MASSACHUSETTS INSTITUTE OF TECHNOLOGY
ARTIFICIAL INTELLIGENCE LABORATORY

AI Memo No. 526 May 1979

Computer Aided Evolutionary Design for Digital Integrated Systems

by
Gerald Jay Sussman, Jack Holloway, and Thomas F. Knight, Jr.

Abstract:

We propose to develop a computer aided design tool which can help an engineer deal with system evolution from the initial phases of design right through the testing and maintenance phases. We imagine a design system which can function as a junior assistant. It provides a total conversational and graphical environment. It remembers the reasons for design choices and can retrieve and do simple deductions with them. Such a system can provide a designer with information relevant to a proposed modification and can help him understand the consequences of simple modifications by pointing out the structures and functions which will be affected by the modifications. The designer's assistant will maintain a vast amount of such annotation on the structure and function of the system being evolved and will be able to retrieve the appropriate annotation and remind the designer about the features which he installed too long ago to remember, or which were installed by other designers who work with him. We will develop the fundamental principles behind such a designer's assistant and we will construct a prototype system which meets many of these desiderata.

Keywords: Computer-aided design, integrated circuits, VLSI, dependencies, constraints, engineering problem solving, layout.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's Artificial Intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643. This work was supported in part by the National Science Foundation under Grant MCS77-04828, and in part by Air Force Office of Scientific Research Grant AFOSR-78-3593.

The Problem

The integrated circuit revolution has led to a vast increase in the complexity of the electrical artifacts which can be constructed monolithically. In the design of hardware systems, we are rapidly approaching the complexity barrier which has for long been apparent in the design of software systems. The turn-around time for realization of a new design, from conception, through synthesis and debugging has become excessive; hence we are not developing new designs at a reasonable rate. This is not particularly a problem of integrated circuits, or of programming systems, but rather a fundamental problem which can best be viewed in a larger context. There are inherent limitations to the complexity that the unaided designer can control in any engineering situation -- from a complex electrical system to a space vehicle or a nuclear power plant. The thrust of our proposal must be viewed as attacking the problems associated with integrated systems from this larger context.[1]

The evolutionary nature of large engineered systems is a crucial feature of their complexity. The specifications change, the design changes, and as bugs are discovered, the implementation changes to correct them. The changes are required because it is not possible for the designers, or the potential users of a system, to foresee all of the opportunities for using the system. Also, the environment in which the system will operate is itself subject to change. Besides this external reason for the evolutionary nature of large systems, there is also an internal reason. If all of the relevant constraints were considered at once in order to arrive at a perfect solution in the first place, the details would overwhelm the designer's cognitive abilities. A more effective strategy is to start with a solution which is reasonably close to correct and modify it repeatedly until an acceptable solution is reached.[2]

What is needed is a computer aided design tool which can help an engineer deal with system evolution from the initial phases of design right through the testing and maintenance phases. We imagine a design system which can function as a junior assistant. It provides a total conversational and graphical environment. It remembers the reasons for design choices and can retrieve and do simple deductions with them. Such a system can provide a designer with information relevant to a proposed modification and can help him understand the consequences of simple modifications by pointing out the structures and functions which will be affected by the modifications. The designer's assistant will maintain a vast amount of such annotation on the structure and function of the system being evolved and will be able to retrieve the appropriate annotation and remind the designer about the features which he installed too long ago to remember, or which were installed by other designers who work with him. We will develop the fundamental principles behind such a designer's assistant and we will construct a prototype system which meets many of these desiderata.

Engineering Problem Solving

One necessary subgoal of our integrated system research program is to further develop our theory of how skilled people (such as engineers and technicians) understand deliberately constructed technological artifacts. In most engineering disciplines there is already an extensive theory of how the physical principles which underlie the operation of the artifacts are applied in any particular design. In fact most of the formal knowledge taught in engineering classes is a (mathematical) theory of how the artifacts work -- how their behavior may be derived from fundamental physical principles. But an engineer knows much more than just the physical principles and their consequences. He has a great deal of "tacit knowledge" which allows him to apply his physical knowledge efficiently to solve problems of design, synthesis and analysis. This tacit knowledge is not taught explicitly in engineering classes nor is it written in engineering texts. It is usually considered informal and unteachable, except by actual experience.

There is almost no formalized theory of how the engineer himself operates -- how he must proceed in evolving a design when given a set of requirements or even how he must proceed in understanding an existing design. There is a "competence theory" of the engineered structures, but there is no "performance theory" of the engineering process[3]. This is not surprising. The performance theory is fundamentally imperative, but before people began to study algorithms as a subject there were no formal languages in which it was convenient to express such a theory. In fact, before this time it was not even realized that such languages were necessary. The advent of programmable computing machines placed great emphasis on the development of convenient and expressive formalisms for describing procedures. We have developed performance theories for some aspects of engineering. Such a theory is a set of rules which guide the behavior of engineers. We test our theories by implementing computer programs based on these rules which model the behavior of engineers. Successful theories are directly of practical value because they automate newly understood parts of the engineering process and can thus be turned into engineering tools.

The development of a theory of engineering performance knowledge is of considerable significance.

> 1. Understanding this currently tacit knowledge will result in the construction of powerful computer-aided systems for automating the routine aspects of design, construction, testing, and maintenance of complex systems. Such aids cease being luxuries and quickly become essential as the complexity of systems increases. We are already beginning to hit the complexity barrier in the long turn-around time for design of integrated circuits. We have long been on the wrong side of

this barrier in the design of large software systems.

2. Making the tacit knowledge of engineers more explicit will result in the development of more effective design methodologies. We are now in the descriptive phase of development of our theories. Predictive results will improve both computer-driven and human performance in developing complex systems.

