement ripples left to right through the stack. It requires half as many registers as the previous version of the stack, but requires time proportional to the current depth of the stack.

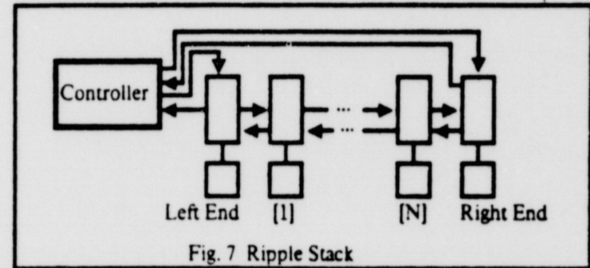

Fig. 7 Ripple Stack

```
Module RippleStack
inputs          [command: Action, dataIn: item]
outputs         [dataOut: item]
parameters      [depth: Integer, item: Type]
constants       [moveRight, startMoveLeft, pop, store: Action]
components      [ DC: Set[i, depth,
                        DataCell[item: item
                                 LNbr: when i>1 then DC[i-1],
                                 RNbr: when i<depth
                                       then DC[i + 1]]]]
action     [Case command
     [push      { DC[1](moveRight); DC[1](store, dataIn)}]
     [pop {dataOut←DC[1](pop); DC[depth] (startMoveLeft)}]
interrupt  [<& Set[i, depth] {DC[i](store, 0); DC[i](pop)}] &>}]
end Module RippleStack
```

```
Module DataCell
inputs          [command: action, dataIn: item]
outputs         [dataOut: item]
parameters      [item: Type, LNbr: DataCell[item], RNbr: DataCell[item]]
components      [Filled: Bit]
action          [Case command
     [moveRight       () ← ()          {if Filled=1
                                        then RNbr(moveRight);
                                        RNbr(store, dataIn)}]
     [startMoveLeft   () ← ()          {if Filled=0
                                        then LNbr(startMoveLeft)
                                        else {LNbr(moveLeft);Filled←0}}]
     [moveLeft        () ← ()          {if Filled=1
                                        then LNbr(moveLeft);
                                        LNbr(store, dataIn)}]
     [store           (dataIn) ← ()    {-- data stored in dataIn]
     [pop             () ← (dataOut)   {dataOut←dataIn; Filled←0}]]
end Module DataCell
```

<u>Architectural Comparisons</u>. The stack specifications in the LMA notation are sufficiently descriptive to provide a basis for answering questions like the following:

How much storage is needed per element of capacity?

> All of the stacks require at least at least one register per element. The roving marker stack, ganged marker stack, and ripple stack each require one additional bit per element. The buffered stages stack requires two registers per element.

What fanout of control logic is required?

> The ripple stack requires control connections only to the first and last stack elements. The other versions require connections to every stack cell.

What determines the minimum delay between successive push commands?

> The ripple stack has a delay which increases as the stack is filled. This is because the push command must ripple through to the last filled stack cell. (We have not worked out the details of a version of a ripple stack in which the first few elements would not have to wait for the ripple to finish; it would allow successive push and pop transactions to cancel without extensive rippling.) The time complexity of the pointer stack depends on the time complexity of the adder and of the random access memory.

LMA notation for distributed computing (i.e., *active architectures*) is analogous to high level programming languages for sequential and parallel algorithms. Specification at the LMA level highlights critical architectural tradeoffs such as communication versus redundant computation, copied structures versus shared structures, serial versus parallel computation. We believe that LMA programs are amenable to a physical theory of computation, perhaps along the lines of the entropy model in Mead and Conway[1] pages 365-370. This would provide an added abstract framework for space-time-energy complexity analysis beyond that now emerging for VLSI circuits[12].

The Clocked Registers and Logic (CRL) Level

The *clocked registers and logic* (CRL) level is concerned with the composition of combinational and register logic. We have considered only two-phase clocking systems. In the future we will develop an abstraction level based on self-timed systems.

