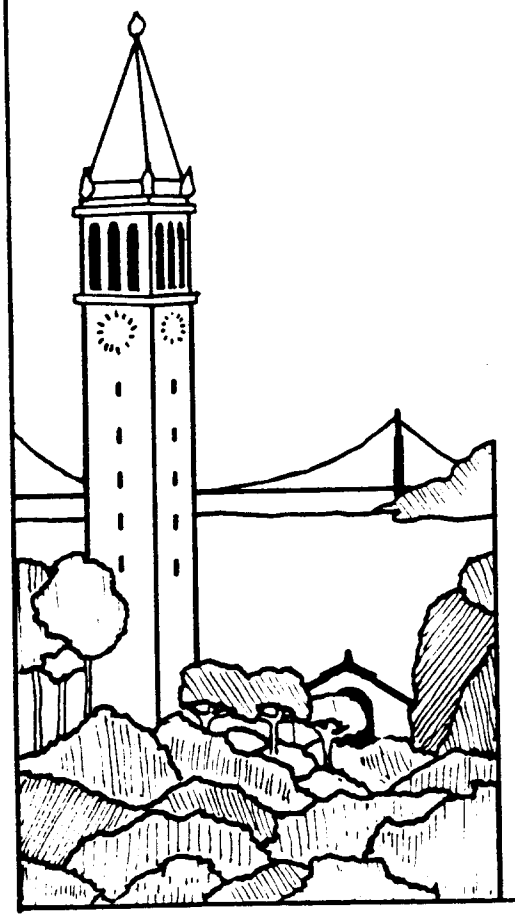
Transistor Sizing

Jonathan Pincus



Report No. UCB/CSD 86/285

February 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

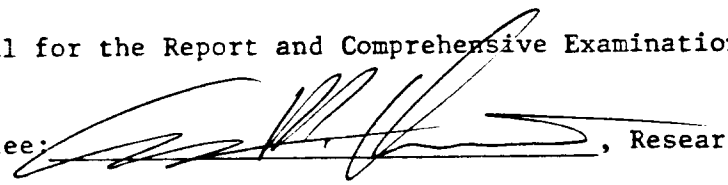
Author Jonathan Pincus

Title Transistor Sizing

RESEARCH PROJECT

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, in partial satisfaction of the requirements for the degree of Master of Science, Plan II.

Approval for the Report and Comprehensive Examination:

Committee:  , Research Adviser

3 December 1985 Date

Randy H. Katz

December 6, 1985 Date

Transistor Sizing

Jonathan Pincus

ABSTRACT

Several methods of choosing appropriate sizes for transistors in a VLSI schematic to meet a specified delay criteria are considered. Simulated annealing and heuristic techniques are investigated. MOST is a Prolog program which makes use of information provided by the PTA timing analyzer to implement these various approaches. Both MOST and PTA are written entirely in (interpreted) Prolog; nonetheless, performance gains of over 50% as compared to an unsized circuit can be realized in a few minutes of CPU time. Using a simple RC timing model, heuristics are found to be more efficient than simulated annealing.

Part of this research was sponsored by Defense Advance Research Projects Agency (DoD) Arpa Order No. 4871 Monitored by Naval Electronics Systems under Contract No. N00039-84-C-0089

Chapter 1
Introduction

The designer of a VLSI circuit must consider not only functional correctness but timing behavior. Usually, there is some specification of how quickly the circuit must produce its output. Once a schematic, transistor-level description of the circuit is produced, it must be forced to meet the *delay constraint*. This is done by assigning sizes to the transistors.

Note that this is a different problem than the ratioing of transistor sizes necessary in nMOS and some CMOS methodologies. Those considerations involve waveform shape and may affect the circuit's correctness; this paper only deals with the speed of the circuit.

Increasing the size of transistors in a VLSI circuit tends to decrease the delay through the circuit, but at the cost of increasing its area. While transistor area is usually only a small component of total chip area, that is only because transistor sizes are usually "reasonable." Minimizing delay can result in huge transistors; beyond a certain point, however, larger transistors actually increase delay.

Actual minimization of the circuit's delay is usually not required. Instead, the delay must be reduced to meet the specified constraint. Given a delay model, some expression for maximum delay through the circuit can be derived. It is thus possible to view the problem as one of constrained minimization:

1) minimize: total transistor area

   subject to: actual delay $<$ delay constraint

Truly minimizing transistor area is not vital, however; in fact, any "reasonable" solution which reduces the delay below the constraint will be acceptable. Thus the problem can also be cast as

2) minimize: excess delay above constraint

subject to: reasonable total transistor area.

Note that only excess delay is being minimized; no reward is given for reducing delay below the constraint.

Standard non-linear optimization techniques are not well suited to these problems. In problem 1, the objective function is quite simple, but the constraint is both highly non-linear and expensive to compute — even finding a feasible solution is very difficult. In problem 2, it is the objective function which is extremely complex and difficult to deal with. The major difficulty is that circuit delay is the maximum path delay, and there are a combinatoric number of paths through the circuit; furthermore, path delay itself is an extremely complex function. Previous work, frequently involving simplified delay models, is covered in Chapter 2.

Human designers avoid considering all these paths by using intuition and heuristics. After some initial configuration is chosen, simulations and timing analyses are run on the circuit to find its critical paths — the paths through the circuit whose delay exceeds the constraints — and the designer reduces their delay sufficiently. Now some other paths may be critical, so the process iterates until the maximum delay through the circuit is satisfactory. No formal attention is paid to transistor area; presumably, by only dealing with critical paths, unimportant transistors will be left at minimum size. Such *critical-path heuristics* are one of the subjects of Chapter 3; simpler heuristics, involving modifying the sizes of individual transistors, are also dealt with.

In a large circuit, however, there may be many paths each requiring more time than permissible; if an iteration of the critical-path heuristic is required for each such path, the total computation required may be immense. One solution is to consider more than one critical path at once; this unfortunately leads to

extremely complicated decisions, involving simultaneous minimization of several equations. Another approach is to work with the entire circuit at once by using a probablistic hill-climbing technique — such as *simulated annealing* — in the hope that the cost function can be chosen so that the process will reduce the delay on many paths simultaneously. This alternate tactic is considered in Chapter 4.

MOST (Method for Ordering and Sizing Transistors) is a Prolog program which makes use of the information supplied by PTA (the Prolog Timing Analyzer) in conjunction with either heuristics or a simulated annealing algorithm to assign sizes to the transistors. In contrast to most previous work, it sizes transistors without guidance from the designer; there is no need to specify which paths to examine, for example. The two programs together are approximately 1500 lines long, or 350 clauses; the source code is included as an appendix. MOST's design and implementation are described in Chapter 5.

The current version of PTA uses a simple lumped RC delay model to find the maximum delay at and critical path to each signal within a circuit. An interesting feature of PTA is its ability to provide symbolic equations for the resistance and capacitance (and hence delay) at each node. The details of PTA and its implementation are presented in Chapter 6.

Throughout the paper, fragments of Prolog code are included to illustrate some of the algorithms being described. This code is almost invariably an oversimplification, but much clearer than the complete implementation. In particular, questions of efficiency — either of storage or computation — are ignored.

Detailed results for the various approaches are presented in chapter 7, but Table 0 provides a short summary. All CPU times throughout the paper are for a VAX 785 running interpreted Cprolog.

Terminology

The usual statement of a problem is "Assign sizes to the components of circuit X so that all its outputs are produced by time T." T is called the *maximum delay* or *constraint*, and the entire process is called *sizing* the circuit. The *actual delay* through the circuit is the delay given a particular assignment of sizes to its components, while the *size* or *area* of a circuit is the sum of the component areas.

A circuit is described hierarchically in terms of *cells*; the particular data structure used is called a *constrained hierarchical schematic*, and so the terms *cell* and *CHS* are used interchangeably. A cell is either a *primitive* cell, or it is made up of *subcells*. Transistors are usually regarded as being the primitive elements, but most techniques apply equally well if logic gates or even macro cells are taken as primitives.

Since the components of the circuit may be cells, rather than transistors, sizing a circuit may involve sizing the cells. This is a rather unfortunate choice of phrasing: sizing a cell does not mean deriving its maximum bounding box, but rather assigning sizes to the primitives in its substructure.

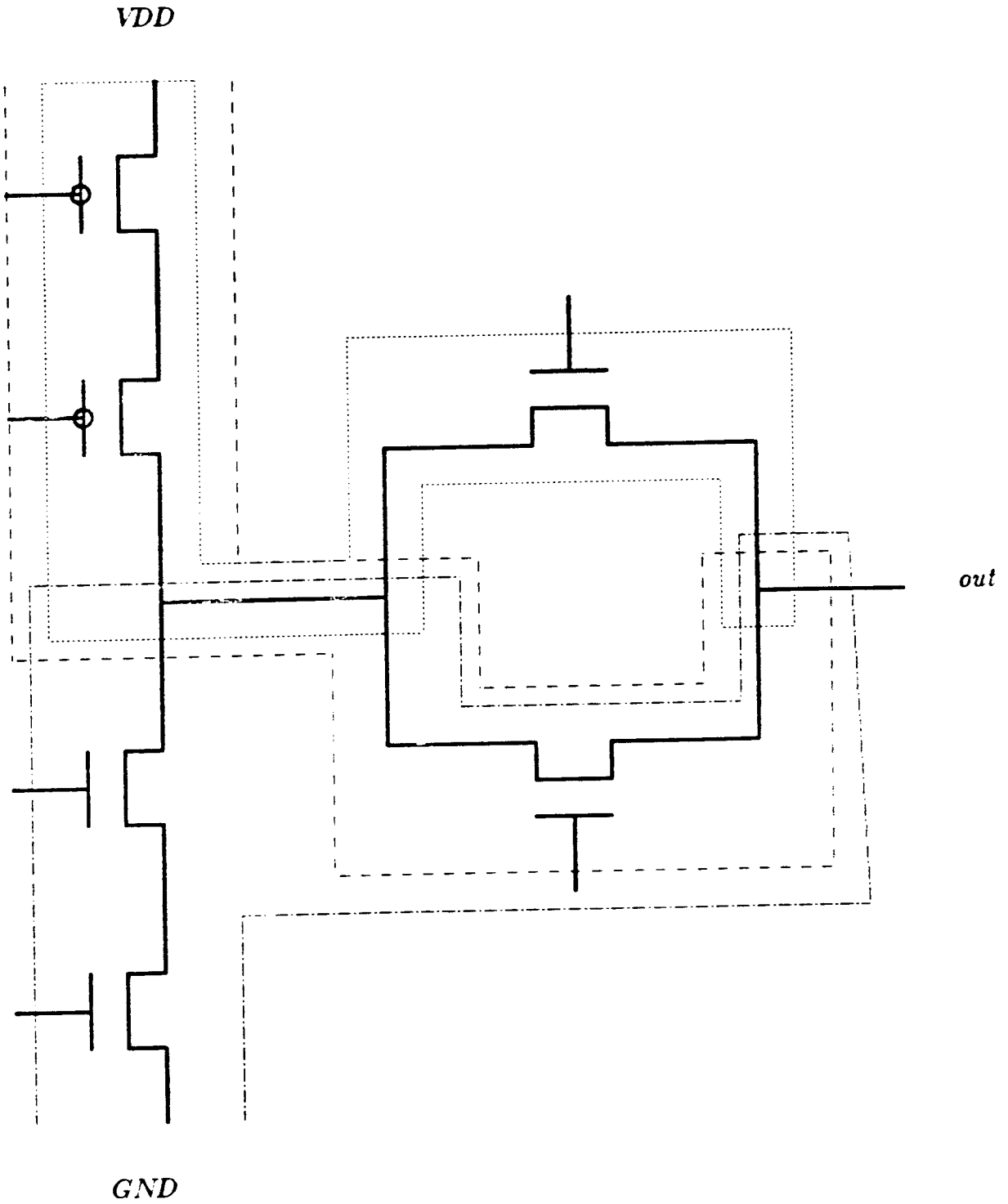
A related problem is that of *minimizing delay* through the circuit. In this case, there is no explicit delay constraint; the goal is to make the circuit run as fast as possible. Additionally, there may be an *area constraint*, or some maximum permissible transistor area.

Computing the delay through the circuit is the job of the timing analyzer. Essentially, what needs to be done is find each *path* by which an input to the circuit can affect an output, and then take the maximum of all these *path delays*. The path with the longest delay is referred to as the *critical path*.

A path will be made up of several *stages*. A stage (the term is borrowed from Ousterhout [14]) is a chain of transistors from a driving source (usually an

input to the chip) to a use of the signal — either as an output of the chip, or as a gate to another transistor. A stage usually corresponds to a path through a logic gate and its associated pass transistors.

All three of the boxed areas in the diagram below are stages.


VDD
out
GND

Chapter 2
Path Sizing and Previous Work

A more restricted form of the problem only considers a single path instead of the entire circuit. Techniques for *path-sizing* do not generalize well, for two related reasons: the larger size of an entire circuit, and the additional complexity caused by the possibility of multiple paths through the circuit. Much previous work has been done on this problem, however, and it makes a good introduction to the more general case. Moreover, it can be an useful component of a general solution, especially in conjunction with critical-path heuristics.

Path Sizing

In most previous work, the path is viewed as being made up of logic gates, rather than individual transistors. Furthermore, most authors assume that signals generated by gates in the path are not used anywhere but their successor, and that inputs similarly do not come from outside the path. These very restrictive assumptions allow a variety of approaches — summarized nicely by Matson [10] — to be successful.

Comparing these results is difficult, largely because of the paucity of statistics provided by the various authors. Table 1 at the end of this section summarizes as well as possible, leaving question marks for figures not provided.

The most obvious approach is simply to use an already-existing general purpose optimization package along with a highly accurate timing analyzer such as SPICE. This turns out to be impractical: too much time is spent in simulation. A particular problem is that symbolic derivatives are not available, and so must be computed numerically at great expense. Matson gave results for using the DELIGHT package along with SPICE in [10], but only as a contrast to the efficiency of his work.

One way of avoiding the high computational cost of such an approach is to use a simplified model for transistors. At some cost in accuracy, this saves greatly in computation, especially if symbolic derivative information can be calculated.

When using the simple RC model, it is possible to derive the equations for delay in terms of the transistor sizes, and then solve these by a quasi-Newton method. Consider the critical path as made up of n stages, each of which in turn drives the next stage, and let D_i be the delay of stage i. Then

$$D_{tot}=\sum_i D_i$$

and, due to the lumped RC model

$$D_i=R_i*C_i \qquad (*)$$

Now let T be all the transistors making up stage i; if R_{other} and C_{other} capture the interconnect and output resistances and capacitances, then

$$R_i=\sum_{t\epsilon T} R_t+R_{other}$$
$$C_i=\sum_{t\epsilon T} C_t+C_{other}$$

The resistance of a transistor is inversely proportional to its size S_t; the capacitance directly. Using this fact, equation (*) can be rewritten as

$$D_i=(\sum_{t\epsilon T}\frac{k_1}{S_t}+R_{other})*(\sum_{t\epsilon T} k_2*S_t+C_{other})$$

All the D_i's can be summed to give an equation for the total delay, and this equation can be minimized or set to a particular value. This task is greatly simplified by the ease of computing the partial derivatives.

One fact not immediately obvious in the above description is that the C_{other} of one stage may involve the sizes of transistors in the next stage. One component of C_{other} is the load capacitance, which includes the gate capacitance of whatever gate in the next stage being driven by the current stage. This means that the problem is not truly separable: stages cannot be treated independently.

It is not clear how easy it is to perform optimization even given the simplified model: it depends on how many variables (or transistors) are involved in the equations. Individual stages are likely to be short — Ousterhout [14] claims that most are only two or three transistors in length — but the critical path may consist of a large number of stages. In a 32-bit processor, for example, the critical path is likely to be the carry chain through the ALU, which will have at least 32 stages.

Several authors use variations on this approach. Glasser and Hoyte [3] model the delay on a path as the sum of the gate delays. This model ignores the shape of the input waveforms, but Glasser and Hoyte argue that its estimates are accurate within 30%. Each gate is modeled as a capacitor and a resistor, and their program minimizes the equation for maximum delay using relaxation techniques in order to find the proper "scale factor" for each gate.

Hedlund's **EO** [5] (for Electrical Optimizer) can either minimize delay or minimize power consumption with bounds on delay. It deals with several paths simultaneously, as well as both polarities of input on a single path, by minimizing (over the set of assignments to transistors) the maximum (over the paths and polarities) delay. In other words, if $D_P(S)$ is the delay for assignment S on path P, EO computes

$$\min(\max_P D_P(S))$$

The maximum is approximated by the continuous function smax (for "smoothed maximum"; see Ruehli *et. al.* [19]), where

$$\mathrm{smax}(x_1, \cdots, x_n) = \frac{1}{\lambda}\ln(e^{\lambda x_1} + \cdots + e^{\lambda x_n})$$

The minimization is done by a quasi-newton non-linear optimization method.