3. Making the tacit knowledge of engineers more explicit will improve our ability to describe, explain, and teach the process of engineering.

4. Engineering design is an almost ideal domain in which to learn about how experts reason, and how students learn to be experts. Much of the actual competence knowledge is already formalized. Answers produced by a performance theory are thus testable. Much of the structure of, and the motivation for the performance theory is already in place as engineers have an extensive vocabulary of informal descriptions of what they are doing.

5. Results obtained in the study of design methodology for digital integrated systems may be applicable in other problem domains.

Why We Need A Sophisticated Theory of Design

The basic strategy of coping with a complex problem is to find or impose structure on the problem which allows breaking it up into manageable pieces. Each piece can then be worked on separately. This must be done so that a solution to the whole can be composed from the solution to the parts of the problem. Often a system can be partitioned into pieces which are more or less disjoint and which together cover the entire system. The total system can be understood by combining our understanding of the pieces and our understanding of the composition by which the pieces constitute the system. Similarly, each piece may be further partitioned. In this way we derive a single tree-like decomposition of the system -- a hierarchy.

This observation has resulted in a plethora of shallow methodologies which are collectively called "structured design".[4] In structured design the system under development is conceptualized as a single hierarchy where the system is recursively broken into parts, each of which represents a particular segment of its ultimate structure. These theories provide considerable power in organizing the thoughts of designers and in structuring computer-aided design systems, but they must ultimately break down in sufficiently complex real designs.

The problem is that, in sufficiently complex systems, at any stage there is usually more than one way of usefully partitioning a segment. If this is so, then a single hierarchy does not suffice to indicate all of the conceptual pieces of interest in the system. Pieces whose sub-pieces are localized by one decomposition will have those sub-pieces widely dispersed throughout another. Additionally, a single sub-piece may play several roles in each decomposition it appears in.[5]

For example, when designing a simple microprocessor, one way to proceed is to think in terms of a state-machine controller which is used to control a set of registers and data paths. The state machine may be implemented as combinational logic and a state register. In some technologies, e.g., two-phase clock dynamic MOS, a register may be expanded as a pair of linked, clocked inverters and a portion of the combinational logic may be done on each phase of the clock. Thus, in this technology there may be no single physical realization of the state register localized on the chip.

Suppose, further, that we want the registers which are controlled by our state machine to be bussed together. The bus is a real conceptual entity about which the data paths are organized. We must have a description of the register array in which the bus is a localized concept so that we can say specific things about it. For example, we may want to make assertions which constrain the communications conventions. However, in a structural hierarchy there is no particular locale for the bus because the bus is structurally distributed throughout the register array.

Even worse, consider the high level block labeled "instruction decoding" in a hierarchical description. Not only is the logic for this box physically distributed, but it also is implemented with techniques which overlap other aspects of the decomposition. A good example is the selective gating of clock signals, overlapping a clock distribution function with a decoding function. Other decoding may be integrated as part of other functional modules in the system. The decoding of the arithmetic function field, for instance, may be an integral part of the structure of the arithmetic logic unit.

Thus one aspect of developing a more powerful theory of the design process than "structured design" is the development of descriptive mechanisms which capture the power of the decomposition strategy without the restrictions on what can actually be expressed imposed by a simple hierarchical development. These problems with structured design theories are not in any way restricted to the world of digital integrated systems. Engineers in any discipline need to examine the systems they are designing from many points of view. A electrical circuit designer is often interested in the bias model of a circuit, the incremental model, the low frequency model of the incremental model, and the high frequency model of the incremental model, the noise model and the power distribution model. Each of these viewpoints imposes its own decomposition of the system under examination, and each provides structure and information to processes working from other viewpoints.

Our Developing Theory

We are developing a performance theory of engineering design, that is, a set of rules which characterize the way in which engineers behave when confronted with a design problem. In order to express this theory we are developing a methodology adequate to capture these rules. We call this the Design Procedure System because we will use it to express the procedures which an engineer will go through in the routine development of a design. The design language has two components: the design procedure language and the design plan language. The design procedure language is a very high level language for expressing design procedures -- the sequences of actions a designer goes through in evolving a design. The design plan language provides special data structures for representing the state of a design. These are used for representing the data on which the design procedures operate.

The design plan is essentially a data structure which describes the object being designed at many levels of detail and which captures the various models which are applied to it. The design plan provides locales to hang information such as why a particular goal, say a multiplexer, was implemented in a particular way, rather than using alternative approaches. The design plan contains active data structures, called constraint boxes, which autonomously check and criticize certain aspects of the evolving design and which compute some properties of the design as consequences of others by a process we call propagation.[6]

The language of design plans is crucial to the success of this project. It must be rich enough to allow the description of complex entities which are not entirely hierarchical. It must be possible to capture the various decompositions of a system that a designer wants to think about. An entity may be described in terms of several alternate decompositions into parts. In fact it may have different names from different viewpoints. Additionally, we must be able to specify that a structure is an instance of a prototype which is an element of a previously defined class from which it inherits structure, appropriate procedures, and decompositions, and that classes may be themselves subclasses of other classes.[7] We already have considerable experience with the development of a language of design plans in two domains which are related to integrated systems -- electrical circuits and LISP programs that manipulate data bases.

The design procedure language is concerned with formalizing the particular tasks that must be performed when attempting to develop a design plan. These tasks are described in terms of rules. The language provides design procedure primitives and means of combining simple design procedures to make compound ones. It also provides abstraction mechanisms which allow one to wrap up and generalize a particular design procedure developed in a particular design. Some of the rules that must be expressed are synthesis rules which tell how particular goals may be implemented. Other rules are for information gathering. These perform analysis on partially instantiated structures. Other rules impose constraints, or critics. A critic watches for and complains about violations of rational form or violations of constraints that must be enforced. Other constraints are used to deduce some design parameters from others by propagation of constraints.