To implement an LMA description at the CRL level requires making a number of design decisions. Event sequences need to be mapped into event times in ways that preserve their partial orderings. Modules need to be divided into stages and clocks need to be assigned. Encodings for symbols need to be chosen. In doing this, the performance and implementation constraints for the ultimate digital system need to be articulated and taken into account. We believe that this process is *knowledge intensive* requiring expertise about tradeoffs and the combination of constraints. This knowledge could be provided by either a human designer or by an expert system. One of our goals is to represent subsets of this knowledge for Palladio. This paper, however, is concerned only with the description levels.

Terms. The terms at the CRL level are stages. Stages are logical devices having both a storage capacity and a transfer function.

Composition Rules. Common design practice is to use alternate clocks to load data into successive stages. This practice is captured by the following two composition rules:

> All of the data inputs to a stage must be valid during the high interval of the same clock.
>
> All of the outputs of a stage must be valid during the high interval of the other clock.

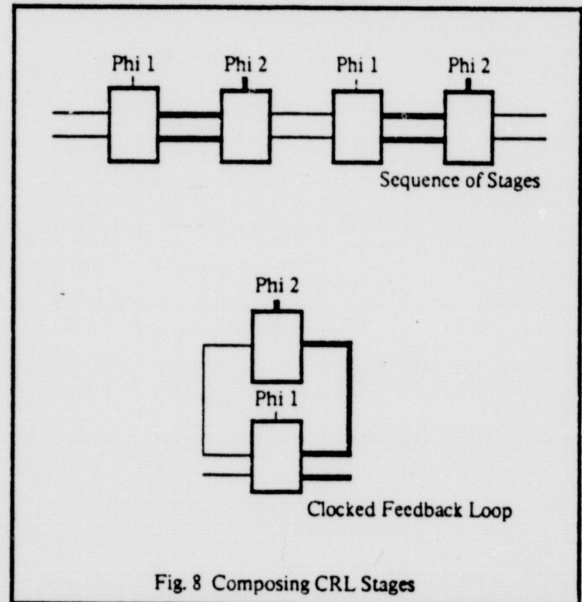

Fig. 8 Composing CRL Stages

Figure 8 uses narrow lines to indicate that data are valid during the high interval of Phi_1, and wide lines for data valid during the high interval of Phi_2. The composition rules prevent the creation of stages with distinct input lines holding data valid on different clocks (mixed clock bugs) and also the creation of unclocked feedback loops.

Optimization Rules. The CRL composition rules embody worst case assumptions. For example, figure 9 shows two versions of a circuit for a memory cell. Technically, the second version has an unclocked feedback violation. The optimization works because the two inverters return the original data. The optimization requires extra-level information, which is suggestive of the idea that global optimizations require passing information among abstraction levels.

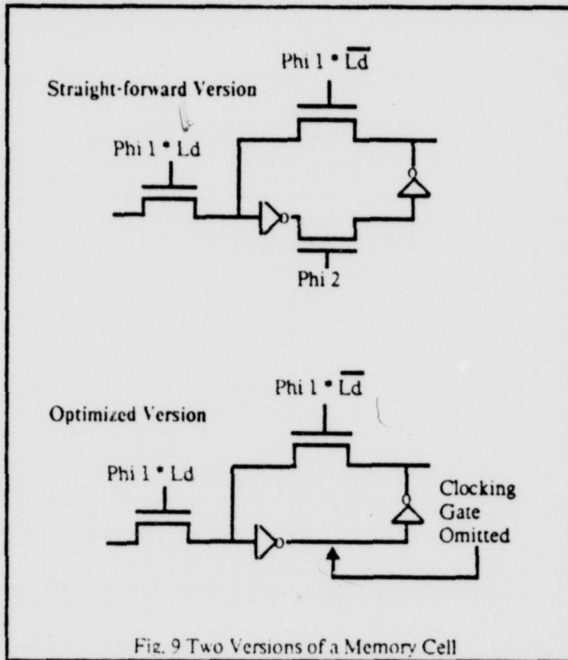

Fig. 9 Two Versions of a Memory Cell

The Clocked Primitive Switches (CPS) Level

The CPS level is concerned with the digital behavior of a system. This requires that circuits have two reliably distinct logical levels. For example, near 5 volts we have a band which we interpret as logical 1, and near 0 volts we have a band which we interpret as logical 0. Intermediate voltages in a digital system have an indeterminate interpretation at sampling times. Such intermediate voltages can be caused by improper interconnection of circuit elements, improper operating regions of devices, and leakage of stored charge. The concepts of the CPS level are closely related to switch-level simulations[13].