Another way of potentially lessening computation is to use heuristics instead of non-linear optimization techniques. Since the problem is rather structured, and

the optimal solution (in this case, the absolutely minimum transistor size) is not required, it may be possible to capitalize on this structure via heuristics as a human designer does. Note these are heuristics for sizing a single path, as distinct from critical-path heuristics for sizing the entire circuit.

Kao, Fathi, and Lee [7] use an extremely simple heuristic: at each step, the gate contributing the most to delay relative to its current area is increased in size. This "scapegoat heuristic" makes no attempt to capture the complex interactions of all the gates within the path, but they claim that performance is satisfactory even for relatively large circuits.

Trimberger's **Andy** [21] uses the "ramped-driver" heuristic. Each gate is divided into several *stages* which increase in size by a fixed *fan-out factor* in order to drive the (presumably) large capacitive load at the output of the gate. For each gate, the capacitive load is computed, and then an equation for the proper number of stages for the gate is solved. The capacitances are computed starting at the end of the circuit, and then the program works backwards, sizing each gate as it goes (this is done because the output capacitance on gate i depends on the input capacitance of gate i+1).

Trimberger claims that the ramped-driver heuristic, although it does not minimize delay, is desirable because in general it requires less power and smaller area. Additionally, he says, it is closer to how human designers attack the problem.

Lee and Soukup [9] take a similar approach, first solving for the optimal number of stages, then optimizing the stage sizes (as opposed to Trimberger's fixed fan-out factor). They also discuss the minimization of area: given a constraint on the delay, they use Lagrange multipliers to solve the optimization problem. However, they quote no statistics on the efficiency of their program.

Matson [10] argues that heuristics are in general less efficient than non-linear optimization methods, particularly when the delay constraint is near the minimum delay achievable by the circuit. Additionally, he claims, the accuracy of the timing models is insufficient for high-performance VLSI design. On the other hand, a general non-linear optimizer fails to take advantage of the structure of the problem. As a result, he uses a special-purpose optimizer in conjunction with a timing model [11] of intermediate complexity.

The particular optimization problem Matson attacks is minimizing power subject to a constraint on maximum delay, but the technique applies equally well to minimizing transistor area. In both cases, the objective function is *separable*: it is the sum of contributions from each of the individual components (either macrocells, gates, or transistors) along the path. If the delay is also regarded as the sum of individual contributions then it too is separable. This is not strictly true, due to the effects of waveform shape and the interactions between input and output capacitances, but is a useful assumption.

Using the method of *duality*, a variation on Lagrange multipliers, Matson takes advantage of the near-separability of area and delay by dividing the minimization into a minimization of each cell in succession. Instead of one minimization over a very large vector space, many minimizations over small vector spaces are performed instead, and since the cost of non-linear minimization grows combinatorically, the divide-and-conquer method greatly speeds up the process.

Table 1					
Path sizing: comparison of previous work					
Algorithm	Circuit	Size	Reduction	Machine	CPU time
DELIGHT/SPICE [10]	Inverter Chain	6	?	VAX 11/750	3151.7
Relaxation [3]	Inverter Chain	500	?	DEC 20/60	50
Quasi-Newton [5]	Control Logic	9 20	63% 52%	VAX 11/750	0.1 2.3
Scapegoat [7]	1-bit adder	13	?	XEROX 1108	20
Ramped Driver [21]	PLA	?	40%	DEC 20/60	?
Duality [10]	Inverter Chain 4-bit adder	6 76	? ?	DEC 20/60	16.3 522.3

Generalization

Most of these techniques consider only a chain of gates, rather than a path at the transistor level, and ignore the possibility of outside influences. The presence of pass transistors complicates the issue. It is not sufficient to size the transistor whose output is the gate of the the next transistor in line; other transistors connected to that transistor may need to be sized as well. Heuristics will have more difficulty in this situation, especially since a single transistor may be connected to several different transistors in the critical path.

Most authors also gloss over the question of minimum widths of transistors. Heuristics which only increase the size of transistors cause no difficulty here, of course, but all non-linear minimizations must actually be constrained minimizations. Strictly speaking, the transistor widths must also be integers (or integer multiples of some fixed lambda); most techniques simply round off in a post-processing phase to deal with this difficulty.

With such modifications, these approaches can be used for path sizing in critical-path heuristics. The ramped-driver heuristic does not really apply to this case, however, since it is not desirable to add new stages.

An important question is how well these techniques generalize to the case of

an entire circuit. The next chapter considers expanding the "scapegoat heuristic" to the entire circuit. Matson's duality technique, however, does not generalize as nicely. The divide-and-conquer nature of the technique would make it particularly appropriate for large circuits, so it is worth examining just how it breaks down.

The key to being able to divide the optimization problem is to be able to separate the total path delay into individual cell delays. Circuit delay can indeed be broken down into the individual cells on the critical path, but as transistors are sized, the critical path changes. In other words, the decomposition is different at different times in the process. Matson only deals with specific paths, which remain the same throughout the analysis.

Fishburn and Dunlop have recently published some impressive work. They have shown that, given an RC delay model, circuit delay is a *convex function* of transistor sizes (so far, they have been unable to generalize their result to slope delay models). The pertinent feature of convex functions is that any local minimum is in fact a global minimum. This in turn implies that simulated annealing or multiple initial configurations are not required to avoid local minima.

The approach used by their **TILOS** program, however, is in fact a slight variation on a scapegoat heuristic described below. The primary value of their result appears to be in the confirmation that heuristics are in general "good," rather than providing any algorithmic method for solving the problem.

Simulated annealing is still a potentially viable technique. As will be discussed in chapter 4, the cost function is not necessarily the maximum delay through the circuit. A more complex cost function — for example, the smoothed maximum of all the path delays — may not have the convex property, but may still be a more accurate measure of how close to a solution the configuration is. In addition, simulated annealing techniques will still apply in the case of non-convex

delay functions, which may arise from a more accurate delay model.

Chapter 3

Heuristic approaches

In the absence of an algorithmic solution, it is natural to search for viable heuristics — all the more so since this is how human designers currently attack the problem. The usual course of such heuristics is to perform an analysis of the circuit, giving such information as the maximum delay through the circuit and the critical path. and then use this information to guide the next resizing step.

```
size(Circuit,Constraint) :-
     analyze(Circuit,Delay,Info),
     resize_if_necessary(Circuit,Delay,Constraint,Info).

resize_if_necessary(Circuit,Delay,Constraint,Info) :-
     Delay <= Constraint.  % done!

resize_if_necessary(Circuit,Delay,Constraint,Info) :-
     Delay > Constraint,
     apply_heuristic(...),
     size(Circuit,Constraint).
```

Two criteria can be applied to heuristics: efficiency and optimality. Optimality is simply a measure of how close to the optimum performance circuit the heuristic can come. Although the absolutely optimum circuit is not required, if a heuristic can not even approach it with consistency, it is not particularly useful.

Assuming a heuristic does give reasonable results, efficiency measures how quickly it does. Some heuristics may be extremely efficient for reducing delay up to, say, 40%, but extremely inefficient beyond that. A major factor in efficiency is the number of timing analyses required, so many of the different approaches are attempts to reduce this number. Potentially, however, the amount of work done to avoid reanalysis may actually become the dominant factor.

Heuristics fall into two major categories: transistor-level and critical-path. Transistor-level heuristics work with one transistor at a time. A timing analysis is performed, and then one transistor is resized. The advantage of this scheme is its

simplicity; in general, no complicated equations need to be solved, and interactions between critical paths do not need to be considered. The corresponding disadvantage is the lack of efficiency, particularly if a full timing analysis needs to be performed after sizing each transistor. Compromise schemes involving sizing several transistors before performing another analysis avoid this problem, but only at the cost of increasing complexity.

Critical-path heuristics mirror human designers' strategies. The critical path through the circuit is found, and then sized so that it meets the delay constraint. The process is repeated until all path delays have been reduced sufficiently. This reduces the number of analyses of the circuit, but problems now arise due to the potential interaction of critical paths: if a transistor is on two different paths, what should its size be?

To reduce the number of analyses even further, more than one critical path can be sized before re-analyzing. Either all the paths are considered simultaneously, or some form of iteration is performed without reanalysis. This approach compounds the difficulties of interaction, and potentially increases the complexity of the path-sizing problem. On the other hand, it can drastically reduce the number of analyses, particularly when many paths are critical simultaneously.

Mixed approaches, combining the above heuristics, are also possible. For example, a critical-path heuristic may be combined with a transistor-level heuristic for transistors not on the critical path. Alternatively, one heuristic may follow another: a heuristic considering multiple critical-paths may be used initially, and once the number of critical paths is reduced sufficiently, a standard critical-path heuristic may take over. Currently, very little work has been done in this promising area.

Transistor-level Heuristics

The two important questions at this level are what transistor to resize and how to do the resizing. There are several ways to choose what transistor (or transistors) to resize.

Most obvious is random choice. This is extremely simple, and fast to compute, but results in an unacceptably large number of timing analyses.

As simple generalization of the "scapegoat heuristic" in which the transistor contributing the most delay is resized is quite simple to implement.

```
apply_heuristic(Transistors,Equations) :-
     choose_scapegoat(Sizes,Equations,Culprit),
     adjust_size(Culprit).
```

Once again, this computation is not too expensive; on the other hand, it is difficult to say just what "contributing the most delay" means. In the final configuration, some transistors will still contribute more delay than others, and eventually the point of diminishing returns is reached for an individual transistor: even though it contributes much of the delay, it is better to leave it that way and resize another transistor instead.

One way of avoiding this problem is by instead examining the change in delay resulting from resizing each transistor. In essence, this approach works with the derivatives rather than the delay function itself. The derivatives can be computed symbolically, or some numeric approximation may be used instead. A simple and useful approximation to the derivative is the change in cost given a unit change in the size of the transistor.

Of course, there is no need to restrict these techniques to a single transistor. More than one transistor can be resized in each pass. The above approaches can all be generalized in very straightforward ways to consider multiple transistors.

This leads to methods in which every transistor contributing more than a specified amount of delay is increased in size, or where every transistor whose resizing would decrease delay is resized.

Dealing with multiple transistors simultaneously is almost always advantageous. Very little additional work needs to be done, since the effect of each change needs to be computed in order to find the best change, and the number of analyses is almost always reduced. The one way in which this decision can be harmful is if the interrelationships between transistors are too great, and resizing one transistor affects the decisions about whether to resize others. This can lead to oscillation.

There are several ways to decide how much to adjust the size of the transistors. Simplest, and surprisingly efficient, is simply increasing the size of the transistor by one unit. The fact that other transistor sizes will continue to change reduces the advantage of more complicated schemes, such as solving for the optimal size given the current size assignments to other transistors.

Critical-path Heuristics

The basic idea of finding the critical path and then resizing it is simple enough to implement.

```
apply_heuristic(Circuit,Crit_path,Constraint,Delay) :-
    resize(Critical_path,Constraint,Delay).
```

One question is how to do the path sizing. Three of the approaches discussed in chapter 2 are worth considering:

1) Individual transistor heuristic, in which a single transistor at a time is increased in size.

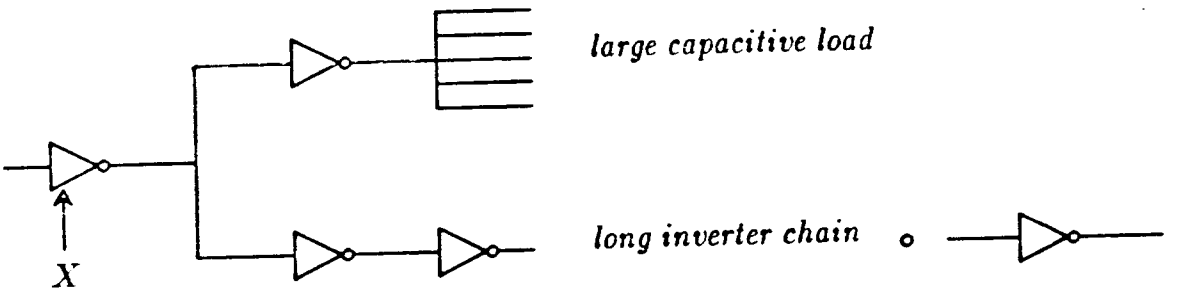
2) Numeric solution of the path delay equations for optimal sizes.

3) A simplified numeric solution, given some assumptions about the eventual sizes.

Each of these methods is more complicated and time-consuming than the ways of sizing individual transistors discussed above. From an efficiency standpoint, then, the question is whether the additional computation done here is compensated by a corresponding decrease in the number of timing analyses required.

The real difficulty in critical-path heuristics is the interaction between different paths. What is to be done if a transistor has had a size assigned to it in the course of sizing a path, and then is a component of another critical path considered later.

There are two possible strategies for dealing with this situation. One is to allow each transistor to be sized only once; once it has a size assigned to it, it is fixed. This is quite simple to implement, but may not be sufficient. Consider the following circuit:



In the case of a chain of identical dates, the ramped-driver heuristic provides optimal solutions. The proper size assignment, though, depends on the length of the path. Assume both paths exceed the delay constraint, and further assume that the lower (longer) path is the most critical. When it is sized, inverter X will have some relatively small size assigned to it. Now the upper path is still critical; to reduce its delay, inverter X will have to be increased in size.

The other method allows transistors to be resized as many times as necessary. However, resizing a transistor affects the delays along paths which have previously been sized, potentially requiring once again sizing those paths. This in turn creates the possibility of a loop between two paths, where sizing one undoes the effect of sizing the other. In terms of the above example, when the size of transistor X is increased to reduce the delay on the upper path, the lower path's delay will be increased.

It might seem that Prolog's backtracking mechanism offers an elegant solution to this problem: assign a set of sizes to a path, and if no global assignment can be reached satisfying the delay constraints, simply backtrack and size the path differently. This is undeniably theoretically possible, but in practice extremely inefficient. Backtracking throws away all the information gained, and so there is no way to know what caused the failure or how next to size the path. Although assert could be used to keep the information, so much would need to be asserted that the Cprolog interpreter, at any rate, would not allow it.

It is possible to set up relationships among transistor sizes along a path so that resizing one of the transistors implicitly causes the resizing of all the others. Techniques from the AI community such as access demons might be used for this problem: whenever one transistor's size is modified, the demon could change other sizes as necessary. Once again, however, there is the potential for loops in which two transistors' sizes mutually depend on each other. This would seem to require a rather general symbolic mathematics package for solving simultaneous non-linear equations at each step.

Multiple critical paths can be considered simultaneously. As more and more paths are considered, the improvements made at each stage are potentially greater, but the amount of computation that needs to be done also increases.

Chapter 4
Simulated Annealing

The basic algorithm

Simulated annealing [8] is a probablistic hill-climbing algorithm. It differs from standard hill-climbing in that a new configuration may be accepted even if it increases the cost; the chance of this occurring is controlled by a parameter called the temperature, which steadily decreases. This prevents getting trapped in a local minimum due to an unfortunate choice of initial configuration.

The algorithm divides into an outer loop, which gradually decreases the temperature, and an inner loop, which performs a number of iterations at each temperature. At each iteration, a new configuration is generated, its cost is evaluated, and then the acceptance function determines whether or not the configuration is accepted. Any configuration decreasing the cost will of course be accepted; different acceptance functions give different chances of accepting configurations which increase the cost. The usual acceptance function, used throughout this paper, is

$$\text{acceptance chance} = \exp(-\frac{\Delta\text{cost}}{T})$$

The algorithm can terminate in two ways: it succeeds if delay is reduced below the desired goal (although this success may be postponed briefly in order to minimize the sizes of the transistors), and it fails if some failure criterion is met. A standard failure criterion is no change in the configuration after a certain number of times through the main loop.

```
anneal(Circuit,Constraint) :-
     initialize(Configuration,Delay,Temperature),
     outer_loop(Configuration,Constraint,Temperature).

outer_loop(Configuration,Delay,Constraint,T) :-
     Delay =< Constraint.    % success
outer_loop(Config,Delay,Constraint,T) :-
     Delay > Constraint,
     iterations_at_temperature(T,N),
     inner_loop(N,Config,Cost,T,New_config,New_delay),
     update_temperature(T,New_t),
     outer_loop(New_config,New_delay,Constraint,New_t).

inner_loop(0,...).
inner_loop(N,Config,Cost,T,New_config,New_delay) :-
     generate(Config,Test_config),
     cost(Test_config,Test_cost),
     accept(Test_cost,Cost,T),
     N1 is N-1,
     inner_loop(N1,Test_config,Test_cost,T,
               New_config,New_delay).
inner_loop(N,Config,Cost,T,New_config,New_delay) :-
     N1 is N - 1,
     inner_loop(N1,Config,Cost,T,New_config,New_delay).
```

In terms of the particular problem being attacked, a configuration is simply an assignment of sizes to the transistors. New configurations are generated by random perturbations of each size; by restricting these perturbations to be integers, we assure that the final transistor size is also integral. The cost of a configuration includes a penalty for exceeding the specified delay, and another term relating to the total size of the transistors (in order to keep the circuit from growing too large).