Every deduction made by any design procedure or constraint must be annotated by the name of the procedure which made the deduction and the data which went into that particular deduction. We believe that it is essential that the design system which we are envisioning be thoroughly responsible for its behavior. This is essential to the debugging of the design procedures and also it is essential to the control of the deductive system so that we can retract any assumption and all of its consequences. It is critical in the context of evolutionary design. These dependencies are also very useful to a user who wants to discover why the system believes what it does about his design -- especially if it is reminding him of some detail which he has forgotten. We have had considerable experience now with dependency-controlled data bases.

The language of design must be powerful enough to capture such subtle notions as a methodology. For example, the single level polysilicon NMOS process places enough restrictions on relationships between wire levels that we can quickly develop an idea of a "reasonable" geometric methodology. The ground and VDD wires carry power, and except perhaps at their deepest branches, are required, for reasonable voltage drops, to be run in low resistance metal. The direction of these ground and VDD lines defines a local coordinate axis. Other metal lines must be routed parallel to the power wiring, to avoid crossing it. We need a set of wires which can locally travel at right angles to the metal wiring. Either polysilicon or diffusion would serve the purpose. But, in the silicon gate technology, transistors are formed wherever there is polysilicon over diffusion. If both poly and diffusion are to be used as wires, their predominant axes must be parallel, lest unwanted transistors will be formed at their crossovers.

Lead bonding constrains locations of I/O pads to the periphery of the chip. Long standing convention defines the location of certain pads, such as power, ground, and clocks.

With a few simple constraints, we have arrived at a very clear picture of what the major wire orientation of almost any large NMOS circuit is likely to be. We will develop a way to express such geometric conventions, in the same way we describe a methodology for logic design. In the NMOS design situation, the choice available in this methodology is very small. With multi-level metal or a different process, the designer may independently be able to specify a process, a geometric design methodology, and a logic design methodology.

Our Design System

Our computer-aided design system is to be built around the design procedure system and an appropriate library of design procedures and design plans. We expect that designers using our system will develop additional plans and procedures which will be added to the library and thus shared with other members of the design community. A designer may at any time either generalize or specialize a procedure or plan for his immediate use. Each element of the library will be indexed for easy reference, and will be annotated by information describing how it was derived, by the application of procedures to other design plans and procedures.

The design plan/procedure languages form an extensible base upon which the designer builds a vocabulary of cliches -- a new language which he then uses to describe his system. This is not just a hardware description language, though it is certainly powerful enough to describe hardware structures. In fact it is a language in which one describes methods of design. The designer tunes the methodology to meet the architecture being implemented. Part of the design specification is generated automatically by instantiating customized abstractions. These fragments can be the basis of a library of commonly used functions and procedures.

One way of providing flexibility in either layout or logical structure is by associating design procedures with generic design plan fragments. The design languages allow one to write custom design procedures that are local to a fragment. In this way it is possible to craft very general abstractions. Simple methods may in fact be just the instantiating and interconnecting previously defined chunks of hardware. One may define more advanced methods such as "the method of running a power wire through a particular kind of register cell" or "the method of computing the pull-up ratio for driving a particular capacitive load".

This is preferable to having a macrocell library consisting of a plethora of minor variations on one theme. The design procedure language provides expressive constructs for developing appropriately tailored instantiations of the general concept. It has control structure, a sub-part naming mechanism and a vocabulary of methods for synthesis and modification. A design procedure may either construct an expansion or modify a prototype according to the parameters of the call. For example, the generic concept "multiplexor" should suffice to implement instances that vary in the number of bits, control buffering, or select decoding method. There should also be flexibility in the layout to accommodate different spacing in the array of input lines.

The design plan language is used to represent the state of a design. We envision the designer as using design procedures to manipulate the current design plan. He may refine it, modify it, extend it, study it, analyze parts of it, and use it to help debug and test hardware described by the plan. The design state of a functional block is a mixture of the instantiation state of the fragment, associated mask layout, constraints relative to other structures, design decisions, annotations, and violations. Some parts of the design may be completely specified, but others may only exist as uninstantiated or even unspecified fragments. A group of elements can be represented by an abstraction that captures the important external shape and connections, or by an abstraction which captures the essential functionality, but suppresses the internal detail. Deductions such as rough estimates of propagation delay or chip area can be made in the face of incomplete information. This allows the designer to use a top-down approach to a complex design problem when this is appropriate.

In order to allow general design procedures there must be a uniform naming mechanism by which a conceptual entity may be referenced relative to the some current focus of attention. To this end every conceptual entity, be it a physical object or location, or a functional object, has an explicit data structure which designates it. This data structure may have several names, but it has at least one name by which it may be referenced in a uniform manner by any design procedure. Thus there are no "hidden variables" or implicit references in the system. This allows us to attach information (assertions, properties, constraints) to any object, facilitating complete documentation of the design plan.

Any fact or value known by the system has a justification which describes <u>why</u> the system believes it. These justifications must be either that the fact was tendered by a user or that it was derived by some design procedure (or constraint) from other known facts. These justifications make it possible for a user or design procedure to consider or make incremental modifications to a design, without disturbing features of the design which are not dependent upon the incremental modification. They allow the user, in an evolutionary setting, to consider consequences of minor modifications. They also allow a user or a design procedure to determine what assumptions any fact depends upon, and how.