Implementation of CRL descriptions at the CPS level requires choices about using steering logic versus restoring logic, and choices about various regular structures such as PLA's and multiplexors. Power and performance constraints of the ultimate circuit need to be taken into account. Again, we believe that this process requires substantial knowledge and we plan to articulate and represent a subset of this knowledge in Palladio.

Terms. The terms at the CPS level are steering logic, clocking logic, and restoring logic. The basic element of steering logic is a pass transistor. We label its terminals as follows:

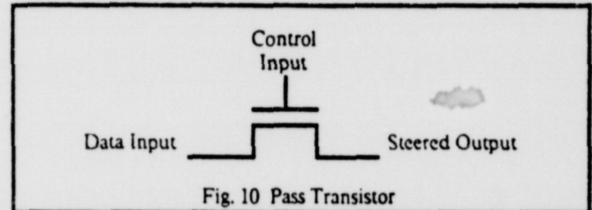

Fig. 10 Pass Transistor

We define a steering logic chain (SLC) as series-connected pass transistors and define a steered logic network (SLN) as the parallel composition of SLCs. An SLN can have several outputs.

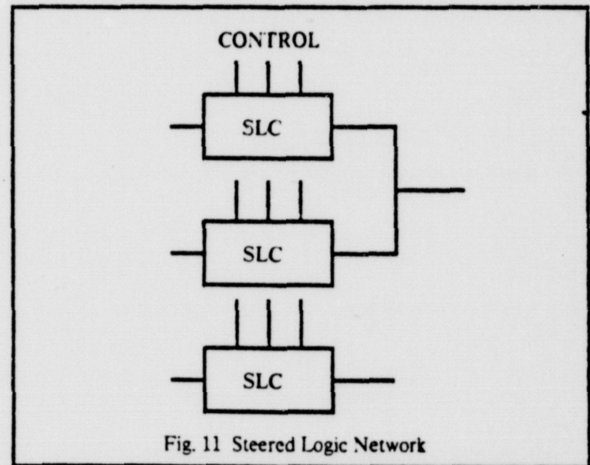

Fig. 11 Steered Logic Network

Clocking logic is a pass transistor controlled by a qualified clock input as shown.

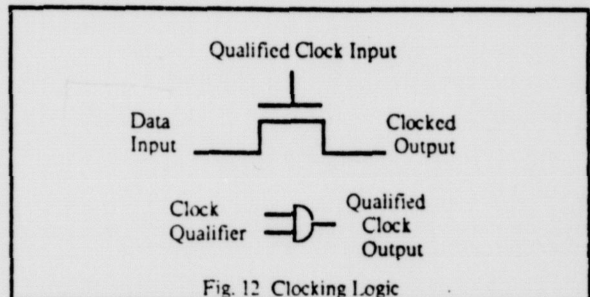

Fig. 12 Clocking Logic

The basic element of restoring logic is a *restoring element.*

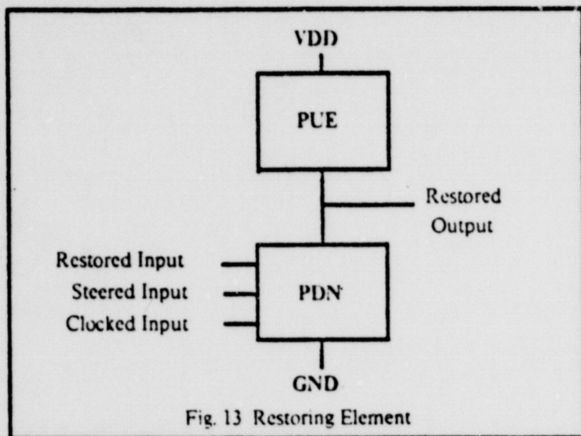

Fig. 13 Restoring Element

A restoring element is made of a *pull-up element* (PUE) and a *pull-down network* (PDN). A PDN has three kinds of inputs *restored, steered, or clocked.* The three kinds of inputs name the type of logic to which they are connected. The composition rules in the next section exploit the *type* information to help prevent errors.