Computing the cost involves computing the actual delay through the circuit. The PTA timing analyzer takes a given configuration and finds the delays of all the nodes. A large amount of additional information is supplied as well: the transistor causing the maximum delay for each node, allowing critical paths to be recreated, and symbolic equations for the resistance and capacitance of each transistor. PTA uses a depth-first search algorithm, ensuring that nodes will not

be reprocessed, and a simple RC model for simplicity and speed. Despite this, it consumes the bulk of the program's time; for a 100-transistor circuit, for example, timing analysis requires over 10 cpu seconds.

Many parameters can be varied in an attempt to improve performance. Among these are the initial temperature and configuration, the rate at which the temperature decreases, the proper number of iterations at a given temperature, the acceptance chance, and the generation procedure. Much theoretical research has been done in this area, but so far none of these results have been incorporated into this work.

Cost functions

A major determining factor in the performance of the algorithm is the cost function. Several different functions have been tried, all revolving around the idea of charging a penalty for a delay exceeding the constraint. If the desire were simply to reduce circuit delay to a minimum, then the penalty could just be the delay; since the problem is only to meet a specified criteria, though, no bonus is given for reducing delay below this bound.

$$\text{Penalty} = \max(\text{Delay}-\text{Constraint},0)$$

Initially, this was the entire cost function. Since it ignored transistor sizes, it led to very large circuits.

The first cost function still weighted the maximum delay through the circuit much more heavily than the total size:

$$\text{Cost} = 10*\text{Penalty}+\text{TotalSize} \qquad (1$$

The process essentially divides into two steps: first the sizes of the transistors increased as delay is reduced to the constraint, and then the total size component of the cost takes over, and the sizes are gradually reduced. A satisfactory solution is usually reached, but rather slowly, since essentially only one critical path is being considered.

Since one of the justifications for using simulated annealing was the possibility of dealing with multiple paths simultaneously, the next improvement was to consider all critical paths in the cost function.

$$\text{Cost} = 5*\text{Penalty}_{max} + 5* \sum_{i \text{nodes}} \text{Penalty}_i + 2*\text{TotalSize} \quad (2$$

The most critical path is weighted more heavily than others, since it is still the primary limitation on the circuit.

The third cost function weights the sizes more heavily.

$$\text{Cost} = 2*\text{Penalty}_{max} + \sum_{i \text{nodes}} \text{Penalty}_i + \text{Total Size} \quad (3$$

For each of these three cost functions, maximum permutation sizes of 1, 2, and 4 were tried in turn. Results are summarized in Table 2.

Table 2

Cost function performance
(average of two runs, 48-transistor circuit)

cost function	maximum perturbation	reduction requested	reduction achieved	cpu time (seconds)	size increase
1	1	35%	38%	441	114%
	2	35%	38%	153	129%
	4	35%	38%	153	204%
	1	50%	*44%	824	150%
	2	50%	50%	537	276%
	4	50%	*47%	637	415%
2	1	35%	38%	297	114%
	2	35%	39%	155	188%
	4	35%	39%	396	282%
	1	50%	50%	972	203%
	2	50%	50%	485	227%
	4	50%	*45%	845	351%
3	1	35%	35%	254	150%
	2	35%	35%	596	210%
	4	35%	38%	155	321%
	1	50%	50%	791	240%
	2	50%	50%	595	304%
	4	50%	*25%	204	152%

* -- failure

The only clear result is that a maximum permutation of 2 is the best choice; no obvious indication as to the most desirable cost function is apparent. Once delay reductions beyond 50% are requested, however, cost functions weighting delay

much more heavily than size are required to obtain solutions.

Obviously, the major bottleneck of the program is the time required to analyze the circuit. As described above, the algorithm requires an analysis for each configuration, and then discards all the information except the delays. Using this extra information to avoid some analyses can result in large performance gains.

Near the solution, most new configurations will be rejected; in fact, most are "obviously wrong" in that they increase the critical path delay sharply. The goal is to *screen* the "obviously wrong" configurations by using approximate timing analysis and avoid fully analyzing them. This idea is similar to the one suggested by Greene and Supowit [4].

Consider the delay along the critical path. If the new configuration increases the delay on the critical path, it will certainly increase the maximum delay through the circuit as a whole. Conversely, if a configuration decreases the critical path delay, it will probably decrease the delay through the circuit as a whole. Thus, analysis of the effect of a change on the critical path is "almost" as useful as analysis of the circuit as a whole, and — since only one path needs to be considered — much quicker.

Since PTA provides symbolic equations for delay at each node, all that needs to be done is evaluate these equations with the new gate sizes included. This is substantially faster than performing a complete timing analysis (see table Y).

Instead of just computing the path delay, the cost of the new configuration given the previous delay equations is calculated. This computation is still substantially quicker than reanalyzing the circuit.

Table 3				
Circuit Analysis vs Equation Evaluation				
size	circuit analysis	evaluation (delay equation)	evaluation (cost equations)	ratio
6	0.95	0.02	0.07	13.6
8	1.38	0.02	0.08	17.3
24	2.15	0.02	0.07	30.7
48	4.70	0.03	0.15	30.1
96	10.60	0.07	0.48	22.1

A standard acceptance test is performed on this estimated cost; if it passes, then full analysis and another acceptance test occur. To avoid biasing the algorithm against configurations which increase the cost (since they now must pass two tests) the same random number is used for both acceptance tests.

Greene and Supowit view the screening process as biasing the generation function; I prefer to consider it as a simple preliminary cost function. In either case, the effect is the same. If a new configuration cannot pass this test, it can be rejected without doing a complete analysis of it. Table 9 in Chapter 7 presents a comparison of the same cost function with and without screening.

Note that this prediction function is not perfectly accurate. This differs from the situation considered by Greene and Supowit: a configuration may pass the screening test only to be rejected. However, a configuration's actual maximum delay can only be greater than the screening function predicts, so no potentially acceptable configurations are ever eliminated in the preliminary stage. As a result, the theoretical properties of the algorithm are unaffected.

Combining Heuristics and Simulated Annealing

One promising areas of exploration is the integration of simulated annealing with other heuristics for sizing. A major problem with heuristic approaches is that it is difficult to say ahead of time which solution is desirable for a critical path — it may be important to size some transistors larger than they would otherwise need to be due to their effect on other paths. If simulated annealing is

combined with a critical-path sizing heuristic, the heuristic can generate an acceptable solution and then rely on simulated annealing to find the proper solution.

One obvious way of combining the two approaches is to alternate them: allow the annealing algorithm to run for a time, reduce the critical path delay using a heuristic, and repeat the process. Another potential method is to use a heuristic in the generation function of the annealing algorithm, with some random perturbations thrown in. More simply, a heuristic may be used to give a starting configuration. Finally, a post-processing heuristic may be used to improve the solution generated by annealing.

One particular area in which such a post-processing heuristic might be useful is in reducing transistor sizes. Since area minimization is less important than reducing the delay to the constraint, transistors not on the critical path tend to be larger than they need to be. Detecting and then examining these transistors is certainly more efficient than allowing the annealing algorithm to continue.

Chapter 5
MOST

The MOST program allows the various approaches to transistor sizing to be tested. It consists of PTA (the Prolog Timing Analyzer), various front ends corresponding to different heuristics, and a simulated annealing interface. MOST is not only a test program for these methods, but is a CAD tool in its own right.

MOST is designed as a component of the ASP (Advanced Silicon Compiler in Prolog) project. Compatibility with ASP is an absolute requirement; this determines the implementation language and interface conventions for MOST. Furthermore, MOST is tuned to its use within ASP.

ASP

The goal of the ASP Project is to produce a high-performance silicon compiler tuned to the development of a logic processor. For a fuller description of ASP, see McGeer *et. al.* [12]. From the viewpoint of MOST, the salient feature of ASP is that it defines a single interface for all its component programs: the constrained hierarchical schematic (CHS). MOST takes its input in this format, and simply attaches additional constraints to the schematic.

A CHS contains a listing of inputs and outputs, as well as additional constraints not important within our framework. Functionally, a CHS may be either a primitive (a transistor, for example) and its associated structure (in this case, the source, gate, and drain signals, as well as the gate size) or a collection of subcells, each of which in turn is a CHS. The hierarchical nature of the CHS allows a building-block approach to silicon compilations, and the notion of constraints interacts well with the Prolog language.

Within ASP, MOST is meant to be run before layout takes place. This implies that the exact lengths of the interconnections are not known, and so some

estimates have to be made on the parasitic resistances and capacitances. Although not a restriction on the program — if exact values are available they can be used — this is the normal situation, and so algorithms are designed with it in mind. In particular, the timing analyzer currently uses the computationally simple but less accurate lumped RC model for delay. The justification for this is that since the uncertainty of the parasitics limits the accuracy of any delay computations, there is no point in spending extra effort to arrive at similarly inaccurate results.

The choice of Prolog as the implementation language (necessary for compatibility with the rest of ASP) strongly influences the design of MOST. The implementation of the simulated annealing algorithm makes great use of the *delayed binding* and *backtracking* provided by Prolog. In effect, implicit pointers throughout the data structures allow many variables to be equated, and then binding one sets all the values simultaneously. When a procedure fails, however, any assignments are undone.

These two features are used for substituting the next configuration into the CHS. The variables in the CHS are collected in a pre-processing stage, and then at each iteration this list of variables is unified with the list of sizes making up the next configuration.

Instead of using delayed binding, the configuration could be substituted into the CHS simply by traversing the entire structure. The cost of this traversal is small compared to the cost of timing analysis, but it can still be substantial; a straightforward implementation, done for comparison's sake, required over .8 cpu seconds for a circuit of 96 transistors. For the same circuit, the unification takes less than .1 cpu second.

Implementation

The algorithms used by MOST have already been described. The two major ways in which MOST differs from the pseudo-code provided above are in attention to storage management and efficiency.

The thorniest implementation problem was that of iteration. Both simulated annealing and the heuristics fall nicely into the paradigm of

```
iterate(Circuit,Configuration) :-
     evaluate(Circuit,Configuration),
     adjust(Configuration,New_configuration),
     iterate(Circuit,New_configruation).
```

The question is how to do this gracefully within the framework of a language that has no destructive assignment. The evaluation step consists of binding the transistor sizes to the current configuration and then performing a circuit analysis. When it comes time to evaluate the next configuration, however, the transistor sizes are no longer unbound variables!

Short of using a technique such as difference lists (which mimics destructive assignment substantially less efficiently) the only alternative is to make use once again of Prolog's backtracking.

```
iterate(Circuit,Configuration) :-
     repeat,
     evaluate(Circuit,Current_configuration),
     adjust(Configuration,New_configuration),
     fail.
```

The fail unbinds the variables, so that they can be rebound for the next evaluation.

The difficulty in this approach is made clear in the call to **evaluate**: just what configuration is being evaluated? The backtracking also throws away the

binding of **New_configuration**. The only way to retain this necessary information is to **assert** it.

```
iterate(Circuit,Configuration) :-
     assert($current(Configuration)),
     repeat,
     retract($current(Current_configuration)),
     evaluate(Circuit,Current_configuration),
     adjust(Current_configuration,New_configuration),
     assert($current(New_configuration)),
     fail.
```

Although any use of **assert** violates Prolog's logical paradigm, this method is fairly reasonable. **assert** is not being used to communicate between procedures, only to retain information over backtracking within a single clause. From an efficiency standpoint, relatively little — less than 1% — of the program's time is spent in this assertion and retraction, despite the fact that somewhat more information than just the configuration needs to be retained: both the heuristics and the screening function of simulated annealing need the critical path equations from the previous configuration.

The symbolic mathematic portion of MOST can evaluate, simplify, and take derivatives of equations. The simplifier is rather mediocre; it does not deal with the distributive law, for example. Its main purpose is to remove zeroes, and the main requirement is that it be fast, so adding in more complicated laws would be counter-productive.

Similarly, the equation evaluator needs to be fast. The equations being evaluated can include unbound variables, which default to zero, so the standard prolog **is** function cannot be used. Analysis showed that the **is** is a factor of 10 faster than a symbolic evaluator, primarily due to the necessity for unifying and setting up a new environment at each level of the expression tree; since evaluation costs are a significant factor in the overall time, this penalty is unacceptable.

Table 4				
is *vs.* evaluation				
circuit size	equation size	evaluation time	**is** time	ratio
6	37	0.06	0.02	3.0
8	109	0.17	0.02	8.5
24	119	0.21	0.02	10.5
48	263	0.42	0.03	14.0
96	551	0.90	0.06	15.0

What really needs to be done is to define an additional operator, say **default**, such that

default(X) = 0 if X is unbound

default(X) = X otherwise.

Then **is** can be used freely simply by applying **default** to all the pontentially unbound variables. However, Cprolog does not permit the definition of new arithmetic operators.

The eventual solution was to prepare a modified version of the interpreter in which unbound variables default to 0. This is a hideous hack, but the performance gains are well worthwhile. The program will still run in standard Prologs, however, due to the use of macro definitions of the relevant procedures.

The modified interpreter asserts the fact **$fast_interpreter** in its environment, and so programs are able to test to see which version of the interpreter is in use. Using the **expand_term** preprocessor, the changes can be implemented in a completely transparent manner. When a program is being read in, **expand_term** is applied to each clause (this is also how grammar rules are implemented). If no **expand_term** succeeds, then the clause is simply asserted as is; otherwise, the second argument of the appropriate **expand_term** is asserted.

```
% if Condition is true, assert the Then clause
expand_term(ifdef(Condition,Then,Else),Then) :-
     Condition,!.
% otherwise, the Else clause
expand_term(ifdef(Condition,Then,Else),Else) :-
     + Condition,!.

ifdef($fast_interpreter,
(dummy(...) :-
     ...
     X is Eq
     ...),
% otherwise
(dummy(...) :-
     ...
     evaluate(Eq,X),
     ...)).
```

The modification to the interpreter greatly increases MOST's speed. Table 10 in chapter 7 contains the statistics; the gain is always at least a factor of two.

Chapter 6
PTA

PTA is a vital component of MOST; it provides information for both the heuristic and the simulated annealing top ends. Although it is scarcely an advance on the state of the art — it is both slower and somewhat less accurate than Crystal. for example — it has several interesting features. In particular, it is tuned to repeated use as part of a sizing program, and thus preprocesses as much of its input as possible; it orders series transistors if their order is initially unknown; it provides symbolic equations for delay at the various nodes; and it capitalizes on the hierarchical structure of the input schematic.

It is possible to modify PTA to use a more accurate delay model, so the lack of accuracy is not inherent. PTA also has the ability to treat higher levels of abstractions — logic gates or even macro cells — as primitives if the proper delay model is provided. Currently, however, MOST does not take advantage of this capability.

An obvious question is the necessity of designing a new timing analyzer. Why not just use Crystal, for example, with (if necessary) a few modifications? One objection, of course, is that Crystal (or any other timing analyzer) is not written in Prolog; however, the aesthetic desire for a system written entirely in Prolog is not sufficient justification for reinventing the wheel.

In order to understand the reasons for writing PTA from scratch, it is necessary to understand its functions. The justification will thus be postponed until after a description of PTA and its implementation.

Implementation

PTA takes a CHS as its input. The output of PTA is the same CHS, with additional timing information attached; the information is sufficient to reconstruct

the critical path to any node within the circuit. In addition, any initially unordered series transistors will have orders attached.

If the CHS has subcells, then each subcell is analyzed in turn. The delay at an output of the CHS is simply the maximum of the delays of that output in the subcells. This process is repeated recursively until a primitive element is reached. In the standard version of PTA, a primitive is a collection of transistors connecting a single input to a single output.

In order to process a primitive element, the delays of all its inputs must first be known. This may involve first processing other CHSs whose outputs are inputs to the current CHS; since the information is retained, this does not cause any additional work, just reorders the schedule. Each path to each input will thus be considered. Once the input delays are known, a delay modeler for the primitive element is called on to calculate the output delays.

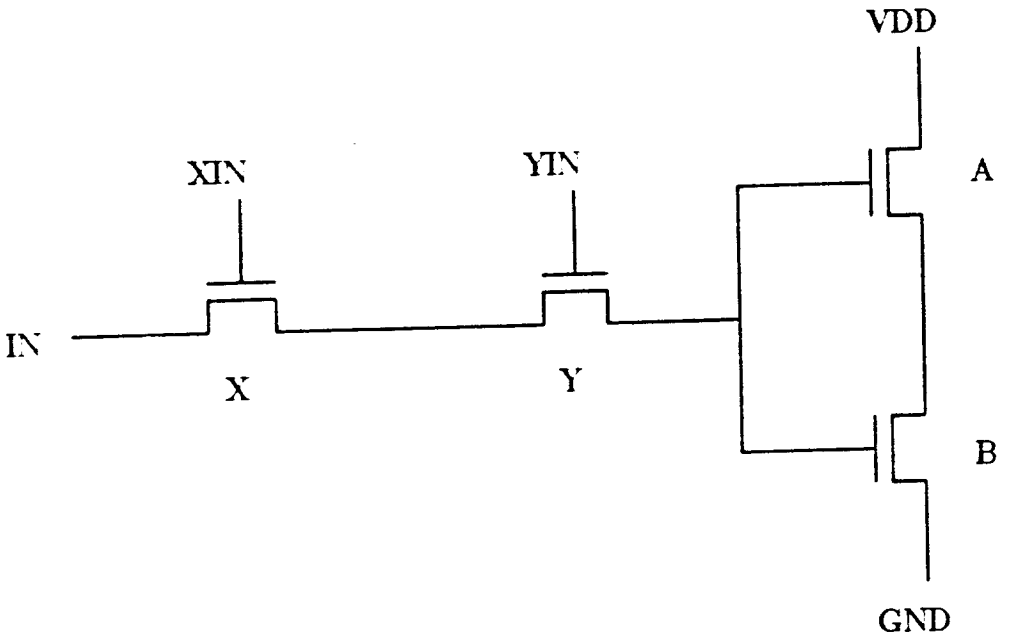
```
analyze_chs(CHS) :-
     is_primitive(CHS),
     known_input_delays(CHS),
     process_primitive(CHS).
analyze_chs(CHS) :-
     + is_primitive(CHS),
     subcells(CHS,Cells),
     apply_to_each(analyze_chs,Cells).
```

This scheme has several benefits. In the first place, it permits any level to be viewed as a primitive, as long as the required delay modeler is supplied. For example, rather than going down to the transistor level, logic gates might be considered primitive. Secondly, it capitalizes on the hierarchical structure, which limits the number of paths through any one cell. Finally, it eases the burden on the delay modeler, which is able to assume that all the input delays are known.