The system allows multiple alternate representations of the same entity, and it allows these representations to communicate. In many cases some of the representations are the results of applying design procedures to other representations. For example the maze router may be applied to a partially specified layout of a circuit segment to produce a further specified layout of that segment. Each of these representations is, however, independently manipulable by further design procedures (or by the user calling the design procedure primitives). So if the engineer really doesn't want a particular wire routed by the router to go where it put it, he can change it in the representation which was the output of the router. This will have the effect of updating the justifications in the connections between the two representations so that the new representation will be thought of descending from the old representation through the maze router except that the particular wire was changed.

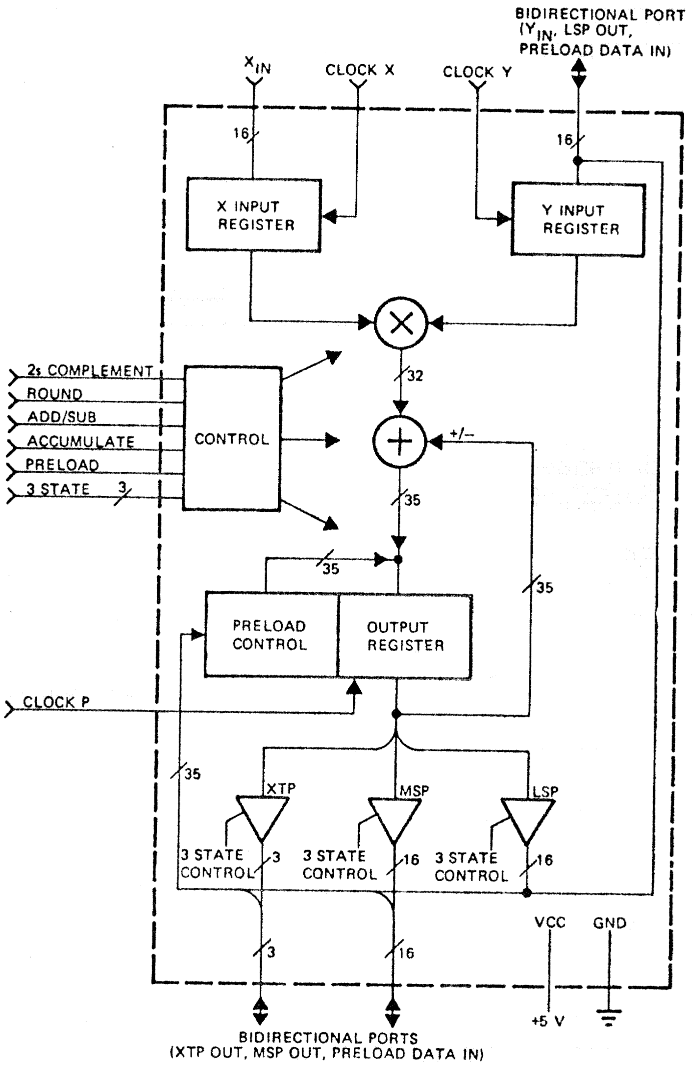
An Example of Design

We now display an example of the type of behavior that we expect from our proposed computer aided design system. While we are not entirely sure of the detailed implementation of the capabilities which we indicate, we feel confident that they are feasible.

Consider the problem of designing a parallel multiplier-accumulator which might be a component of a signal processing chip. The designer first produces a rough data path diagram (see below) showing the interconnection of instances of fragments such as registers, adders, and multipliers. This captures one decomposition that seems to be functionally appropriate and which perhaps will reflect the physical layout of the device as well. However, certain information is missing or incompletely specified.

Some deductions can already be made to fill in some of the missing information. For example, consider the adder in the middle of the block diagram. It is shown to have a 35 bit input and a 32 bit input. The adder fragment type can immediately infer the length of the adder chain and thus estimate the maximum time delay, the approximate area that the adder will take up, the power that will be needed to fuel the adder, etc. These approximate deductions can be made without further examination of the details of the adder chain because of default assumptions stored with the adder fragment. Similar deductions can be made about other blocks in the block diagram. This information can be combined to give estimates of total delay, power consumed, and area needed -- all without further refinement. In fact, we can simulate the circuit at this level of detail enough to determine that the proposed design actually has a chance of working.

Simple qualitative analysis of the circuit at this level of detail can point out potential problems and situations which will have to be attended to later. For example, critics associated with the adder fragment will notice that the instance shown does not have the same number of bits on both inputs. This



either is an error or will require that the designer decide how to rectify the anomaly. The system will notify the designer of the problem and add it to the queue of pending problems which must be solved before the device is considered ready for cooking.

Next the designer directs the system's focus to a particular part of the design. Normally this is what he thinks will be the most constraining segment. For example, one common design goal in constructing circuits is to compose regular arrays of logic by abutting them. This requires that a common pitch be found. The designer is now going to try to think about his structure in terms of the pitch of the regular substructures which he will have to come up with. This is a different decomposition of the problem from the original one, and will impose different submodule boundaries on the overall device. The first stage is to examine the pitch of the default layouts of the fragments which exist in the original conception of the problem.

We now enter geometric layout space where we are given a bunch of pieces which are the external boundaries of the default layouts with the associated pitch indicated. We place the pieces in such a way as to try to limit the problem of interconnection. A good first idea (which is the default layout that the system starts us up with) is the layout implied by the block diagram. We now abut the pieces and try to adjust the pitch of the abutting pieces so that they match. We do this by communicating the constraint imposed by the unit with the largest pitch to the other fragments, which will perhaps modify their default cell layouts. This constraint is noted so that any later changes to one of the fragments will trigger a check for violations of the pitch correspondence.