Composition Rules. The composition rules at the CPS level specify how the terms can be connected and take into account requirements for voltage level restoration and charge storage. For example:

> A control input can only be connected to a restored output.
>
> A qualified clock input can only be connected to a clock output.
>
> Every SLN output must have exactly one *on* path from one SLN input during the period of validity of data output.

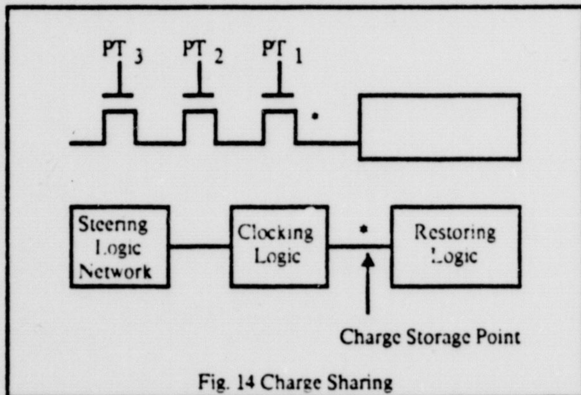

Fig. 14 Charge Sharing

Because of our rules, clocking logic must always separate switching logic from restoring logic. The purpose of this is to prevent charge sharing. Charge sharing occurs when charge is allowed to leak to or from a charge storage point, as illustrated in figure 14. For example, if PT1 is on, the charge can spread through PT1 into the capacitance of points further back. Since the amount of spreading can depend on the patterns of logical signals to the pass transistors, failures can seem intermittent depending on rare combinations of signals to a circuit.

Another set of composition rules constrain the layouts of pull-downs in terms of their impedances:

Element	Impedance Constraint
Pull-up element	$Z_{PUE} = L/W$
Restored PUE	$Z_{PUE} = L/W$
Clocked PDE	$Z_{PDE} = 2L/W$
Steered PDE	$Z_{PDE} = 2L/W$
Restored PDE	$Z_{PDE} = L/W$

The impedance of elements connected in series is computed as the sum of the individual impedances; the impedance of elements connected in parallel is computed as the maximum of the individual impedances. For the restoring logic to restore voltage levels correctly, the impedance ratio Z_{PU}/Z_{PDN} must be approximately 4.

Optimization Rules. According to the ratio rules above, the impedance for the following network would be computed as 8. In making this recommendation, the composition rules implicitly make a ***worst case assumption.*** If we had the information that signals on A1 and A2 were complementary and B1 and B2 were complementary, then we could compute the impedance of the network as 6.

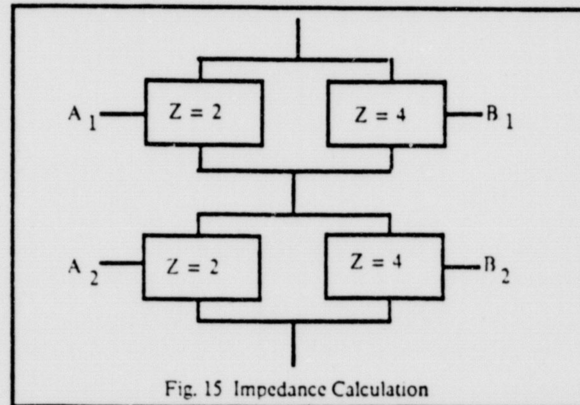

Fig. 15 Impedance Calculation

Comparison with Other Approaches

The philosophy behind our approach differs significantly from that used in the construction of silicon compilers. In a silicon compiler, the desired behavior of a system is specified in a language at a single level. The compiler converts this behavioral description into a structural description in a standard format. This fails to exploit many possibilities in the design space. In our use of multiple descriptions, each level specifies both behavioral and structural information. User-chosen transformations can be made in the design at many different levels all the way down. For example, in the LMA level, one might find optimizations which yielded substantially different structures before deciding whether to use two-phased clocking or self-timed circuits.