The delay model currently used for transistors is the lumped RC model, which views the entire resistance and capacitance of a stage as concentrated at

the end of the stage. Clearly, this model is pessimistic; however, it is reasonably accurate, and computationally fast.

Each transistor in turn is considered as the *trigger* transistor, or the last transistor to change. The delay on the stage given this choice of trigger transistor is then computed, and the maximum of these delays is taken as the output delay. The trigger transistor is also recorded, allowing later reconstruction of the critical path.



Transistors X and Y are the possible choices for trigger transistor.

If the stage consists of the set of transistors X, each with its associated resistance and capacitance; interconnections I; and drives capacitive load C_{out}, then the delay with t as trigger is

$$D_t = \text{Input delay}_t + \left(\sum_{x \in X} R_x + \sum_{i \in I} R_i + R_{in}\right) * \left(\sum_{x > t} C_x + \sum_{i > t} C_i + C_{out}\right)$$

where ">" means "follows in the path." In terms of the above diagram, and neglecting interconnect resistance for simplicity,

$$C_{out} = C_A + C_B$$
$$D_X = \text{Input delay}_{XIN} + (R_{in} + R_X + R_Y) * (C_X + C_Y + C_{out})$$
$$D_Y = \text{Input delay}_{YIN} + (R_{in} + R_X + R_Y) * (C_Y + C_{out})$$

If the primitive element contains unordered series transistors, they are ordered before this computation takes place. Having known input delays allows this ordering: otherwise, the final order of the transistors is not know when their delay is calculated, and some assumption must be made. The only safe assumption is the worst-case one for each transistor, but this leads to wildly pessimistic results.

Determining the input resistance and output capacitance of a primitive element will involve tracing a path through the circuit. The path must come from an input to the circuit to the input of the CHS being considered; in the case of a transistor, the path goes to the source. Finding paths is a classic prolog pseudo-breadth-first search problem; the simplistic implementation is quite straightforward:

```
path(X,X,[]).          % path from X to X
path(X,Y,[connection(X,Z)|P]) :-
     connection(X,Z),
     path(Z,Y,P2).
```

The use of **connection** is meant to hide the exact structure of CHSs; X and Y are connected if there is a primitive CHS with X as an input and Y as an output. In practice, however, this simple algorithm is not sufficient, since some additional checking needs to be done.

A cycle among CHSs implies a memory node or a latch. In this case, the cyclic path is ignored. This check is potentially rather time-consuming, if paths are long, since it uses an order N^2 algorithm. In practice, however, it is relatively inexpensive.

In order to avoid considering paths which are blocked by non-overlapping signals, the list of signals influencing each path is retained. Only signals which have been specified as potentially non-overlapping are included in this list. All

values are kept for each set of signals.

In effect, this mechanism trades the space for storing all the different combinations for the time required to do recomputation if each case is considered separately as it is in a traditional timing analyzer. Potentially, storage can increase combinatorially with the number of non-overlapping signals. Most paths, however, do not involve more than one or two such signals, so the price is not too great. Furthermore, it is exactly these paths where the values computed can be used for multiple cases.

With these additions, the path algorithm is somewhat more complex:

```
path(X,X,Signals,Path,Path).
path(X,Y,Signals,Path_so_far,Path) :-
    connection(X,Z),
    % check for circularities
    + member(connection(X,Y),Path_so_far),
    % check for overlapping signals
    signals(connection(X,Y),S),
    overlap(S,Signals),
    add_signals(S,Signals,New_signals),
    path(Z,Y,New_signals,[connection(X,Y) | Path_so_far],Path).
```

PTA is tuned to its mode of use within MOST: repeated analyses of the same schematic with additional sizes attached. As a result, before the first analysis of a given circuit, it does as much *preprocessing* as possible, since preprocessing costs only need to be paid once. All the paths within each individual CHS are computed and stored (the use of hierarchies keeps this space requirement from growing exponentially), as are symbolic formulas for each node's output capacitance. For a 100-transistor circuit, preprocessing requires 8 cpu seconds; by comparison, the rest of the analysis only takes 13 seconds. The same algorithm with the preprocessing removed requires 27 cpu seconds, substantially more than the sum of the two times.

Justification

The strongest argument in favor of a completely new timing analyzer is the absence of any Prolog timing analyzer. This is not just an aesthetic argument. Due to the nature of the interpreter, interfacing problems are particularly daunting. It is relatively easy to make use of a C procedure which returns a numeric value simply by adding a hook to the interpreter allowing function calls, but returning a structure is far more difficult.

In the first place, Prolog procedures do not return values, but rather unify them with their arguments. This problem requires only "syntactic sugar" to avoid, but the two languages represent structures differently; Prolog structures need to be converted into C's format when the procedure is called, then the C structures must be massaged to convert them into the proper Cprolog format. Finally, Prolog variables are fundamentally different than C variables (there is no Prolog analog to assignment, for example, and pointers are implicitly dereferenced); it is not at all clear how to remedy this difficulty.

Beyond these language issues, we reach the question of how much an existing timing analyzer would need to be modified to fill its role as part of MOST. For concreteness, Crystal is considered. At least four areas need to be dealt with:

1) MOST requires symbolic delay equations from the timing analyzer. It is possible to add these to Crystal in much the same way they have been implemented in the current version of PTA; to correspond to unbound variables in the Prolog version, pointers to unfilled memory locations could be used in C.

2) PTA must order series transistors in the cases where the order is not fixed ahead of time. No facility for unordered lists is present in Crystal, and even if this were added there would still be major difficulties. PTA only considers a CHS once all its input delays are known; given this

information, it is possible to decide on an ordering for the transistors. Crystal does not make such a stipulation, so the ordering information may not be known or — even worse — may change as the circuit is being analyzed.

3) In the ASP environment, transistor sizing takes place before layout. This implies that the exact interconnect capacitances are not known, and some estimates must be made. This would be rather simple to do within the framework of Crystal.

4) Crystal cannot deal with non-overlapping signals. A human designer can do case analysis by fixing on each possibility in turn; this results in significant recomputation, however. Once again, this capability could be added into Crystal in the same way it has been implemented in PTA.

Most of these features, then, could be added into the existing framework of Crystal. On the other hand, these areas consumed the bulk of time implementing PTA, and it seems fair to assume that as much time would have been required to modify Crystal. The final decision on whether to use Crystal was that the implementation difficulties, unordered transistors, and the all-Prolog aesthetic argument outweighed the already existing speed and accuracy of Crystal.

In retrospect, substantially more time than expected was spent implementing PTA; however, almost all of this time was spent in the areas which would have had to be added to Crystal as well. PTA is substantially slower than Crystal, and because of its choice of the lumped RC timing model, somewhat less accurate. Despite this, I believe the decision was a good one.

PTA's accuracy can be improved by incorporating the distributed RC model and taking waveform shape into account. These gains will, of course, be limited by the fact that interconnect lengths are only estimates, but should make PTA's accuracy competitive to other timing analyzers. Furthermore, from a

development standpoint, there was a great advantage in being able to work with the relatively simple equations of the lumped RC model. The extra accuracy of the slope model in particular is accompanied by a dramatic increase in complexity of the equations.

Admittedly, PTA is at least an order of magnitude slower than Crystal, and for some choices of algorithms this is the limiting factor for the MOST program. On the other hand, this difference is simply due to the fact that Cprolog is interpreted Estimates for the performance of the PLM machine [1], in conjunction with the Berkeley Prolog compiler [18], predict a 200-fold increase in speed.

Chapter 7

Results

Five algorithms have been evaluated.

SIMPLE

— a simple scapegoat heuristic: the size of the "most useful" transistor (the transistor whose modification does the most good) is increased by one

MS — the size of the most useful transistor is increased by a varying amount

MT — the size of any transistor whose increase would reduce delay is increased by one

CP — a critical path heuristic which uses partial derivative information for the path sizing

ANNEAL

— simulated annealing using screening and the cost function

$$\text{Cost} = 5*\text{Penalty}_{max}+5*\sum_{i\text{nodes}}\text{Penalty}_i+\text{TotalSize}$$

Due to the random nature of this algorithm, there is a fair amount of uncertainty in the results quoted for ANNEAL, most of which are based on only a few runs.

The first question to be considered is how well the various algorithms perform on two mid-sized circuits. A one-bit full adder consisting of 24 transistors is a small enough circuit that all the algorithms perform reasonably well.

Table 5

Algorithm performance — 1-bit adder (24 transistors)

algorithm	delay reduction requested	delay reduction achieved	size increase	timing analyses	cpu time (seconds)
SIMPLE	30%	31%	13%	7	18.7
	50%	51%	63%	31	80.3
	60%	60%	131%	64	163.0
MS	30%	44%	67%	6	16.4
	50%	50%	152%	13	34.4
	60%	63%	298%	24	63.1
MT	30%	31%	19%	3	8.7
	50%	50%	77%	11	29.4
	60%	60%	148%	20	52.8
CP	30%	33%	25%	3	15.3
	50%	51%	94%	9	47.2
	60%	60%	127%	15	80.7
ANNEAL	30%	42%	152%	6	23.3
	50%	52%	218%	11	40.3
	60%	60%	194%	84	230.9

Doubling the size of the circuit to two bits and 48 transistors causes problems for the SIMPLE heuristic and the simulated annealing algorithm. The results reported by Fishburn and Dunlop are included for comparison; the time is for a 68000-based workstation running C code.

Table 6

Algorithm performance — 2-bit adder (48 transistors)

algorithm	delay reduction requested	delay reduction achieved	size increase	timing analyses	cpu time (seconds)
SIMPLE	30%	31%	15%	15	93.0
	50%	50%	74%	72	430.0
MS	30%	40%	35%	8	51.4
	50%	52%	272%	34	207.5
	60%	60%	275%	50	301.4
MT	30%	38%	29%	5	32.5
	50%	50%	90%	15	92.4
	60%	61%	261%	48	285.0
CP	30%	32%	35%	3	48.3
	50%	52%	173%	11	192.9
	60%	61%	355%	22	373.5
ANNEAL	30%	32%	123%	9	79.4
	50%	51%	318%	75	553.0
TILOS		43%	32%		6

Note that although it requires more cpu time, SIMPLE is the most effective in limiting transistor size increase. In general, cleverer algorithms tend to overestimate the sizes of transistors not on the eventual critical path. MS is particularly prone to this problem because of the difficulty of deciding on the proper size for a transistor before surrounding transistors have had their final size determined.

PTA is currently limited to circuits of approximately 100 transistors. As a result, it is difficult to say how well various algorithms scale to larger circuits. The available data makes it seem that they are roughly quadratic in the size of the circuit.

Table 7

Algorithm performance *vs.* circuit size

		Circuit Size			
Algorithm	Reduction	8	24	48	96
SIMPLE	30%	8.0	18.7	93.0	451.2
	50%	11.5	80.3	430.0	*
MS	30%	6.7	16.4	51.4	239.0
	50%	6.7	34.4	207.5	1095.8
MT	30%	8.2	8.7	32.5	84.9
	50%	11.9	29.4	92.4	331.2
CP	30%	11.0	15.3	48.3	137.6
	50%	11.0	47.2	192.9	?
ANNEAL	30%	4.5	23.3	79.4	?
	50%	16.1	40.3	553.0	?

* — failed to satisfy request

? — data not yet available

As the circuit gets larger, time for symbolic derivatives increases dramatically. For even the 48-transistor circuit, over 50% of the computation time is spent taking derivatives; for 96 transistors, the percentage increases to above 70%. A faster derivative procedure should make this method more competitive with the others.

The somewhat arbitrary example of a chain of four inverters driving a fairly large output capacitcance makes a good test of how the algorithms perform while

driving the ciucuit as close as possible to its optimum size. Although this is rather impractical — reducing the delay by 85% increases the circuit's size by a factor of 15 — it is nonetheless a measure of the capabilities of the algorithms.

Table 8

Performance near optimum configuration
Chain of 4 inverters — 8 transistors
(figures are cpu times to attain reductions)

requested delay	Algorithm				
	SIMPLE	MS	MT	CP	ANNEAL
30%	8.0	6.7	8.2	11.0	4.5
40%	8.0	6.7	8.2	11.0	6.4
50%	11.5	6.7	11.9	11.0	16.4
60%	18.5	6.7	15.4	11.0	20.0
70%	38.9	11.9	26.0	19.6	29.5
80%	126.9	30.3	51.7	47.8	116.0
85%	317.6	55.5	102.7	114.5	416.5
90%	*	394.1	*	*	*

* — failed to satisfy request

The screening procedure in the simulated annealing algorithm does indeed boost efficiency substantially. The savings increase with the size of the circuit (as timing analysis becomes more expensive) and with the percentage reduction requested (as more configurations become "obviously wrong").

The same cost function was used both with and without screening. One method of seeing the increase in efficiency is to calculate the "success rate" — how often an evaluation results in an acceptance.

Table 9

Advantages of screening
(averages of 10 runs)

Circuit Size	Delay Reduction	Screen?	Evaluation Percentage	Success Percentage	Cpu Time
24	30%	NO	100%	58%	23.0
		YES	70%	74%	19.7
24	50%	NO	100%	40%	74.4
		YES	57%	77%	54.6
48	30%	NO	100%	37%	122.6
		YES	47%	68%	80.9
48	50%*	NO	100%	15%	1100.1
		YES	40%	37%	631.7

* -- only one run due to cpu time

Finally, it is worth investigating how useful the modification to the interpreter actually was. Table 10 includes the ratio of the time to evaluate an expression to the time required by **is**. Simply multiplying this ratio by the time spent in evaluating equations using the fast interpreter will give a fairly good estimate of the time required to evaluate the equations in the standard interpreter. Different algorithms do different amounts of evaluation, so the exact benefit they obtain differs, but it is invariably large.

Table 10

Performance gains due to modified interpreter

Algorithm	Circuit Size	Modified Interpreter (measured) Is time	Modified Interpreter (measured) Total	Standard Interpreter (projected) Eval Time	Standard Interpreter (projected) Total	Ratio
MS	8	1.4	6.7	11.9	17.2	2.6
	24	9.1	24.4	95.5	110.8	4.5
	48	65.7	207.5	919.8	1061.6	5.1
	96	383.7	585.3	5755.5	5957.1	10.2
MT	8	2.9	11.9	24.6	33.6	2.8
	24	7.4	29.4	77.7	99.7	3.4
	48	27.8	92.4	389.2	453.8	4.9
	96	138.2	331.2	2073.0	2266.0	6.8
CP	8	0.4	1.1	3.4	4.1	3.7
	24	5.1	47.2	53.5	95.6	2.0
	48	14.0	192.0	196.0	374.0	1.9
ANNEAL	8	2.9	20.6	24.6	42.3	2.1
	24	5.6	40.3	58.8	93.5	2.3
	48	182.5	553.0	2555.0	2925.5	5.3

Chapter 8
Conclusion

Summary of results

Both simulated annealing and heuristic methods can reduce delay through a circuit by 50% in a few minutes of CPU time using a simple delay model. Heuristics are tend to be more efficient and produce smaller final circuits; even simple heuristics give surprisingly good results. Although no guarantees can be provided, several of these approaches almost invariably succeeds in satisfying reduction requests up to 60%.

Using symbolic equations is a key to improved performance both for heuristics and simulated annealing. Using a more accurate delay model might cause the complexity of these equations to increase dramatically, and so it is not clear how this would affect the program's speed.

The limitations on circuit size are largely a function of the Prolog implementation in use, particularly its failure to perform tail-recursion optimization. Other than this, performance scales fairly well with size.

Future work

Many promising areas for research are still almost untouched. Several have been mentioned in passing above; this final discusses them in somewhat greater detail.

The most attractive possibility is taking advantage of the circuits hierarchical structure. As mentioned, PTA is able to view different levels of abstraction as primitive; this feature was added primarily for the benefit of MOST, but no use has been made of it so far. Instead of sizing the entire circuit simultaneously, it should be more efficient to assign delay goals to cells and then size the cells recursively.