A fragment has a choice of techniques by which to respond to a request to adjust its pitch. There may be a general expert that can stretch simple cells, preserving their functionality. Failing that, the fragment may contain a specific design procedure that can modify the placement of some parts in the layout prototype in order to produce a new layout with the needed pitch. Or the designer may have specified internal seams in the specification of the cell where it may be stretched without interfering with the functionality, and where wires that pass through the cell may be added. Or it may choose among a set of predefined cell layouts that the community of designers arranged to be available under the generic fragment type. Finally, the designer can interactively modify a cell to produce a new layout with the correct pitch. Of course, this interaction may be postponed at the designer's choice, with the system maintaining a record of deferred problems.

While the abutting strategy will complete many of the signal connections, there are others that will require explicit routing. For some of the cases where not much optimization is possible or desired, a simple maze router is provided. In other cases, such as the interconnections among the output register, its preload control, and output buffers, some preplanning is required. Here, a small set of bus signals is shared among the fragments in such a way that we suspect that merging the three original blocks into one will eliminate the need for explicit interconnecting wires. We create a new cell type by editing together the corresponding layouts of the cells that we are merging. Note however that this will change the basic pitch of the system, triggering the adjustment of the pitch of the coupled fragments.

Observe that this merging operation has apparently destroyed the module boundaries implied in the original block diagram. This does not mean that this block diagram is wrong or should be discarded. It is still the right way of describing the functionality of the device, but the same simple hierarchical decomposition no longer suffices to describe both the logical and physical structure. The system must be able to manage this and, for instance, be able to give the correspondence between signal nodes in the logical and physical representations.

All through this process we assume that the system is monitoring various distributed constraints. For example, the actual propagation delay is compared to the design goal value. The dimensions of the power distribution lines depend upon the current estimate produced by the electrical modeling of the system. Changes may result in a readjustment of the line width or a design problem complaint. This monitoring is motivated by an assumption that it is probably effective to start with legal layouts and preserve the design correctness by the enforcement of constraints.

We have used such notions as "the adder fragment" without ever giving an example of what we expect such a piece of the design plan language to look like. Let us now examine the adder fragment in some detail to get a more concrete idea of what we have in mind. At the block diagram level an adder abstractly has three terminals, each of which is a word composed of a number of bits. The number of bits in each word is the same. The three terminals are called the addend, the augend and the sum. The addend and augend are inputs and the sum is an output. There is an extra bit coming out of the adder which is the carry-out and there is an extra bit coming in called the carry-in. We notate this block-diagram level description as follows:

```
(body adder (model: block-diagram)
      (parts: (addend word input)            ;declaration of parts
              (augend word input)
              (sum word output)
              (carry-out bit output)
              (carry-in bit input))
      (constraints: (= (length addend)       ;a constraint
                       (length augend)
                       (length sum))))
```

This fragment made use of another fragment, `word`, which is an ordered sequence of bits whose length is the number of bits.

```
(body word (model: block-diagram)
      (parameters: (length number)
                   (bottom number)
                   (top number))
      (sequence (enumerator: n)
                (low: bottom)
                (high: top)
                (generic: (parts: ((bit n) bit))))
      (constraints: (= (- top bottom)
                       (- length 1))))
```

The adder fragment has associated with its block-diagram level several implementation strategies. The simplest is the sequence of full adders such that the carry-out from each significant bit enters the carry-in from the next significant bit. Another implementation strategy is the carry look-ahead adder. We will only show the simple strategy here. Each strategy is attached to the fragment in the same way.

```
(implementation adder (model: block-diagram)
                (strategy-name: simple-sequence)
                (sequence (enumerator: n)
                          (low: 0)
                          (high: (- (length addend) 1))
                          (generic: (parts: ((f n) full-adder))
                                    (= ((bit (+ n (bottom addend))) addend)
                                       (a1 (f n)))
                                    (= ((bit (+ n (bottom augend))) augend)
                                       (a2 (f n)))
                                    (= ((bit (+ n (bottom sum))) sum)
                                       (s (f n))))
                          (between: (= (co (f *lower*)) (ci (f *upper*)))))
                (= carry-in (ci (f 0)))
                (= carry-out (co (f (- (length addend) 1)))))
```

You may have noticed that these descriptions of various aspects of an adder have parts that seem procedural. This is not accidental. There is a fundamental duality between object descriptions and procedures. Here we see that some aspects of an object involve design procedures which describe how to make a data structure describing that aspect of a particular object.

Complex Design Procedures

The unique aspects of our approach to the computer-aided design of integrated systems are illustrated by the use of design procedures which are considerably more complex than ones which just instantiate parts and connect them together. For example, we can make design procedures which assign propagation delay constraints to the full adders in the implementations shown above to make it possible to get estimates of the propagation delay for the adder. This can be used to decide among the implementations when weighed against other measures such as real estate taken up on the chip and power consumed.

One powerful kind of design procedure is for editing layout structures. These procedures allow us to generalize logic cells and enhance our ability to achieve close packing and logical geometric layout of the cells.

A way to generalize a fixed geometry is for the designer to include within the cell's definition a class of layout hints. These hints may be specified either when a cell is initially defined, or later, when a more general version is necessary. Some of these hints might concern the options available for connecting this cell with other cells. For example, in the trivial ratioed inverter, the output node is available on any of the three mask levels, metal, poly, or diffusion.

A more useful hint is the concept of a seam. A seam indicates places in the layout that have flexibility for expansion and shows how the expansion is to be done. Seams are conceptual dividing lines in a stick cell layout along which the cell may be expanded and through which specified color stick wires may be routed. The seam specifies the manner in which cell is to be expanded to make room for the newly routed wires. Seam expansion may require cell modifications such as transitions of interfering signals from one layer to another, but in the most common case, merely involves knowledge of what parts of the cell geometry expand in which direction. For example, if a seam goes vertically through a diffusion, the diffusion may be expanded in the direction perpendicular to the seam. Each seam describes what materials can be run through without shorting to a feature of the cell or manufacturing a parasitic component.