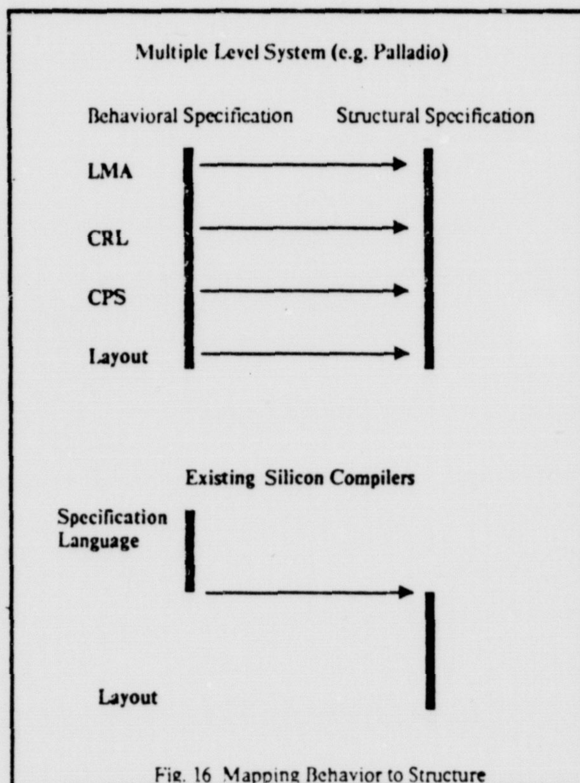

Fig. 16 Mapping Behavior to Structure

The use of multiple levels for describing hardware has been tried many times. For example, there are logic descriptions and register transfer descriptions[14]. We believe that the logic descriptions are too isolated and the RT descriptions are incomplete and insufficiently formalized. For example, it is difficult even to find the clocking specifications in a typical RT description. The composition of partial RT descriptions does not yield a *test of correctness* for clocking. In essence, those descriptions were not designed for synthesis. They provide no composition rules, optimization rules, or bug characterizations. Our goal is to understand and formalize descriptive levels whose utility derives from their coverage of critical design concerns.

Part of the culture of expert system building is the explicit representation of entities from problem-solving, such as goals, constraints, tradeoffs, and reasons. Symbolic expressions for these need to refer to circuit descriptions (e.g., modules or pull-down networks) as part of the natural bookkeeping of the design process. Such symbolic representations are a prerequisite to embodying expertise *about design* in an expert system. Figure 17 provides a sketch of how our abstraction levels will be used in Palladio.

Closing Remarks

The utility of these levels of description remains to be tested. Embedding them in Palladio and exercising them will be our reality test. We expect libraries of system descriptions at the abstract levels to be useful for collaborating designers by allowing some insulation from the inevitable technology changes that affect layout cell descriptions. Our preliminary experiments with hand worked examples have been very encouraging. Palladio will be a forcing function for articulating, representing, and sharing the heuristics and other knowledge about the design process.

We thank Chuck Seitz for pointing out to us the behavior-structure cross connection property of our multi-level description paradigm. Thanks also to Mary Hausladen and Terri Doughty for preparing the illustrations.