Annealing and heuristics should certainly be combined, as should heuristics of different types. For different circuits, different approaches are desirable; some way of determining what method is right for a given circuit would be extremely useful.

PTA is easily modifiable to include a more accurate delay model. The Penfield-Rubenstein-Horowitz distributed RC model is only slightly more complex than the lumped RC model, and current sizing techniques should continue to perform much the same. Models taking the waveform's slope into account cause more difficulty, but provide potentially large rewards. Of course, from the standpoint of MOST's usage within ASP, the increased accuracy will do little good, due to the estimates of interconnect length, but they are important for use as a stand-alone tool.

From an aesthetic standpoint, using heuristics is rather unsatisfactory. Fishburn and Dunlop's work [2] points the way towards a sounder theoretical basis, but is currently restricted to the distributed RC model. This result needs to be extended towards a more general model. Additionally, decomposition techniques such as Matson's — much like the hierarchical decomposition described above — show great promise.

Acknowledgments

First of all, my parents and my brother Greg deserve all the thanks in the world ... but they know that already. Blood is thicker than water, so others will have to be content with the second paragraph.

My ASP project colleagues Rick McGeer and Bill Bush spent vast amounts of time discussing the issues with me, and later gave incisive comments on the rough drafts of this paper. Paul Chang's PC was indispensible, and he also had to live with my traumas and occasional bad moods all semester long. Al Despain, my

advisor, was also incredibly helpful.

The list goes on Andrew Kahng provided useful feedback. Ubli Mitra gave me incentive, and her Macintosh bailed me out of a jam. Jordan Hayes chipped in with some needed typesetting help. Colleagues on the Aquarius project were kind enough to listen to me, and I should really thank Paul again for putting up with me this semester. Mike Clancy, however, failed to give me any cheesecake, despite repeated subtle hints.

BIBLIOGRAPHY

1. Dobry, T., "A Prolog Machine Architecture," *UCB/CSD Technical Note*, 1984.
2. Fishburn, J. P. and A. E. Dunlop, "TILOS: a Posynomial Programming Approach to Transistor Sizing," *Proc. ICCAD*, pp. 326-8, 1985.
3. Glasser, L. and L. Hoyte, "Delay and Power Optimization in VLSI Circuits," *Proc. 21st DAC*, 1984.
4. Greene, J. W. and K. J. Supowit, "Simulated Annealing without Rejected Moves," *Proc. ICCD*, 1984.
5. Hedlund, K., "Models and Algorithms for Transistor Sizing in NMOS Circuits," *Intl. Conf. on CAD*, 1984.
6. Jouppi, N. P., "TV: an nMOS Timing Analyzer," *Proc. 3rd Caltech Conf. on VLSI*, 1985.
7. Kao, W. H., N. Fathi, and C.-H. Lee, "Algorithms for Automatic Transistor Sizing in CMOS Digital Circuits," *Proc. 22nd DAC*, 1985.
8. Kirkpatrick, S., C. Gelatt, and M. Vecchi, "Optimization by simulated annealing," *IBM Yorktown Heights Tech. Report*, 1982.
9. Lee, C. M. and H. Soukup, "An Algorithm for CMOS Timing and Area Optimization," *IEEE Journal of Solid State Circuits*, vol. SC-19, 1984.
10. Matson, M. D., "Optimization of Digital MOS VLSI Circuits," *Proc. Chapel Hill Conf. on VLSI*, 1985.
11. Matson, M. D., "Macromodeling of Digital VLSI Circuits," *Proc. 22nd DAC*, pp. 144-51, 1985.
12. McGeer, R., B. Bush, A. Despain, and J. Pincus, *The ASP Silicon Compiler*, (submitted to) 23rd DAC, 1986.
13. Mead, T.-M. Lin and C. and C. Mead, "Timing Simulation of Digital Integrated Circuits," *Proc. Conf on Adv. Research in VLSI*, pp. 93-99, 1984.
14. Ousterhout, J., "Switch-Level Models for Digital MOS VLSI," *Proc. 21st DAC*, 1984.
15. Penfield, P. Jr. and J. Rubenstein, "Signal Delay in RC Tree Networks," *Proc. 2nd Caltech Conf on VLSI*, pp. 263-84, 1981.
16. Pereira, L., *Cprolog Users' Manual*, p. Edinburgh University, 1982.
17. Pincus, J. and A. Despain, *Transistor Sizing Using Simulated Annealing*, (submitted to) 23rd DAC, 1986.
18. Roy, P. Van. "A Prolog Compiler for the PLM," *UCB/CSD Tech. Report*, 1984.
19. Ruehli, A. E., P. K. Wolff, and G. Gortzed, "Analytic Power/Timing Optimization Techniques for Digital Systems," *Proc. 14th DAC*, 1977.
20. Sechen, C. and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package," *IEEE J. of Solid-State Circuits*, vol. SC-20, 1985.
21. Trimberger, S. M., "Automated Performance optimzation of Custom ICs," *Proc. International Symposium on Circuits and Systems*, pp. 194-7, 1983.