Suppose that we had to run a signal up through each cell of the adder in the multiplier-accumulator. We would invoke a design procedure:

```
(for-each cell ((f ?n) (accumulator-adder multiplier-accumulator))
          (invoke Run-Through cell Vertical Poly))
```

This iterates the application of the Run-Through procedure over each part of the accumulator-adder whose name matches `(f ?n)`. These are the full-adders created by implementing it. Run-Through examines the vertical seams of the full-adder fragment, looking for one which can be used. If none can be found a failure message is produced which will inform the caller that he had better look for another way to accomplish his goal or that he should try to edit the full-adder cell to install a seam which can do the job. If there is an appropriate seam, the cell is stretched and the poly is run through. This changes the pitch of the cell and the interdependent objects are informed that they had better adjust to the new condition. Actually, here, things are pretty complicated. We cannot run polysilicon over a diffusion without creating an active transistor. Thus, the design procedure may only do this by changing the wire to a metal one, but this takes up lots of space so it can only be done if there is enough space at the desired pitch.

We can certainly anticipate that a moderately clever program could automatically generate seams within existing logic cells. The use of seams as "hints" to the design system is one example of how we intend to gradually develop a sophisticated design system. In the initial design system we avoid the requirement that very complex programs exist, are debugged, and are practical to use. These design system hints can gradually be augmented by sophisticated programs as they develop, but the success of the design system is not dependent on their development.

Our Initial Efforts

There are several reasons why we feel it is important to adopt an evolutionary approach to the development of the design system, starting with the implementation of a sophisticated interactive graphic design editor. First, it is important to have some capability to design integrated circuits in the early part of the research project. It is impossible to create the advanced design environment while working in a vacuum. The early design exercises will test the significance of our ideas and also allow us to develop more insight into the nature of integrated system design. Secondly, an evolutionary growth will enforce the principle that each major module must have a clean interface so that it can later be combined with more powerful system components.

Initially we will construct an interactive design editor. We will start from the example of ICARUS.[8] Our editor will handle structural models such as logic diagrams, mask layouts, and design descriptions in a text form. It will be capable of editing several design files simultaneously with a display organized as multiple windows. Both color and high resolution black and white screens will be used.

The editor commands will be interpreted as calls to primitives of the design procedure language which will be operating on various models in the design plan. The interactive component will have a clean interface to the design procedure system. The design plans are constrained to represent meaningful structures. Thus connectedness information from the logic schematic is used during mask editing to insure that the geometric operations don't inadvertently destroy the logical function of the circuit. Paths of diffusion or polysilicon will be stretched and re-layed out as pieces of the structure are moved. The correspondence between nodes in the logical structure and the geometric layout will be maintained automatically.

Incremental corrections that require insertion of new structures will be aided by automatic editing operations that move collections of objects while preserving layout design rules.

The layout model will consist of multiple representations, such as mask geometry, STICKS schematics[9], and logic schematics. These representations can be mixed on a single page, where the more abstract representations stand for what will eventually be mask geometry on the chip. A STICKS compiler transforms the non-metric representation into mask artwork that obeys the process layout rules. Simple design procedures will be included for regular structures such as PLA's and ROMs.

Some VLSI System Projects

We will develop several projects involving the design of particular VLSI chips. These projects cover a large range of difficulty, speculativeness and utility. Some of our projects are simple extensions or developments of existing technology which will give us some familiarity with the medium and some perspective with actual designs. We believe that it is useless to try to build tools to aid the engineering process, or to study the engineering process in the abstract, without some concrete projects to develop real engineering competence and taste.

For example, we have already developed a prototype LISP interpreter chip which has been through design and fabrication once[10]. We intend to use this chip and its successor -- a full sized interpreter and storage management system -- as a benchmark project to test some of our computer-aided design tools and methodologies as they emerge.

We also wish to design and fabricate a local-network interface chip. This is a similar "service project" chip which will help us improve our computer facilities and which will provide similar engineering exercises.

The MIT VLSI effort will build some considerably more complex systems using our computer-aided design technology. These projects will exercise our systems and methodology. Most of these projects have direct application to real world problems. We expect to support and learn from our interaction with these efforts.

We will also be interested in some specific chips for use in artificial intelligence research. One area that seems ripe for consideration is the problem of implementing processes that operate on very large stores of semantically related information, such as the semantic nets studied by Fahlman[11]. Fahlman was concerned with the fact that most problem solver programs have to labor over very simple deductions which seem instantaneous to a human. For example, if we learn that Clyde is an elephant, we can immediately answer questions such as whether Clyde is grey, or whether he can climb trees, as well as the answers to hundreds of other default facts about him. Fahlman worked out a scheme by which an important subset of these shallow but numerous deductions could be done very efficiently with specially constructed parallel hardware in the form of a network of simple, identical processing nodes with static interconnections. Fahlman's proposal is communication-intensive with almost no processing or memory at the individual nodes. All computations in a Fahlman net are done by "marker propagation". The nodes just have a few bits of memory which are used to store markers which are propagated in parallel along the static interconnections.

Implementation of marker propagation networks would be easy except for the enormous number of nodes required to construct a useful system. We estimate that a useful AI system requires at least 10^6 nodes. We do not yet know how to build the programmable connections, on the scale required by such a machine. Therefore, to investigate their properties these systems must be simulated, currently on general-purpose computers. Unfortunately the simulations are far too slow to be adequately tested, let alone be used as a support for other parts of a problem solving system.

We have some ideas about how such a system might be implemented and we expect that we will want to work on such an exotic architecture as part of our artificial intelligence research (in cooperation with Fahlman, who is now at CMU). One helpful constraint is that the computation is decomposible into essentially independent computational nodes such that each node's communication with other nodes is limited. When this is true, we may be able to configure a machine so that the computational nodes are allocated to segments of hardware with communications lines allocated to interconnect them. We will investigate a spectrum of such configurable architectures.