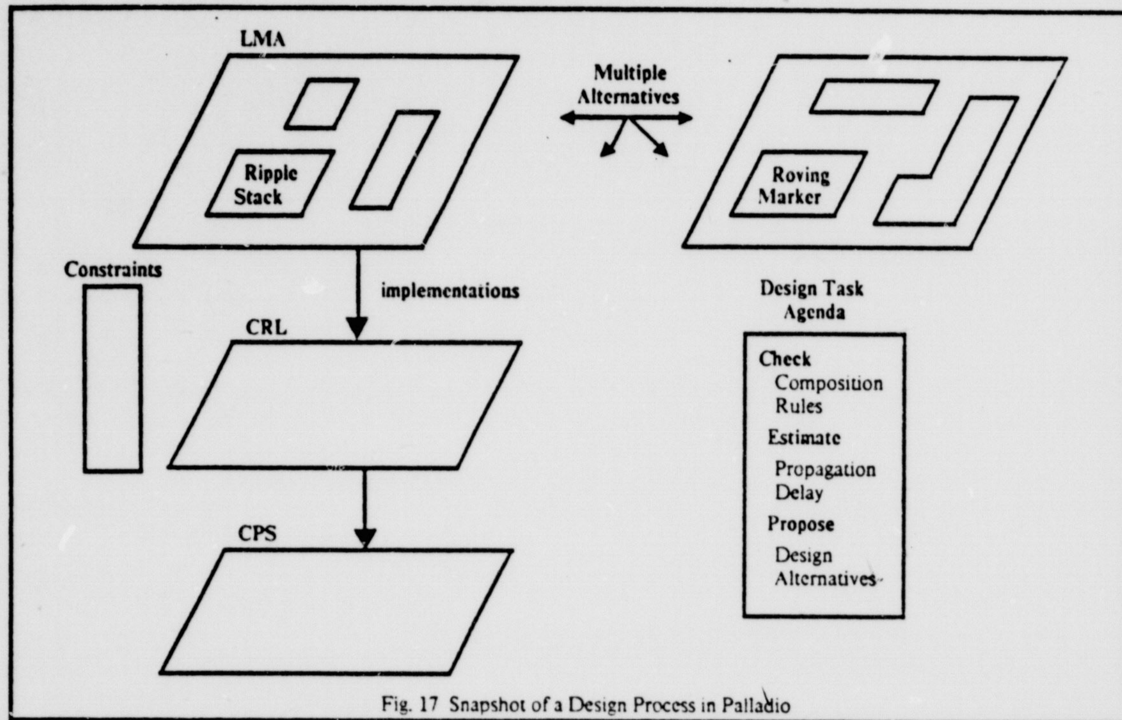

Fig. 17 Snapshot of a Design Process in Palladio

References

1. Mead, C., and Conway, L. *Introduction to VLSI Systems.* Addison-Wesley Publishing Company, 1980.

2. Feigenbaum, E. A. The art of artificial intelligence: themes and case studies of knowledge engineering. *Proceedings of the National Computer Conference*, AFIPS 1978, pp. 227-240.

3. Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., Sacerdoti, E. The organization of expert systems: a tutorial. *Artificial Intelligence* (in press), 1982.

4. Simon, H. A. *The Sciences of the Artificial.* Cambridge, Massachusetts: The MIT Press, (second edition) 1981.

5. Erman, L. D., Hayes-Roth, F., Lesser, V. R., Reddy, D. R. The Hearsay-II speech-understanding system: integrating knowledge to resolve uncertainty. *ACM Computing Surveys* 12:2, June 1980, pp. 213-253.

6. Nii, H. P. and Feigenbaum, E. A. Rule-based understanding of signals. in Waterman, D.A. and Hayes-Roth, F., Eds., *Pattern-Directed Inference Systems.* Academic Press, New York, 1978, pp. 483-501.

7. Stefik, M. Planning with constraints (Molgen: part 1). *Artificial Intelligence* 16:2, May 1981, pp. 111-140.

8. Stefik, M. & Bobrow, D. G. Linked module abstraction: A methodology for designing the architectures of digital systems. (working paper), Knowledge-Based VLSI Design Group KB-VLSI-81-9, 1981.

9. Peterson, J. L. Petri nets. *Computing Surveys*, 9:3, September 1977, pp. 223-252.

10. Clark, W. A. Macromodular computer systems. *Spring Joint Computer Conference*, 1967.

11. Keller, R. M. Towards a theory of universal speed-independent modules. *IEEE Transactions on Computers*, C-23:1, January 1974.

12. Thompson, C. D. Area-time complexity for VLSI. *11th Annual ACM Symposium on Theory of Computing*, 1979, pp. 81-88.

13. Bryant, R. E. *A switch-level simulation model for integrated circuits.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology MIT/LCS/TR-259 (March 1981).

14. vanCleemput, W. M. *Computer-Aided Design Tools for Digital Systems.* IEEE Catalog No. EHO 132-1 (second edition) 1979.

Copyright © 1985 by KSL and
Comtex Scientific Corporation

FILMED FROM BEST AVAILABLE COPY