Appendix
Source Code of MOST

```
% this is the top-level file which causes the others to be loaded in

:-([-'UTILS/macros',    % this has to be first so it applies to the others
    -'anneal',
    -'heuristic',
    -'PTA/pta',
    -'PTA/ppp',
    -'PTA/order',
    -'PTA/primitive',
    -'PTA/critpath',
    -'PTA/delay',
    -'UTILS/utils',
    -'UTILS/symbolic',
    -'UTILS/structs',
    -'UTILS/print',
    -'UTILS/makechs',
    -'UTILS/random',
    -'UTILS/minimize']).
```

```
% ifdef macros.  In order to have precedence over the other macros,
% these need to be applied first, and so must be 'asserta'ed.

:- asserta((expand_term(ifdef(Condition,Clause1,Clause2),Clause2) :- ! )).
:- asserta((expand_term(ifdef(Condition,Clause1,Clause2),Clause1) :-
    Condition, !)).
```

```
% prolog code to do simulated annealing


measure(Delay) :-
        N is cputime,
        pp(Chs),
        make_sizes(Chs,Vars),
        initialize(Chs,Delay,Vars,Initial),
        anneal(Delay,Chs,Vars,Final,Cost,Actual),
        N1 is cputime,
        Pct is Actual/Initial,
        print('Delay reduced from '),print(Initial), print(' to '),
        print(Actual),print(' ('),print(Pct),print('%)'),nl,
        print('Total transistor size '),total_size(Final,Tsize),print(Tsize),nl
        print('cpu time required is '),Diff is N1 - N,
        print(Diff),print(' seconds'),nl.
measure(_) :-
        print('Failure'),nl,
        !,
        fail.

initialize(L,Delay,Vars,Init_delay) :-
        length(Vars,Number),
        print('There are '),print(Number),print(' transistors to size'),nl,
        init_configuration(Vars,Init),
        cost(L,Delay,Vars,Init,Init_cost,Init_delay,Init_eq,Init_other),
        init_stopinfo(Init_cost,Stopinfo),
        init_temperature(Delay,Init_delay,T),
        note_values(Init,Init_cost,Init_delay,Init_eq,Init_other),
        fail.
initialize(L,Delay,Vars,Init_delay) :-
        $current_delay(Init_delay),
        !.

anneal(Delay,Chs,Vars,X,Xc,Xd) :-
        repeat,
        get_temperature(T),
        get_stopinfo(Stopinfo),
        iterations_at_temp(T,N),
        inner_loop(N,T,Delay,Chs,Vars,X,Xc,Xd),
        update_stopinfo(Stopinfo,Xc,New_si),
        ((stop(Delay,Xd,New_si);          % success
% or
            give_up(New_si));             % failure
% or
            update_temperature(T,New_t),          % keep going
            fail).                                % the fail returns to the repea

% inner loop goes through N iterations at the specified temperature
inner_loop(N,T,Delay,Chs,Vars,J,Cost,Actual) :-
```

```
        range(1,I,N),
        get_values(X,Xcost,Xdelay,Xdeq,Xoeq),
        generate(X),
        screen(X,Xdeq,Xoeq,Xcost,Delay,T,R),
        make_real_configuration(X,J),
        cost(Chs,Delay,Vars,J,Cost,Actual,Deq,Oeq),
        accept(Xcost,Cost,T,R),
        replace_values(J,Cost,Actual,Deq,Oeq),
        Actual < Delay,          % succeed only if done
    % otherwise, fail and the retry goes back to 'range'
        !.

inner_loop(_,_,_,_,_,_,X,Xcost,Xdelay) :-
        get_values(X,Xcost,Xdelay,Xdeq,Xoeq),
        !.

% the screening function -- throw out "obviously wrong" configurations
screen(Config,Deq,Oeq,Old_cost,Delay,T,R) :-
        size_cost(Config,Size_cost),
        Diff is Deq - Delay,
        max(0,Diff,Delay_cost),
        map(other_penalty,Delay,Oeq,Penalties),
        sum(Penalties,Other_cost),
        make_cost(Delay_cost,0,Size_cost,Test_cost),
        random(R),
        !,
        accept(Old_cost,Test_cost,T,R),
        !.

other_penalty(Delay,Eq,Penalty) :-
        Actual is Eq - Delay,
        max(0,Actual,Penalty).

accept(Xcost,Jcost,T,R) :-
        Del_c is Jcost - Xcost,
        f(Del_c,T,Y),
        R < Y.

f(Del_c,_,1) :-
        Del_c < 0,
        !.
f(Del_c,T,Y) :-
        Y is exp(-Del_c/T),
        !.
% annealing utility functions, including intializing and updating
% parameters.  most of these are very sketchy, and lots of useful
% work could no doubt be done here

init_temperature(Delay,Init_delay,T) :-
        T is (Init_delay - Delay)/4,
```

```
        asserta($current_temp(T)),
        !.

init_stopinfo(Init,[Init,0,1,2]) :-  % as long as they're different, it's cool
        asserta($current_info([Init,0,1,2])),
        !.

% currently, iterations_at_temp doesn't depend on the temperature.  clearly
% it should for better performance.  sorry.
iterations_at_temp(Temp,25) :- !.

% this should also be somewhat more complex
update_temperature(T,Newt) :-
        Newt is 0.8*T,
        asserta($current_temp(Newt)),
        !.

get_temperature(T) :-
        retract($current_temp(T)),
        !.

update_stopinfo([X1,X2,X3,_],X,[X,X1,X2,X3]) :-
        asserta($current_info([X,X1,X2,X3])),
        !.

get_stopinfo(Stopinfo) :-
        retract($current_info(Stopinfo)),
        !.

stop(Delay,Xdelay,Stopinfo) :-
        Xdelay < Delay.

% give up if no change (in cost) in three iterations
give_up([X,X,X,X]).

% try to make the asserts and retracts as transparent as possible
note_values(X,Xc,Xd,Xdeq,Xoeq) :-
        asserta($current_config(X,Xdeq,Xoeq)),
        asserta($current_cost(Xc)),
        asserta($current_delay(Xd)),
        !.

get_values(X,Xc,Xd,Xdeq,Xoeq) :-
        $current_config(X,Xdeq,Xoeq),
        $current_cost(Xc),
        $current_delay(Xd),
        !.

replace_values(X,Xc,Xd,Xdeq,Xoeq) :-
        retract($current_config(_,_,_)),
```

```
        retract($current_cost(_)),
        retract($current_delay(_)),
        note_values(X,Xc,Xd,Xdeq,Xoeq),
        !.


% interface the simulated annealing algorithm with the timing analyzyer


% cost takes the variables as its third argument, and the actual configuration
% as its fourth.  This binds them, and thus sets the sizes in the Chs itself

cost(Chs,Delay,Config,Config,Cost,Actual_delay,Delay_eq,Other_eq) :-
        process_chs([],Chs),
        find_critical_path(Chs,Cp),
        max_delay_cost(Chs,Cp,Delay,Actual_delay,Delay_cost,Delay_eq),
        other_delay_cost(Chs,Cp,Delay,Other_cost,Other_eq),
        size_cost(Config,Size_cost),
        make_cost(Delay_cost,Other_cost,Size_cost,Cost),
        !.


max_delay_cost(Chs,Cp,Delay,Actual_delay,Cost,Eq) :-
        delay_equation(Cp,Eq),
        delay(Cp,Actual_delay),
        Diff is Actual_delay - Delay,
        max(Diff,0,Cost).

other_delay_cost(Chs,Cp,Delay,Total_over,Eq) :-
        all_delays_over(Chs,Delay,Total_over,Eq).

size_cost(Config,Size) :-
        total_size(Config,Size).

all_delays_over(Chs,Delay,Total,Eqs) :-
        signals(Chs,Signals),
        map(delay_equation,Signals,Eqs),
        map(penalty,Delay,Signals,Penalties),
        sum(Penalties,Total).

penalty(Delay,Entry,Penalty) :-
        delay(Entry,This_delay),
        Diff is This_delay - Delay,
        max(Diff,0,Penalty).

total_size(Config,Size) :-
        map(gate_size,Config,Combined),
        sum(Combined,Size).

% generate modifies the old configuration, which is in the form Size+Change
```

```
% by binding Change to some number
generate(Old) :-
        minimum_gate_size(Min),
        apply_to_each(perturb,Min,Old),
        !.

perturb(_,Size+Change) :-
        number(Change),
        !.

perturb(Min,Size+Change) :-
        var(Change),
        perturbation(Change),
        Size + Change >= Min,
        !.

perturb(Min,Size+0) :-
        !.
perturb(_,Size) :-
        number(Size),
        !.


perturbation_size(2).                   % maximum perturbation

perturbation(Change) :-
        perturbation_size(Max_change),
        Mmc is -Max_change,
        Maxc is Max_change+1,
        random_int(Mmc,Maxc,Change),
        !.


% the initial configuration is simply with all gates at the minimum size

init_configuration(Vars,Init) :-
        map(init_gate_size,Vars,Init).

init_gate_size(X,Y+_) :-
        var(X),
        minimum_gate_size(Y).

% unless, of course, they happen to have a size already fixed
init_gate_size(X,X) :-
        number(X).

make_cost(Delay_cost,Other_cost,Size_cost,Cost) :-
        max_weight(K1),
        all_weight(K2),
        size_weight(K3),
```

```
        Cost is K1 * Delay_cost + K2 * Other_cost + K3 * Size_cost.

max_weight(5).
all_weight(5).
size_weight(1).

make_real_configuration(Config,Trial) :-
        map(free,Config,Trial).

% free changes from the form Size+Change (with both bound) to the form
% Newsize+_, where Newsize = Size+Change
free(S,Result+_) :-
        gate_size(S,Result).
```

```
% try a heuristic
try_heuristic(Delay) :-
        clear_globs,
        N is cputime,
        pp(L),
        make_sizes(L,Vars),
        init_configuration(Vars,Init),
        asserta($current_configuration(Init)),
        heuristic_iterate(L,Vars,Delay),
        print('final configuration is '),print(Vars),nl,
        total_size(Vars,Size),print('final size is '),print(Size),nl,
        print('time required is '),Time is cputime - N,print(Time),nl.

heuristic_iterate(Chs,Vars,Delay) :-
        repeat,
        current_configuration(Config),
        cost(Chs,Delay,Vars,Config,_,Actual,Delay_eq,_),
        (Actual =< Delay  -> print('whee!'),nl;
                             make_next_config(Config,Delay,Delay_eq),
                             fail).

make_next_config(Config,Cost,Eq) :-
        collect_vars(Eq,[],Vars),
        partials(Eq,Vars,Derivs),
        map(zero,Vars,Constraints),      % oversimplification; they might
                                         % already have values, in which case
                                         % they could decrease
        minimize(Eq,Cost,Vars,Constraints,Derivs,New_cost),
        make_real_configuration(Config,New),
        retract($current_configuration(_)),
        asserta($current_configuration(New)),
        !.

current_configuration(X) :-
        $current_configuration(X),
        !.
% MS heuristic -- increase size of transistor by more than 1

clear_globs :-
        abolish(best,2),
        abolish(this,1),
        abolish(bump,1).

choose_best(Config,Cost,Eq,Actual) :-
        asserta(best(0,Cost)),
        try_each(1,Config,Eq,Cost),
        retract(best(_,Actual)).

try_each(_,[],_,_).
ifdef($fast_interpreter,(
```

```
% hacked interpreter
try_each(N,[1 | Config],Eq,Cost) :-
        This_one is Eq,
        replace_if_necessary(N,This_one,Cost),
        fail),
% normal interpreter
(try_each(N,[1 | Config],Eq,Cost) :-
        evaluate(Eq,This_one),
        replace_if_necessary(N,This_one,Cost),
        fail)).
try_each(N,[Mod | Config],Eq,Cost) :-
        N1 is N+1,
        try_each(N1,Config,Eq,Cost),
        choose_size(Mod,Eq,Best,N).

choose_size(Mod,Eq,Best,N) :-
        is_best(N),
        !,
        choose_individual_size(Mod,Eq,Best).

choose_size(0,_,_,_).

replace_if_necessary(N,Current,Cost) :-
        best(_,Best),
        Current < Best,
        retract(best(_,_)),
        asserta(best(N,Current)),
        !.

is_best(N) :-
        best(N,_),
        !.

choose_individual_size(Mod,Eq,Best) :-
        best(_,One),
        try_sizes(Mod,2,Eq,One,Best,1,Choice),
        asserta(best_size(Best,Choice)),
        fail.
choose_individual_size(Mod,Eq,Best) :-
        retract(best_size(Best,Mod)),
        !.

try_sizes(Mod,Mod,Eq,Best,Best,Choice,Choice) :-
        max_change(K),
        Mod > K,
        !.

ifdef($fast_interpreter,(
try_sizes(Mod,Mod,Eq,Best,Best,Choice,Choice) :-
        This is Eq,
```

```
        asserta(this(This)),
        This > Best,
        !),
% usual interpreter
(try_sizes(Mod,Mod,Eq,Best,Best,Choice,Choice) :-
        evaluate(Eq,This),
        asserta(this(This)),
        This > Best,
        !)).

try_sizes(Mod,Current,Eq,_,Best,_,Choice) :-
        retract(this(Bsf)),
        Next is 2*Current,
        !,
        try_sizes(Mod,Next,Eq,Bsf,Best,Current,Choice).

max_change(8).
```

```
process_chs(Env,Chs) :-
        is_processed(Chs),      % don't want to reprocess
        !.

process_chs(Env,Chs) :-
        is_primitive(Chs),
        process_primitive(Chs,Env),
        !.

process_chs(Env,Chs) :-
        subcells(Chs,cells(Subcells)),
        apply_to_each(process_chs,[Chs|Env],Subcells),
        all_max_delays(Chs,Subcells).

is_primitive(Chs) :- is_net(Chs).

% processing only one signal in a chs means that we don't need to
% process *all* the subcells, just the ones in which the signal is
% an output.
process_signal_in_chs(Env,Sig,Chs,Signal_entry) :-
        subcells(Chs,cells(Subcells)),
        relevant_cells(Subcells,Sig,Relevant),
        apply_to_each(process_chs,[Chs|Env],Relevant),
        find_signal_entry(Sig,Chs,Signal_entry),
        max_delay(Relevant,Signal_entry).

get_signal_entry(Sig,Chs,Entry) :-
        signals(Chs,Signals),
        assoc(Sig,Signals,Entry).

relevant_cells([],_,[]).
relevant_cells([C|Cs],Sig,[C|Newcs]) :-
        is_output(Sig,C),
        !,
        relevant_cells(Cs,Sig,Newcs).
relevant_cells([C|Cs],Sig,Newcs) :-
        relevant_cells(Cs,Sig,Newcs).

% the subcells are done, and so each output will have a delay for several
% of the subcells.  find the max.
all_max_delays(Chs,Subcells) :-
        signals(Chs,Signals),
        apply_to_each(one_max_delay,Subcells,Signals).

% for a particular signal, collect 'em all
one_max_delay(Subcells,Signal_entry) :-
        max_delay(Subcells,Signal_entry).

max_delay(Subcells,Signal_entry) :-
```

```
        signal_name(Signal_entry,Sig),
        find_max_delay(Subcells,Sig,Signal_entry).

set_signal_delay(sig(_,D,_),D).


make_dummy_signal_entry(sig(_,0,_)) :- !.

find_max_delay([],_,Dummy) :- make_dummy_signal_entry(Dummy).
find_max_delay([Cell|Cells],Sig,Entry) :-
        find_signal_entry(Sig,Cell,E0), % this will fail if Sig isn't in Cell
        !,
        find_max_delay(Cells,Sig,E1),
        bigger_delay(E0,E1,Entry).

find_max_delay([Cell|Cells],Sig,Entry) :-
        find_max_delay(Cells,Sig,Entry).

bigger_delay(D1,D2,D1) :-
        delay(D1,Delay1),
        delay(D2,Delay2),
        Delay1 > Delay2,
        !.
bigger_delay(D1,D2,D2).

delay(Signal,Delay) :- arg(2,Signal,Delay).

delay_in_cell(Chs,Sig,Delay,Info) :-
        make_signal_entry(Sig,Delay,Info,Entry),
        find_signal_entry(Sig,Chs,Entry).

make_signal_entry(Sig,Delay,Info,sig(Sig,Delay,Info)).

process_primitive(Chs,Env) :-
        set_input_delays(Chs,Env),
        attach_orders(Chs),
        output_signals(Chs,0),
        apply_to_each(primitive_delay,[Chs|Env],0).

set_input_delays(Chs,Env) :-
        inputs(Chs,I),
        apply_to_each(check_input,Env,I).

check_input(_,I) :-
        known_input_delay(I),
        !.

check_input(Env,I) :-
        signal_name(I,Name),
        delay_in_env(Env,Name,Delay_entry),
```

```
        set_input_delay(I,Delay_entry).

known_input_delay(in(_,Delay,_)) :-
        number(Delay).

set_input_delay(in(_,Delay,Info),sig(_,Delay,Info)).

% delay in env -- make sure a given signal has a known delay
delay_in_env([Chs|Env],Sig,Delay) :-
        known_delay(Chs,Sig,Delay),     % it does already
        !.

delay_in_env([Chs|Env],Sig,Delay) :-
        is_input(Sig,Chs,Delay), % it doesn't but it's outside our current chs
        delay_in_env(Env,Sig,Delay).

delay_in_env([Chs|Env],Sig,Delay) :-
        process_signal_in_chs(Env,Sig,Chs,Delay).

known_delay(Chs,Signal,Entry) :-
        find_signal_entry(Signal,Chs,Entry),
        known_signal_delay(Entry).

known_signal_delay(sig(_,Delay,_)) :-
        \+ var(Delay).

find_signal_entry(Name,Chs,Entry) :-
        signals(Chs,Signals),
        assoc(Name,Signals,Entry).


is_processed(Chs) :-
        signals(Chs,Signals),
        apply_to_each(known_signal_delay,Signals).
```

```
% path pre-processing


%  make_signals collects all the signal names and puts in slots in the outputs
% for the delays

make_signals(Chs,[O]) :-
        is_net(Chs),
        !,
        inputs(Chs,I),
        add_delays_to_inputs(I,NewI),
        output_signals(Chs,[O]),                    % only one output for a net
        add_delays_to_outputs([O],[NewO]),
        signals(Chs,[NewO|NewI]).

make_signals(Chs,O) :-
        subcells(Chs,cells(Cs)),
        map(make_signals,Cs,Subcell_outputs),
        flatten(Subcell_outputs,Temp),
        remove_dupes(Temp,[],O),
        inputs(Chs,I),
        add_delays_to_inputs(I,NewI),
        add_delays_to_outputs(O,NewO),
        append(NewI,NewO,Sigs),
        signals(Chs,Sigs).

make_outputs(Env,Chs) :-
        outputs(Chs,Outputs),
        apply_to_each(symbolic_terminal_capacitance,Env,Outputs),
        (is_primitive(Chs) -> true;
                subcells(Chs,cells(Sub)),
                apply_to_each(make_outputs,[Chs|Env],Sub)).


add_delays_to_inputs([],[]).
add_delays_to_inputs([in(Name,Delay,Info)|Is],[sig(Name,Delay,Info)|Xs]) :-
        add_delays_to_inputs(Is,Xs).
add_delays_to_inputs([in(Name)|Is],[sig(Name,_,_)|Xs]) :-
        add_delays_to_inputs(Is,Xs).

add_delays_to_outputs([],[]).
add_delays_to_outputs([O|Os],[sig(O,_,_)|Xs]) :-
        add_delays_to_outputs(Os,Xs).


make_structure(chs(Name,Inputs,Outputs,Subcells),C) :-
        map(make_structure,Subcells,Newsubs),
        C =.. [chs,Name,Inputs,Ouputs,Newsubs,_,_],
        make_signals(C,_),
        paths_in_env(C,_).
```

```
make_paths(C,[P]) :-
        is_net(C),
        !,
        subcells(C,Net),
        source(C,S),
        drain(C,D),
        P =.. [path,S,D,[[Net]]],
        paths(C,[P]).

make_paths(C,P) :-
        subcells(C,cells(Subcells)),
        length(Subcells,L),
        maplist(make_paths,Subcells,Subpaths),
        flatten(Subpaths,P1),
        make_all_paths(L,P1,P2),
        input_signals(C,I),
        select_input(P2,I,P3),
        paths(C,P3),
        output_signals(C,O),
        select_output(P3,O,P).

select_input([],_,[]).
select_input([P|Ps],I,[P|Rest]) :-
        P =.. [path,X,_,_],
        member(X,I),