If the computational nodes and the communication channels to be established among them can be allocated at the outset, and if the set of nodes which must communicate with a given node is small, we may think of the computational problem as simulating a "wiring diagram". In fact, one interesting problem which breaks up in exactly this way is the simulation and analysis of systems which may be characterized by lumped-parameter models, such as electrical circuits.[12] In a more general setting, one can think of systems of algebraic constraints as networks which can be studied as if they were electrical circuits. A configurable architecture for such problems is constructed from a set of general-purpose processors, each of which is given the computational task of one component of the system. The architecture also requires a "patchboard" which programs the interconnectivity of the components for a problem. The patchboard may be a physical entity, such as a sorting network, or it may be virtual, such as a packet-switched network. One useful task for such a "circuit machine" is as a high-performance digital logic simulator, which can be used for experimenting with unusual computer architectures.

A class of architectures that we will investigate are network simulators. We do not really understand how to make completely parallel network machines, but there is an intermediate position. We imagine a hybrid between the conventional sequential architectures that we understand and the fully parallel architectures that have not yet been developed. With a machine of this type we can at least perform experiments on proposed parallel designs before they are constructed. A module in such a simulator consists of two parts -- several large memories defining node types, node states, and interconnections, along with a VLSI interpreter engine that makes a sequential pass performing a processing step on all nodes. With several such modules interconnected, networks of a million nodes can be simulated 2 or 3 orders of magnitude faster than can be done on purely sequential machines.

The performance advantage of the hybrid network simulator comes from several sources. First, the parallelism of the simulator modules provides a straightforward factor of 8, 16 or so. Second, a dedicated memory structure internal to the module provides several times the bandwith of the memory on a conventional machine. At each processing tick, the node's state and the state of its topological neighbors are fetched in a continuous stream of data pipelined into the interpreter engine. A network simulator in stream mode enjoys much the same advantage over conventional machines as vectorized arithmetic processors such as the Cray-1 enjoy over scalar processors. Third, unlike an instruction stream driven processor, each step of the simulator engine is interpreting an independent node. Thus pipelining and overlap can be freely used without the need of complicated interlock hardware. This freedom allows cascading several microcodable processing stages so that a multi-step node interpretation can be performed in one cycle.

Such network simulators are well suited to experimentation with proposed designs for parallel machines having large arrays of nearly uniform nodes. Some of these problem areas are digital logic simulation, marker propogation in semantic nets, and pattern matching for AI data base systems. However, in addition to being a research vehicle for parallel architectures, the hybrid sequential/parallel computer is a novel architectural paradigm that may have applications in many domains where the natural formulation of computation is object based as opposed to function based. We may find such possibilities in areas such as signal processing and discrete particle simulations.

Notes

1. This cognitive complexity barrier has been apparent for some time in the design of large software systems. The development of very high level languages is one approach to controlling this complexity. Software engineers have also developed methodologies such as "structured programming" [Dahl, Dijkstra & Hoare 1972] to help cope with the problem. Our Engineering Problem Solving project is an outgrowth of another approach concerned with the construction of intelligent design tools [Winograd 1973]. We are engaged in related research on the computer-aided design and analysis of analog electrical circuits [Sussman 1977a] and of software systems [Rich, Shrobe, Waters, Sussman, & Hewitt 1978].

2. Sussman [Sussman 1973] [Sussman 1977a] introduced a theory of problem solving, called Problem Solving by Debugging Almost-Right Plans, which is based on deliberately making simplifying assumptions which may introduce "bugs" into the solution. The resulting solution is then debugged until it is right. This theory was induced from observations of engineers and programmers in the process of design.

3. The distinctions between a "performance theory" and a "competence theory" for describing aspects of the behavior of humans was introduced by Chomsky [Chomsky 1965] in the context of natural linguistics. Loosely speaking, a competence theory concentrates on the factual issues of a domain whereas a performance theory is concerned with the issues of control and heuristics.

4. The power of a structured theory of design is demonstrated by Mead and Conway in their beautiful book [Mead & Conway 1979] on the design of VLSI systems. They have isolated a level of language which is natural for the design of interesting classes of NMOS chips. They speak in terms of "state machines", "programmed logic arrays", "bussed register arrays", "multiplexers" and other concepts which are primitives of a much higher level language than the AND, OR, NOT, JK flip-flop level of detail which most digital designers are used to. Using their ideas, students are able to design very complex VLSI systems with only a small amount of practice. Structured programming [Dahl, Dijkstra, & Hoare 1972] has had a similar but more controversial effect on the work of programmers.

5. The use of a special formalism for describing an electrical circuit from several points of view simultaneously, so that an automatic deductive system could make use of information deduced from each model was introduced by Sussman [Sussman 1977b]. Steele and Sussman [Steele & Sussman 1979a] have generalized

the notion to be useful for the description of other "almost hierarchical systems" which result from engineering design.

6. "Propagation of constraints" was originally invented as a generalization of "Guillemin's method" of analyzing electrical ladder circuits. It was used in the analysis programs EL [Sussman & Stallman 1975] and ARS [Stallman & Sussman 1976], and in the synthesis program SYN [de Kleer & Sussman 1978]. The basic idea of the method was first described in [Brown 1975] as part of a method for localizing faults in electrical circuits. De Kleer also used propagation analysis in his fault localizer [de Kleer 1976]. Sutherland [Sutherland 1963] appears to have developed a similar technique (the "One Pass Method") for constraint satisfaction in Sketchpad.