        !,
        select_input(Ps,I,Rest).
select_input([P|Ps],I,Rest) :-
        select_input(Ps,I,Rest).

select_output([],_,[]).
select_output([P|Ps],O,[P|Rest]) :-
        P =.. [path,_,Y,_],
        member(Y,O),
        !,
        select_output(Ps,O,Rest).
select_output([P|Ps],O,Rest) :-
        select_output(Ps,O,Rest).


% make_all_paths(Length ,Short paths, all paths)

make_all_paths(0,Paths,Paths) :- !.
make_all_paths(N,Short_paths,Paths) :-
        cross(Short_paths,Short_paths,Somewhat_longer_paths),
        N1 is N // 2,
        make_all_paths(N1,Somewhat_longer_paths,Paths).

cross([],Short,Short).
cross([Pe|Pes],List,X) :-
```

```
        dot(Pe,List,P11),
        cross(Pes,List,P12),
        combine_path_lists(P11,P12,X).

combine_path_lists([],P,P).
combine_path_lists([path(X,Y,P)|Ps],Plist,Newp) :-
        add_new_path(X,Y,P,Plist,Temp),
        combine_path_lists(Ps,Temp,Newp).

dot(_,[],[]).
dot(path(X,Y,Paths),[path(Y,Z,P2) | Pes], Result) :-
        diddle(Paths,P2,Longer),
        dot(path(X,Y,Paths),Pes,More),
        add_new_path(X,Z,Longer,More,Result).
dot(path(X,Y,Paths),[path(Z,_,_) | Pes], Result) :-
        Y \== Z,
        dot(path(X,Y,Paths),Pes,Result).

% add_new_path(F,T,Path,Pathlist,Result)
add_new_path(F,T,P,[],[path(F,T,P)]).
add_new_path(F,T,P,[path(F,T,P)|X],[path(F,T,P)|X]) :- !.
add_new_path(F,T,P,[path(F,T,P1)|X],[path(F,T,Newp)|X]) :-
        put_paths_together(P,P1,Newp),
        !.
add_new_path(F,T,P,[X|Xs],[X|Y]) :-
        add_new_path(F,T,P,Xs,Y).

put_paths_together([],P,P).
put_paths_together([X|Xs],P,Newp) :-
        member(X,P),
        !,
        put_paths_together(Xs,P,Newp).
put_paths_together([X|Xs],P,Newp) :-
        put_paths_together(Xs,[X|P],Newp).

diddle([],_,[]).
diddle([P|Ps],Plist,Result) :-
        add_to_each(P,Plist,R1),
        diddle(Ps,Plist,R2),
        append(R1,R2,Result).

add_to_each(P,[],[]).
add_to_each(P,[L|Ls],[R|Rs]) :-
        append(P,L,R),
        add_to_each(P,Ls,Rs).
```

```
% attach_orders -- once all the inputs have known delays, the
%       gates can be ordered nicely.  I do this by sorting the inputs,
%       and then attaching the correct positions to each gate */

attach_orders(Chs) :-
        inputs(Chs,I),
        sort_inputs(I,Sorted),
        gates(Chs,Glist),
        attach_to_glist(Sorted,Glist,_).

% since I didn't feel like writing a sort routine, I just massaged things
% so that I could use keysort, the prepackagged routine.
sort_inputs(I,Sorted) :-
        make_sortable_inputs(I,Sort),
        keysort(Sort,Ugly),
        beautify(Ugly,Sorted,1).

% keysort wants its inputs in the form "Key-Value"
make_sortable_inputs([],[]).
make_sortable_inputs([In|Ins],[Key-In|Keys]) :-
        input_delay(In,Key),
        make_sortable_inputs(Ins,Keys).

input_delay(in(_,D,_),D).

beautify([],[],_).
beautify([_-In|Ins],[(Sig,Pos)|Rest],Pos) :-
        signal_name(In,Sig),
        P1 is Pos + 1,
        beautify(Ins,Rest,P1).

attach_to_glist(Sorted,G,Order) :-
        is_gate(G),
        signal_name(G,Sig),
        lookup_order(Sig,Sorted,Order).

attach_to_glist(Sorted,series(Gs),Order) :-
        map(order_gate,Sorted,Gs,Orders),
        max(Orders,Order).

attach_to_glist(Sorted,parallel(Gs),Order) :-
        map(attach_to_glist,Sorted,Gs,Orders),
        max(Orders,Order).

% this puts the order in the right field, as well as returning it
order_gate(Sorted,(G,Order),Order) :-
        attach_to_glist(Sorted,G,Order).

lookup_order(Sig,List,Order) :-
        assoc(Sig,List,(Sig,Order)).
```

```
order_list(Gs,Newgs) :-
        make_sortable_gates(Gs,L),
        keysort(L,New1),
        make_sortable_gates(Newgs,New1).

make_sortable_gates([],[]).
make_sortable_gates([G|Gs],[Key-G|Keys]) :-
        gate_order(G,Key),
        make_sortable_gates(Gs,Keys).

gate_order((_,0),0).

/* try_each_gate will go through a series gate to see where the source
        of the maximum delay is.  By now, the proper orders have been
        put on the gates, and they're sorted, so I just try each one in
        turn as the trigger.                                          */

/* syntax:
        try_each_gate(Chs,Newgs,Rin,[],Rlist,
                                    Cout,Clist,
                                    _,Rnet_sym,
                                    _,Cnet_sym,
                                    0,D,
                                    _,Trig).
*/
try_each_gate(_,[],_,_,_,_,_,
                Rsf,Rsf,
                Csf,Csf,
                Dsf,Dsf,
                Tsf,Tsf) :- !.
ifdef($fast_interpreter,(
% hacked interpreter
try_each_gate(Chs,[(Gate,_)|Gates],Rin,Rprev,Rg+R_int+R_rest,
                        Cout,Cg+C_int+C_rest,
                        Rsf,R_sym,
                        Csf,C_sym,
                        Dsf,D,
                        Tsf,T) :-
        Rtot is Rin - Rg,
        Ctot is Cout - Cg,
        % recursive call takes care of nested structures
        net_delay(Chs,Gate,Rtot,Rg_sym,Ctot,Cg_sym,Delay,Trig),
        Delay > Dsf,
        Cout_rest is Ctot - C_int,
        !,
        try_each_gate(Chs,Gates,Rin,Rprev + Rg + R_int, R_rest,
                        Cout_rest,C_rest,
                        Rprev + Rg_sym + R_int + R_rest, R_sym,
                        Cg_sym + C_int + C_rest, C_sym,
```

```
                        Delay,D,
                        Trig,T)),
% normal interpter
(try_each_gate(Chs,[(Gate,_)|Gates],Rin,Rprev,Rg+R_int+R_rest,
                        Cout,Cg+C_int+C_rest,
                        Rsf,R_sym,
                        Csf,C_sym,
                        Dsf,D,
                        Tsf,T) :-
        evaluate(Rg, R_correction),
        evaluate(Cg, C_correction),
        Rtot is Rin - R_correction,
        Ctot is Cout - C_correction,
        % recursive call takes care of nested structures
        net_delay(Chs,Gate,Rtot,Rg_sym,Ctot,Cg_sym,Delay,Trig),
        Delay > Dsf,
        Cout_rest is Ctot - C_int,
        !,
        try_each_gate(Chs,Gates,Rin,Rprev + Rg + R_int, R_rest,
                        Cout_rest,C_rest,
                        Rprev + Rg_sym + R_int + R_rest, R_sym,
                        Cg_sym + C_int + C_rest, C_sym,
                        Delay,D,
                        Trig,T))).

try_each_gate(Chs,[(Gate,_)|Gates],Rin,Rprev,Rg+R_int+R_rest,
                        Cout,Cg+C_int+C_rest,
                        Rsf,R_sym,
                        Csf,C_sym,
                        Dsf,D,
                        Tsf,T) :-
        try_each_gate(Chs,Gates,Rin,Rprev + Rg + R_int,R_rest,
                                Cout,C_rest,
                                Rsf,R_sym,
                                Csf,C_sym,
                                Dsf,D,
                                Tsf,T).

/* syntax:
        do_par_gates(Chs,Gs,Rin,Cout,
                                Rsf,Rnet_sym,
                                Csf,Cnet_sym,
                                Dsf,D,
                                Tsf,Trig).
*/
do_par_gates(_,[],_,_,Rsf,Rsf,Ctot,Ctot,Dsf,Dsf,Trig,Trig).
do_par_gates(Chs,[G|Gs],Rin,Cout,Rsf,Rtot,Csf,Ctot,Dsf,D,Tsf,T) :-
        net_delay(Chs,G,Rin,Rnew,Cout,Cnew,Delay,Trig),
        Delay > Dsf,
        !,
```

```
        do_par_gates(Chs,Gs,Rin,Cout,Rnew,Rtot,Cnew,Ctot,Delay,D,Trig,T).
do_par_gates(Chs,[G|Gs],Rin,Cout,Rsf,Rtot,Csf,Ctot,Dsf,D,Tsf,T) :-
        do_par_gates(Chs,Gs,Rin,Cout,Rsf,Rtot,Csf,Ctot,Dsf,D,Tsf,T).
```

```
% individual net delay in a hierarchical environment.

primitive_delay([Chs|Env],Sig) :-
        is_net(Chs),                    % only case we handle so far
        subcells(Chs,t(S,G,D)),
        paths_from_input(Env,S,Paths),
        terminal_capacitance(Chs,D,Cout_sym,Cout_num),
        max_delay_in_net(Chs,Paths,G,Cout_num,0,[],Delay,Trig),
        make_info_rec(Trig,Cout_sym,Chs,Info),
        delay_in_cell(Chs,Sig,Delay,Info).

make_info_rec(info(T,R,C),Cout_sym,Chs,info(T,R,C+Cout_sym,Chs)).

max_delay_in_net(_,[],_,_,Delay,Info,Delay,Info).
max_delay_in_net(Chs,[P|Ps],G,Ct,Dsf,Isf,Delay,Info) :-
        resistance(P,Rin_sym,Rin_num),
        net_delay(Chs,G,Rin_num,Rnet_sym,Ct,C_sym,D,Trig),
        D > Dsf,
        !,
        I =.. [info,Trig,Rin_sym+Rnet_sym,C_sym],
        max_delay_in_net(Chs,Ps,G,Ct,D,I,Delay,Info).

max_delay_in_net(Chs,[P|Ps],G,Ct,Dsf,Isf,Delay,Info) :-
        max_delay_in_net(Chs,Ps,G,Ct,Dsf,Isf,Delay,Info).

glist_resistance(G,R) :-
        combine(gate_resistance,G,R).


net_delay(Chs,G,Rin,Rnet_sym,Cout,Cnet_sym,D,G) :-
        is_gate(G),                     % note that the gate resistance is in Rin
                                        % (for now)
        gate_capacitance(G,Cnet_sym,Cnet_num),
        gate_resistance(G,Rnet_sym,Rnet_num),
        signal_name(G,Sig),
        find_signal_delay(Chs,Sig,Trigger_delay),
        D is Trigger_delay + (Rin + Rnet_num) * (Cout + Cnet_num).

net_delay(Chs,series(Gs),Rin,Rnet_sym,Cout,Cnet_sym,D,Trig) :-
        order_list(Gs,Newgs),
        combine(gate_resistance,series(Newgs),Rlist,R_num),
%       simplify(Rlist,Simp_rlist),
        Rtot is Rin + R_num,
        combine(gate_capacitance,series(Newgs),Clist,C_num),
%       simplify(Clist,Simp_clist),
        Ctot is Cout + C_num,
        try_each_gate(Chs,Newgs,Rtot,0,Rlist,
                                Ctot,Clist,
                                _,Rnet_sym,
                                _,Cnet_sym,
```

```
                                        O,D,
                                        _,Trig).

net_delay(Chs,parallel(Gs),Rin,Rnet_sym,Cout,Cnet_sym,D,Trig) :-
        do_par_gates(Chs,Gs,Rin,Cout,
                                        _,Rnet_sym,
                                        _,Cnet_sym,
                                        O,D,
                                        _,Trig).


find_signal_delay(Chs,Sig,Delay) :-
        find_signal_entry(Sig,Chs,Entry),
        delay(Entry,Delay).

is_gate(gt(_,_,_)).
gate_signals(gt(X,_,_),[X]).
gate_signals(Glist,[]) :-
        Glist =.. [F,[]].
gate_signals(Glist,Sigs) :-
        Glist =.. [F,[G|Gs]],
        gate_signals(G,X),
        Newglist =.. [F,Gs],
        gate_signals(Newglist,Y),
        append(X,Y,Sigs).


path_in_glist(Gate,Signals,Prev,Gate,Gate) :-
        Gate =.. [gt,Prev,Type,Size],
        \+ member(Prev,Signals).

path_in_glist(series([(G,Order)|Gs]),Signals,Prev,series([(Newg,Order)|Gs]),G
        path_in_glist(G,Signals,Prev,Newg,Gate).

path_in_glist(series([G|Gs]),Signals,Prev,series([G|Newgs]),Gate) :-
        path_in_glist(series(Gs),Signals,Prev,series(Newgs),Gate).

path_in_glist(parallel([G|Gs]),Signals,Prev,P,Gate) :-
        path_in_glist(G,Signals,Prev,P,Gate).

path_in_glist(parallel([G|Gs]),Signals,Prev,P,Gate) :-
        path_in_glist(parallel(Gs),Signals,Prev,P,Gate).

paths_from_input([Chs|Env],Sig,Paths) :-
        paths_to(Sig,Chs,P2),
        continuation(Env,P2,Paths).

paths_to(Sig,Chs,[[]]) :-
        is_input(Sig,Chs),
        !.
```

```
paths_to(Sig,Chs,Flatp) :-
        paths(Chs,Path_list),
        choose_paths_to(Sig,Path_list,P),
        flatten(P,Flatp).

choose_paths_to(_,[],[]).
choose_paths_to(Sig,[Pathrec|Ps],[P|Rest]) :-
        is_path_to(Sig,Pathrec),
        !,
        strip(Pathrec,P),
        choose_paths_to(Sig,Ps,Rest).
choose_paths_to(Sig,[P|Ps],Rest) :-
        choose_paths_to(Sig,Ps,Rest).

strip(path(_,_,P),P).

% continuation takes the existing path list and moves up the environment
% stack until it finally makes it to an input to the whole kitten kaboodle
continuation([],Paths,Paths).
continuation([Chs|Env],Psf,Paths) :-
        print('made it'),nl,
        map(extend,Chs,Psf,Temp),
        flatten(Temp,Newpsf),
        continuation(Env,Newpsf,Paths).

% extend takes a path, which doesn't yet terminate at the inputs of the chs,
% and extends it so that it does terminate at an input
extend(Chs,Path,[Path]) :-
        input_terminal_of_path(Path,S),
        is_input(S,Chs),
        !.

extend(Chs,Path,Path_list) :-
        input_terminal_of_path(Path,S),
        paths_to(S,Chs,P),
        map(add_to_end,Path,P,Path_list).

% bleah, but this is due to the restrictions of map
add_to_end(path(_,Y,Plist1),path(X,_,Plist2),path(X,Y,Plist3)) :-
        diddle(Plist1,Plist2,Plist3).

input_terminal_of_path(path(X,_,_),X).
is_path_to(Sig,path(_,Sig,_)).
```

```
% take a chs with the signals filled in, and find a critical path

find_critical_path(Chs,Cp) :-
        signals(Chs,Signals),
        max_output_delay(Signals,Cp).

max_output_delay([],Entry) :-
        make_dummy_signal_entry(Entry).

max_output_delay([Sig|Sigs],Entry) :-
        max_output_delay(Sigs,E1),
        bigger_delay(Sig,E1,Entry).


% find a signals predecessor in the critical path
prev_cp_entry(Info,Next) :-
        trigger(Info,Trig),
        trig_chs(Info,Chs),
        signal_name(Trig,Sig),
        find_signal_entry(Sig,Chs,Next).

prev_cp_entry(Sig,Next) :-
        info(Sig,Info),
        prev_cp_entry(Info,Next).
% find the delay equations on a path

delay_equation(Cp,Input_delay) :-
        no_predecessor(Cp),
        !,
        delay(Cp,Input_delay).

delay_equation(Cp,R * C + Rest) :-
        info(Cp,Info),
        symbolic_r(Info,R),
        symbolic_c(Info,C),
        prev_cp_entry(Info,Next),
        delay_equation(Next,Rest).


symbolic_r(I,R) :-
        arg(2,I,R).

symbolic_c(I,C) :-
        arg(3,I,C).
```

```
% this is the electrical model, measuring resistance and capacitance
% for a gate or path of gates
resistance([],0,0).
resistance([t(_,Glist,_)|Ts],R1_sym+R2_sym,R) :-
        resistance(Ts,R1_sym,R1_num),
        combine(gate_resistance,Glist,R2_sym,R2_num),
        R is R1_num+R2_num,
        !.

capacitance([],0,0).
capacitance([t(_,Glist,_)|Ts],C1_sym+C2_sym,C) :-
        capacitance(Ts,C1_sym,C1_num),
        combine(gate_capacitance,Glist,C2_sym,C2_num),
        C is C1_num+C2_num.

ifdef($fast_interpreter,(
gate_resistance(gt(_,T,S),Rg / S,Rnum) :-
        gate_resistance(T,Rg),
        Rnum is Rg/S ),
% otherwise
(gate_resistance(gt(_,T,S),Rg / S,Rnum) :-
        gate_resistance(T,Rg),
        evaluate(Rg/S,Rnum))).

ifdef($fast_interpreter,(
% hacked interpreter
gate_capacitance(gt(_,Type,S),S * Ctot,Cnum) :-
        gate_channel_cap(Type,C1),
        gate_drain_cap(Type,C2),
        Ctot is C1 + 2*C2,
        Cnum is S * Ctot),
%usual interpreter
(gate_capacitance(gt(_,Type,S),S * Ctot,Cnum) :-
        gate_channel_cap(Type,C1),
        gate_drain_cap(Type,C2),
        Ctot is C1 + 2*C2,
        evaluate(S * Ctot,Cnum))).

% these are just reasonable constants
gate_resistance(n,8).
gate_resistance(p,10).
gate_resistance(interconnect,0).

gate_capacitance(interconnect,1).
gate_channel_cap(n,4).
gate_channel_cap(p,4).
gate_drain_cap(n,2.5).
gate_drain_cap(p,2.5).

interconnect_resistance(0).
```

```
interconnect_capacitance(1).


% preprocess: get the output capacitance symbolicly at the beginning

ifdef($fast_interpreter,(
%hacked interpreter
terminal_capacitance(Chs,Sig,C_sym,C_num) :-
        outputs(Chs,O),
        assoc(Sig,O,Entry),
        symbolic_cap(Entry,C_sym),
        C_num is C_sym),
% normal interpreter
(terminal_capacitance(Chs,Sig,C_sym,C_num) :-
        outputs(Chs,O),
        assoc(Sig,O,Entry),
        symbolic_cap(Entry,C_sym),
        evaluate(C_sym,C_num))).

symbolic_terminal_capacitance([],Output) :-
        numeric_output_cap(Output,C),
        symbolic_cap(Output,C).

symbolic_terminal_capacitance([Env|_],Output) :-
        signal_name(Output,Sig),
        output_capacitance(Env,Sig,Ocap),
        subcells(Env,cells(Cellist)),
        sum_gate_capacitance(Sig,Cellist,Gcap),
        simplify(Ocap+Gcap,Cap),
        symbolic_cap(Output,Cap).

output_capacitance(Env,Sig,C) :-
        outputs(Env,Os),
        assoc(Sig,Os,Output),
        numeric_output_cap(Output,C),
        !.
output_capacitance(_,_,0). %because we don't want to fail if it's not an output

sum_gate_capacitance(_,[],0).
sum_gate_capacitance(Sig,[Sc|Scs],C1 + C2) :-
        subcell_capacitance(Sig,Sc,C1),
        sum_gate_capacitance(Sig,Scs,C2).

subcell_capacitance(Sig,Chs,C) :-
        is_input(Sig,Chs),
        is_net(Chs),            % only case we handle so far
        !,
        gates(Chs,G),
        signal_gates_cap(Sig,G,C).
subcell_capacitance(_,_,0).     % if it's not an input of that cell
```

```
signal_gates_cap(Sig,(G,_),C) :-        % ugliness for series xsistors
        signal_gates_cap(Sig,G,C),!.

signal_gates_cap(Sig,gt(Sig,Type,Size),C1 + C2) :-
        symbolic_gate_channel_cap(gt(Sig,Type,Size),C1),
        gate_capacitance(interconnect,C2).

signal_gates_cap(Sig,gt(Other,_,_),0) :-
        Sig \== Other.

signal_gates_cap(Sig,Glist,0) :-
        Glist =.. [X,[]].

signal_gates_cap(Sig,Glist,C1 + C2) :-
        Glist =.. [X, [G|Gs]],
        signal_gates_cap(Sig,G,C1),
        Newglist =.. [X, Gs],
        signal_gates_cap(Sig,Newglist,C2).

symbolic_gate_channel_cap(gt(_,T,S),C * S) :-
        gate_channel_cap(T,C).

% combine(Functor,Glist,Symbolic,Numeric)  -- used to sum or take max
% of resistance or capacitance.  the combining rules are the same

combine(Functor,gt(Sig,Type,Size),Rsym,Rnum) :-
        P =.. [Functor,gt(Sig,Type,Size),Rsym,Rnum],
        call(P).

combine(_,series([]),0,0).

combine(Functor,series([(G,_)|Gs]),R1_sym+R_int+R2_sym,Rnum) :-
        combine(Functor,G,R1_sym,R1_num),
        combine(Functor,series(Gs),R2_sym,R2_num),
        P =.. [Functor,interconnect,R_int],
        call(P),
        Rnum is R1_num+R2_num+R_int.

combine(_,parallel([]),0,0).

combine(Functor,parallel([G|Gs]),Rmax_sym,Rnum) :-
        combine(Functor,G,R1_sym,R1_num),
        combine(Functor,parallel(Gs),R2_sym,R2_num),
        (R1_num > R2_num ->