7. SIMULA [Dahl & Nygaard 1966] introduced the "class" as an abstraction mechanism in a programming language.

8. ICARUS is a minimal automated geometric draftsman developed at Xerox PARC by Fairbairn and Rowson [Fairbairn & Rowson 78].

9. STICKS is a semi-geometrical graphical representation of the layout of an integrated design. Features on various mask layers are represented by lines of appropriate color. STICKS diagrams show all topological information and approximate layout, but they suppress most metric information.

10. Our chip [Steele & Sussman 1979b] is an interpreter and storage manager for a dialect of LISP called SCHEME. It was part of the MIT project set for the Fall of 1978. Lynn Conway of PARC was teaching at MIT.

11. Fahlman's semantic memory scheme is described in [Fahlman 1977].

12. John Kassakian [Kassakian 1979] has a neat new approach to the simulation of complex electronic systems which he calls the "Parity Simulator". The basic idea is that he automatically configures a set of universal elements so that each simulates a device in a network and he configures the interconnection between them to be isomorphic to the interconnections in the netowrk being simulated. This turns out to be better for many applications than the traditional approach of simulating the behavior of the equations resulting from an analysis of the network.

References

[Brown 1975]
Brown, Allen L. Qualitative Knowledge, Causal Reasoning, and the Localization of Failures. Ph.D. thesis. MIT (September 1975). Also MIT AI Lab Technical Report 362 (Cambridge, March 1977).

[Chomsky 1965]
Noam Chomsky, Some Aspects of the Theory of Syntax, MIT Press, Cambridge, Mass. 1965

[Dahl, Dijkstra, & Hoare 1972]
O.J. Dahl, E. Dijkstra, and C.A.R. Hoare, Structured Programming, Academic Press 1972.

[Dahl & Nygaard 1966]
O.J. Dahl, and K. Nygaard, "SIMULA -- An ALGOL-based Simulation Language", Communications of the ACM, Vol. 9, No. 9, September 1966.

[de Kleer 1976]
De Kleer, Johan. Local Methods for Localization of Faults in Electronic Circuits. MIT AI Lab Memo 394 (Cambridge, November 1976).

[de Kleer & Sussman 1978] De Kleer, Johan, and Sussman, Gerald Jay. Propagation of Constraints Applied to Circuit Synthesis. MIT AI Lab Memo 485 (Cambridge, September 1978).

[Fahlman 1977]
Scott E. Fahlman, NETL: A System for Representint and Using Real-World Knowledge, PhD Thesis, MIT Department of Electrical Engineering and Computer Science, June 1977; in the MIT Press series in Artificial Intelligence, 1979.

[Fairbairn & Rowson 1978]
"ICARUS: An Interactive Integrated Circuit Layout Program", Proceedings of the 15th Annual IEEE Design Automation Conference, June 1978.

[Kassakian 1979]
John G. Kassakian, "Simulating Power Electronic Systems -- a New Approach", to appear in Proceedings of the IEEE, 1979.

[Mead & Conway 1979]
Carver A. Mead and Lynn A. Conway, Introduction to VLSI Systems, Addison Wesley, 1979.

[Rich, Shrobe, Waters, Sussman & Hewitt]
C. Rich, H.E. Shrobe, R.C. Waters, G.J. Sussman, and C.E. Hewitt, Programming Viewed as an Engineering Activity, MIT Artificial Intelligence Laborarory Memo 459, January 1978.

[Stallman & Sussman 1976]
Stallman, Richard M., and Sussman, Gerald Jay. "Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis." Artificial Intelligence 9 (1977), 135-196; also MIT Artificial Intelligence Laboratory Memo 380, September 1976.

[Steele & Sussman 1979a]
Guy Lewis Steele, Jr. and Gerald Jay Sussman, "Constraints", Proceedings of the STAPL\sigplan Conference on APL, Rochester, New York, June 1979; also MIT Artificial Intelligence Laboratory Memo 502, November 1978.

[Steele & Sussman 1979b]
Guy Lewis Steele, Jr. and Gerald Jay Sussman, "Design of LISP-Based Processors", MIT Artificial Intelligence Laboratory Memo 514, March 1979.

[Sussman 1973]
Gerald Jay Sussman, A Computer Model of Skill Acquisition, PhD Thesis, MIT Department of Mathematics, August 1973; American Elsevier Artificial Intelligence Series, New York 1975; also MIT Artificial Intelligence Laboratory Technical Report 297, August 1973.

[Sussman 1977a]
Gerald Jay Sussman, "Electrical Design: A Problem for Artificial Intelligence Research", Proceedings of the Fifth International Joint Conference on Artificial Intelligence, Cambridge, Massachusetts, August 1977.

[Sussman 1977b]
Gerald Jay Sussman, "SLICES: At the Boundary Between Analysis and Synthesis", Proceedings of the IFIP Working Conference on Artificial Intelligence and Pattern Recognition in Computer-Aided Design, Grenoble 1978; also MIT Artificial Intelligence Laboratory Memo 433, July 1977.

[Sussman & Stallman 1975]
Sussman, Gerald Jay, and Stallman, Richard M. "Heuristic Techniques in Computer-Aided Circuit Analysis." IEEE Transactions on Circuits and Systems, vol. CAS-22 (11) (November 1975).

[Sutherland 1963]
Sutherland, Ivan E. SKETCHPAD: A Man-Machine Graphical Communication System. MIT Lincoln Laboratory Technical Report 296 (January 1963).

[Williams 1977]
J.D. Williams, "Sticks -- a New Approach to LSI Design", M.S.E.E. Thesis, Dept. of Electrical Engineering, MIT, June 1977.

[Winograd 1973]
Terry Winograd, "Breaking the Complexity Barrier, Again", Proceedings of the ACM SIGIR-SIGPLAN Interface Meeting, Nov. 1973.