                Rmax_sym = R1_sym,
                Rnum is R1_num;
                Rmax_sym = R2_sym,
                Rnum is R2_num).
```

```
% utils -- various (non-problem-dependent) utilities

% assoc(X,Y,Z) : Z is the member of Y with X as its first element
assoc(X,[Y|_],Y) :- arg(1,Y,X).
assoc(X,[_|Ys],Z) :- assoc(X,Ys,Z).

member(X,[X|_]).
member(X,[_|L]) :- member(X,L).

% var_member can't use unify, which would bind the variables by mistake
var_member(Var,[Var1|_]) :- Var == Var1, !.
var_member(Var,[_|Vars]) :- var_member(Var,Vars),!.

max(A,B,A) :- A >= B.
max(A,B,B) :- B > A.
min(A,B,A) :- A =< B.
min(A,B,B) :- B < A.

cpu :-
        N is cputime,
        print('cpu time is '),
        print(N),
        nl,
        !.

print_list([]).
print_list([L|Ls]) :-
        print(L),
        nl,
        print_list(Ls).

sum([],0).
ifdef($fast_interpreter,
(sum([X|Xs],Tot) :-
        sum(Xs,Sub),
        Tot is X + Sub),
% otherwise
(sum([X|Xs],Tot) :-
        (number(X),
        sum(Xs,Sub),
        Tot is X + Sub;
        var(X),
        sum(Xs,Sub),
        Tot is Sub))).

max([],0).
max([X|Xs],Max) :-
        max(Xs,M),
        max(X,M,Max).
```

```
map(_,[],[]).
map(Functor,[L|Ls],[New1|New1s]) :-
        P =.. [Functor,L,New1],
        call(P),
        !,
        map(Functor,Ls,New1s).

map(_,_,[],[]).
map(Functor,Args,[L|Ls],[New1|New1s]) :-
        P =.. [Functor,Args,L,New1],
        call(P),
        !,
        map(Functor,Args,Ls,New1s).

range(Lo,Lo,_).
range(Lo,N,Hi) :-
        New is Lo+1,
        New =< Hi,
        !,
        range(New,N,Hi).

% another thing that should be an operator, by the way
abs(X,X) :- X > 0,!.
abs(X,Y) :- Y is -X,!.

% flatten a list (i.e., put sublists into the main list)

flatten([],[]).
flatten([X|Xs],Res) :-
   flatten(Xs,Temp),
   append(X,Temp,Res).

% a particularly ugly n-squared algorithm for removing duplicates from the list
remove_dupes([],L,L).
remove_dupes([X|Xs],L,Result) :-
        member(X,L),
        !,
        remove_dupes(Xs,L,Result).

remove_dupes([X|Xs],L,Result) :-
        remove_dupes(Xs,[X|L],Result).

append([],Result,Result).
append([X|Xs],Y,[X|Temp]) :-
        append(Xs,Y,Temp).

apply_to_each(F,[]).
apply_to_each(F,[L|Ls]) :-
        P =.. [F,L],
        call(P),
```

```
        apply_to_each(F,Ls).

apply_to_each(F,Arg,[]).
apply_to_each(F,Arg,[L|Ls]) :-
        P =.. [F,Arg,L],
        call(P),
        apply_to_each(F,Arg,Ls).

ff :- put(12).
```

```
/* symbolic mathematics: evaluating, simplifying, and taking derivatives
        of equations.  Also, collecting all the variables in a given equation *

% evaluate a symbolic equation.

evaluate(X,X) :-
        number(X),
        !.

evaluate(X,K) :-
        var(X),
        !,
        minimum_gate_size(K).

evaluate(S+Mod,Z) :-
        (var(Mod) ->
            evaluate(S,Z);
            evaluate(S,R1),
            evaluate(Mod,R2),
            Z is R1+R2),
        !.

evaluate(X,Result) :-
        X =.. [Op,Arg1,Arg2],
        evaluate(Arg1,R1),
        evaluate(Arg2,R2),
        Y =.. [Op,R1,R2],
        Result is Y,
        !.


% equation simplifier.  Doesn't even worry about the distributive law:
% speed is the key.  The main purpose of simplify is to get rid of zeros
% being added in.

simplify(Exp,Exp) :-
        number(Exp),
        !.

simplify(Exp,Exp) :-
        var(Exp),
        !.

simplify(Exp,Res) :-
        Exp =.. [Op,Arg1,Arg2],
        simplify(Arg1,New1),
        simplify(Arg2,New2),
        combine_simplified_terms(Op,New1,New2,Res).

% combine_simplifiedd_terms is where the zeroes (and other constants) are
```

```
% removed if possible

combine_simplified_terms(Op,Identity,Res,Res) :-
        left_identity(Op,Id),
        Id == Identity,
        !.
combine_simplified_terms(Op,Res,Identity,Res) :-
        right_identity(Op,Id),
        Id == Identity,
        !.
combine_simplified_terms('*',Arg1,Arg2,0) :-
        (Arg1 == 0 ;     % could be more ggeneral, with a "nullity" like
         Arg2 == 0),     % "identity", but why bother?
        !.
combine_simplified_terms('/',Arg1,_,0) :-
        Arg1 == 0,
        !.
combine_simplified_terms(Op,Arg1,Arg2,Res) :-
        number(Arg1),
        number(Arg2),
        !,
        P =.. [Op,Arg1,Arg2],
        Res is P.
combine_simplified_terms(Op,Arg1,Arg2,Res) :-
        Res =.. [Op,Arg1,Arg2].

left_identity('+',0).
left_identity('*',1).
right_identity('+',0).
right_identity('-',0).
right_identity('*',1).
right_identity('/',1).


% symbolic deriviatives.  This could be data-directed, storing the
% proper information for each operator, but that wouldn't help --
% an attempt would still need to be made to unify with each.  This is
% very slow and special-purpose right now; however, the equations will
% not have any weird operators in them

% some attempt is made to avoid what the symbolic math people call
% "intermediate expression swell" by being somewhat intelligent.  That's
% why all the independence checks.  However, with a large equation, these
% checks wind up taking a lot of time

deriv(Eq,Var,1) :- Eq == Var,!.
deriv(Eq,_,0) :- number(Eq),!.
deriv(Eq,Var,0) :- var(Eq),Eq \== Var,!.
deriv(U+V,X,Dv) :-
        independent(U,X),
```

```
        deriv(V,X,Dv),
        !.
deriv(U+V,X,Du+Dv) :-
        deriv(U,X,Du),
        deriv(V,X,Dv),
        !.
deriv(U*V,X,Res) :-
        deriv(V,X,Dv),
        (independent(U,X) ->
                Res = U * Dv;
                deriv(U,X,Du),
                Res = Du*V + Dv*U),
        !.
deriv(U/V,X,Res) :-
        independent(V,X),
        (number(U) -> Res = 0;
                      deriv(U,X,Du),
                      Res = Du/V),
        !.
deriv(U/V,X,MinU*Dv/(V*V)) :-
        number(U),
        MinU is -U,
        deriv(V,X,Dv),
        !.
deriv(U/V,X,(V*Du-U*Dv)/(V*V)) :-
        deriv(U,X,Du),
        deriv(V,X,Dv),
        !.
deriv(U-V,X,Du-Dv) :-
        deriv(U,X,Du),
        deriv(V,X,Dv),
        !.

% take all the partial derivatives of the equation
partials(Eq,Vars,Partials) :-
        map(partial,Eq,Vars,Partials).

% should be
%       map(deriv,Eq,Vars,Big_partials),
%       map(simplify,Big_partials,Partials).
% but since there's no TRO it won't work like that.

partial(Eq,Var,Partial) :-
        deriv(Eq,Var,Temp),
        simplify(Temp,Temp_partial),
        % keep the variables right
        asserta($this_partial(Eq,Var,Temp_partial)),
        fail.
partial(Eq,Var,Partial) :-
        retract($this_partial(_,_,Partial)),
```

```
        !.

% check if an equation is independent of a variable -- i.e., if that variable
% does NOT appear in the equation

independent(Exp,Var) :-
        number(Exp),
        !.
independent(Exp,Var) :-
        var(Exp),
        Exp \== Var,
        !.

independent(Exp,Var) :-
        \+ var(Exp),
        Exp =.. [_,Arg1,Arg2],
        independent(Arg1,Var),
        independent(Arg2,Var),
        !.


% collect all the variables of an equation

collect_vars(Eq,Vsf,Vsf) :- number(Eq),!.
collect_vars(Eq,Vsf,Vsf) :- var(Eq),var_member(Eq,Vsf),!.
collect_vars(Eq,Vsf,[Eq|Vsf]) :- var(Eq),!.
collect_vars(Eq,Vsf,Vars) :-
        Eq =.. [_,Arg1,Arg2],
        collect_vars(Arg1,Vsf,V1),
        collect_vars(Arg2,V1,Vars),
        !.
```

```
/* STRUCTS: data structure access functions
        of course, these should be macros, but you can't do that in cprolog

        There are two kinds of chs's: the initial 4-field chs, and the
        local 6-field chs.  the 4-field chs looks like

        chs(
                Name
                inputs([Inputlist])
                outputs([Outputlist])
                Subcells)

        where Subcells is either

                cells([Subcelllist])
        or      t(Source,Gatelist,Drain)

        the 6-field chs is a slight extension of that

        chs(
                Name
                inputs([Inputlist])
                outputs([Outputlist])
                Subcells
                [Pathlist]
                [Signal_list])                                          */


chs_name(Chs,N) :-
        arg(1,Chs,N).

inputs(Chs,X) :-
        arg(2,Chs,inputs(X)).

outputs(Chs,X) :-
        arg(3,Chs,outputs(X)).

subcells(Chs,X) :-
        arg(4,Chs,X).

paths(Env,P) :-
        arg(5,Env,P).

signals(Chs,Sigs) :-
        arg(6,Chs,Sigs).


/* "is" procedures are used both to test and generate ...
```

```
        is_input(X,Chs) will instantiate X in turn to each input of the chs
        or, if X is instantiated, fail if it is not an input                */

is_input(X,Chs) :-
        inputs(Chs,I),
        assoc(X,I,_).

is_output(X,Chs) :-
        outputs(Chs,O),
        assoc(X,O,_).

is_subcell(t(S,G,D),Chs) :-
        subcells(Chs,t(S,G,D)).

is_subcell(C,Chs) :-
        subcells(Chs,cells(Cs)),
        member(C,Cs).

is_signal(Chs,X) :-
        signals(Chs,Sigs),
        member(X,Sigs).

is_net(Chs) :-
        subcells(Chs,t(_,_,_)).

/* outputs are of the form
        out(Signal,Cap,Symbolic_capacitance)
    inputs are
        in(Signal,Delay)                        */

signal_name(O,Sig) :-
        arg(1,O,Sig).

output_signals(Chs,X) :-
        outputs(Chs,Y),
        map(signal_name,Y,X).

input_signals(Chs,X) :-
        inputs(Chs,Y),
        map(signal_name,Y,X).

input_delay(Input,Delay) :-
        arg(2,Input,Delay).

numeric_output_cap(Output,C) :- arg(2,Output,C).
symbolic_cap(Output,Eq) :- arg(3,Output,Eq).

% find the input delay of a signal in a Chs
input_delay(Signal,Chs,Delay) :-
        inputs(Chs,I),
```

```
        assoc(Signal,I,Temp),
        input_delay(Temp,Delay).

source(Chs,S) :-
        subcells(Chs,t(S,_,_)).

drain(Chs,D) :-
        subcells(Chs,t(_,_,D)).

gates(Chs,G) :-
        subcells(Chs,t(_,G,_)).


% a delay entry looks like
%       delay_entry(Signal,Delay,Prev,Path)

delay(Signal,Delays,D) :-
        delay_entry(Delays,Signal,D,_,_).

delay_entry(Delays,Signal,Delay,Prev,Path) :-
        assoc(Signal,Delays,delay_entry(Signal,Delay,Prev,Path)).


in_gate(X,T) :-
        in_gate(X,T,_),
        !.

in_gate(X,t(S,Glist,D),G) :-
        Glist =.. [_,L],
        in_glist(X,L,G).

in_glist(X,[gt(X,Y,Z)|_],gt(X,Y,Z)) :- !.
in_glist(X,[Glist|_],G) :-
        Glist =.. [_,L],        % note that this won't match gt(Sig,Type)
        in_glist(X,L,G),
        !.
in_glist(X,[Y|Ys],G) :-
        in_glist(X,Ys,G).


gate_size(S,S) :-
        number(S).              % ! handle both integers and reals

gate_size(S,K) :-
        var(S),
        !,
        minimum_gate_size(K).

gate_size(S+Mod,X) :-
        var(Mod),
```

```
        gate_size(S,X).

gate_size(S+Mod,X) :-
        number(Mod),
        gate_size(S,Size),
        X is Size+Mod.

minimum_gate_size(2).

info(sig(_,_,Info),Info).
trigger(Info_rec,Trigger) :- arg(1,Info_rec,Trigger).
trig_chs(Info_rec,Chs) :- arg(4,Info_rec,Chs).
sig_delay(sig(_,Delay,_),Delay).
```

```
% user print functions

portray(chs(N,I,O,cells(C))) :-
        print('chs '),print(N),nl,
        print('inputs = '),print(I),nl,
        print('outputs = '),print(O),nl,
        print('subcells are '),print_list(C).
portray(chs(N,I,O,t(S,G,D))) :-
        print(t(S,G,D)).


% the six-field local chs also needs a portray function
portray(chs(N,I,O,cells(C),P,S)) :-
        print('chs '),print(N),nl,
        print('inputs = '),print(I),nl,
        print('outputs = '),print(O),nl,
        (var(P) -> true;print('paths are '),print_list(P)),
        (var(S) -> true;print('signals are '),nl,print_list(S)),
        print('subcells are '),print_list(C).

portray(chs(N,I,O,t(S,G,D),_,_)) :-
        print('net '),print(N),nl,
        print('inputs = '),print(I),nl,
        print('outputs= '),print(O),nl,
        print(t(S,G,D)).

print_cp(Entry) :-
        print('Node '),
        signal_name(Entry,Name),
        print(Name),
        print(' is driven at '),
        sig_delay(Entry,Delay),
        print(Delay),
        (info(Entry,[]) -> nl;
                info(Entry,Info),
                trigger(Info,Trig),
                print(' via '),print(Trig),print(' after '),nl,
                prev_cp_entry(Info,Next),
                print_cp(Next) ).
```

```
% make chs: take a standard every-day chs, and make it into a local
%       data structure with all the appropriate fields.  Essentially,
%       what needs to be done is to add the fields for paths and signals,
%       and the extra delay and info fields in the inputs */

make_lds(Chs,Lds) :-
        make_struct(Lds),
        chs_name(Chs,Name),
        chs_name(Lds,Name),
        inputs(Chs,Inputs),
        map(add_input_fields,Inputs,Newin),
        inputs(Lds,Newin),
        outputs(Chs,Outputs),
        map(add_output_fields,Outputs,Newouts),
        outputs(Lds,Newouts),
        (is_primitive(Chs) ->
            (subcells(Chs,Net),
             subcells(Lds,Net));
            % otherwise
            subcells(Chs,cells(Cells)),
            map(make_lds,Cells,Newsubs),
            subcells(Lds,cells(Newsubs))),
        !.

make_struct(chs(_,_,_,_,_,_)).

% who knows how many fields there will be there to begin with?
%       It could be 1, 2, or 3.  Make it 3 in the lds.
add_input_fields(in(Name,Delay,Info),in(Name,Delay,Info)).
% if it has a delay there, that can only mean it's an input to the
%       whole circuit.
add_input_fields(in(Name,Delay),in(Name,Delay,[])).
add_input_fields(in(Name),in(Name,_,_)).


add_output_fields(out(Name,Cap),out(Name,Cap,_)).
add_output_fields(out(Name),out(Name,_,_)).

pp(L) :-
        chs(C),
        make_lds(C,L),
        make_signals(L,_),
        make_paths(L,_),
        make_outputs([],L).

chs(C) :- C =.. [chs,_,_,_,_,_],call(C).

% the badly-named "make_sizes" collects the sizes of all the primitives
% and crams them into a vector
```

```
make_sizes(L,S) :-
        is_primitive(L),
        !,
        primitive_sizes(L,S).

make_sizes(L,S) :-
        subcells(L,cells(C)),
        map(make_sizes,C,NestedS),
        flatten(NestedS,S),
        assert(sizes_to_try(S)).

primitive_sizes(L,S) :-
        is_net(L),
        net_glist(L,G),
        glist_sizes(G,S).

net_glist(C,G) :-
        subcells(C,t(_,G,_)).

% remember that a glist may be a series or parallel connection of gts
glist_sizes(gt(_,_,S),[S]) :- !.
glist_sizes(Glist,S) :-
        Glist =.. [_,Gates],
        map(glist_sizes,Gates,Sizes),
        flatten(Sizes,S).
% this handles the case of series gates with orders attached
glist_sizes((G,_),S) :- glist_sizes(G,S).

no_predecessor(Cp) :-
        info(Cp,[]),
        !.
```

```
% random number generation: I stole this.

%   File    : /usr/lib/prolog/random
%   Author  : R.A.O'Keefe
%   Updated: 27 October 83
%   Purpose: to provide a decent random number generator in C-Prolog.

%   This is algorithm AS 183 from Applied Statistics.  I also have a C
%   version.  It is really very good.  It is straightforward to make a
%   version which yields 15-bit random integers using only integer
%   arithmetic.

'$rstate'(27134, 9213, 17773).          %  initial state

getrand('$rstate'(X,Y,Z)) :-            %  return current state
        '$rstate'(X,Y,Z).


setrand('$rstate'(X,Y,Z)) :-
        integer(X), X > 0, X < 30269,
        integer(Y), Y > 0, Y < 30307,
        integer(Z), Z > 0, Z < 30323,
        retract('$rstate'(_,_,_)),
        asserta('$rstate'(X,Y,Z)), !.


%   random(R) binds R to a new random number in [0.0,1.0)

random(R) :-
        retract('$rstate'(A0,B0,C0)),
        A1 is (A0*171) mod 30269,
        B1 is (B0*172) mod 30307,
        C1 is (C0*170) mod 30323,
        asserta('$rstate'(A1,B1,C1)),
        T is (A1/30269.0) + (B1/30307.0) + (C1/30323.0),
        R is T-floor(T), !.


%   random_int(L, U, R) binds R to a random integer in [L,U)
%   when L and U are integers (note that U will NEVER be generated).

random_int(L, U, R) :-
        integer(L), integer(U),
        random(X), !,
        R is L+floor((U-L)*X).

%   random(L, U, R) binds R to a random real in [L,U)
%   when L and U are numbers (note that U will NEVER be generated).

random(L, U, R) :-
```

```
        number(L), number(U),
        random(X), !,
        R is L+((U-L)*X).
```

```
% minimize an equation -- set it below a given delay

%       minimize(Eq,Delay,Vars,Constraints,Derivs)

%       a variable is not allowed to be less than the corresponding constraint

minimize(Eq,Delay,Vars,Constraints,Derivs,Result) :-
        length(Vars,Length),
        min_init_configuration(Vars),
        repeat,
            get_configuration(Vars),
            Result is Eq,
            print('result is '),print(Result),nl,
            (Result < Delay ->
                true;
                min_new_configuration(Length,Vars,Constraints,Derivs),
                fail).

min_init_configuration(Vars) :-
        map(zero,Vars,New),
        asserta($min_config(New)),
        !.

zero(_,0).

get_configuration(Vars) :-
        retract($min_config(Vars)),
        print('New config is '),print(Vars),nl,
        !.

min_new_configuration(Length,Vars,Constraints,Derivs) :-
        build_new_config(Length,Vars,Constraints,Derivs,New),
        asserta($min_config(New)),
        !.


build_new_config(Length,Vars,Constraints,Derivs,New) :-
        evaluate_derivs(Derivs,Num_derivs,0,Sum),
        Avg is Sum/Length,
        normalize(Avg,Vars,Derivs,Constraints,New).

evaluate_derivs([],[],Sum,Sum).
evaluate_derivs([D|Ds],[N|Ns],Ssf,Sum) :-
        N is D,
        Temp is -N,
        max(0,Temp,Est),
        New_sum is Est + Ssf,
        !,
        evaluate_derivs(Ds,Ns,New_sum,Sum).
```

```
normalize(Sum,[],[],[],[]).
normalize(Avg,[V|Vs],[D|Ds],[C|Cs],[N|Ns]) :-
        Desired is V - D/Avg,
        (Desired < C ->
                N is C;
                N is Desired),
        !,
        normalize(Avg,Vs,Ds,Cs,Ns).
```