Abstract Timing Verification for Synchronous Digital Systems

David E. Wallace

ABSTRACT

ATV, the Abstract Timing Verifier, is a program to perform static timing analysis of dependency graphs derived from logic designs, analyzing worst-case paths. Unlike other timing verifiers, ATV uses an abstract representation of time and delays that enables a user to choose the representation of time and delays most suitable to a particular analysis. Such representations include single numbers, ranges [min-max], and statistical descriptions (mean and standard deviation), or asymmetric rise/fall versions of all of these. The sophisticated user may develop new models and plug them in to the program.

This technical report consists of the main body of my dissertation of the same title. It describes the background of the Abstract Timing Model that ATV uses, several different timing models, implementation of the principle algorithm for clock phase length analysis of transparent latch designs, and results of using the program. Detailed information about how to use the program is available in the companion technical report, *User's Guide to ATV, an Abstract Timing Verifier*, which also appeared as the appendix to my dissertation.

Abstract Timing Verification for Synchronous Digital Systems

David Edward Wallace

Acknowledgments

I am grateful to the fellow members of the SOAR implementation teams, whose struggles inspired me to develop a better timing verifier, and whose discussions helped shape my initial ideas. Dave Patterson led the project, Joan Pendleton headed the nMOS implementation team and ran into all the timing problems ahead of us, and the fellow members of the original CMOS implementation team were B. K. Bose, Mark Hofmann, Peter Moore, and John Zapisek.

Rick Rudell, Peter Moore, Rick Spickelmier and David Harrison contributed many helpful discussions and suggestions on the interface with the Oct database and the VEM editor. Nick Weiner shared many helpful thoughts on various timing verification issues. Kanwar Singh was invaluable in providing the support needed to run the DES chip example through ATV and analyze the results.

Shing Kong and David Wood contributed several helpful discussions on what chip designers want from a timing verifier. Jim Larus and Gaetano Borriello were helpful sources of Lisp expertise, and helped me evaluate various object-oriented Lisp systems. Jim Larus and Rick Spickelmier both provided helpful expertise on Portable Common Loops.

Eli Messinger provided the source code to the sungrab graph layout program, and many helpful discussions on the internal data structures. Rick Spickelmier was equally helpful in providing source to the slide-maker and advice on RPC applications. Paul Hansen provided a working set of thesis formatting macros when I needed them.

Finding good simple examples that cleanly illustrate the core of an idea can be difficult; Tim Miller of Mentor Graphics Corporation suggested two examples that turned into Figure 4-7. My roommate, Davor Sutija, and his mother were a critical and appreciative lay audience for an explanation of my work that turned into the introduction to this report. The example of Figure 6-1 developed out of a discussion with Nick Weiner.

I appreciate the encouragement and support of my committee, Carlo Séquin, Richard Newton, and Mark Latham. I am particularly grateful to my advisor and chairman, Carlo Séquin, who persuaded me to come to Berkeley, and has provided constant encouragement, inspiration, ideas and guidance to me

throughout my stay here. Finally, I am extremely grateful to my fiancée, Jan Bass, who has been a constant source of love, encouragement, and emotional support throughout the writing of this report.

Initial funding for ATV was provided by the Defense Advance Research Project Agency (DoD), ARPA Order No. 4031, monitored by Naval Electronic System Command under Contract No. N00039-C-0107. Subsequent funding was also provided by DARPA, monitored by Naval Electronic System Command under Contract No. N00039-87-C-0182. My first three years at Berkeley were supported by a National Science Foundation Fellowship that made it possible for me to come to Berkeley.

Table of Contents

Table of Figures

List of Tables

Chapter 1 - Introduction

To err is human; to really foul things up requires a computer. - Anon.

Anything that can go wrong, will go wrong. - Murphy's Law

Murphy was an optimist. - O'Toole's Commentary on Murphy's Law

1. THE NEED FOR TIMING VERIFICATION

To err is human, and the greater the complexity of the task attempted, the more likely it is that errors will be committed. As digital systems designs grow more and more complex each year, programs to ensure the correct operation of the design become crucial. These programs fall into several categories, defined by the type of problem they address.

Circuit simulators are concerned with determining the detailed electrical response of the underlying electronic components: transistors, resistors, capacitors, and so on. They examine the values of voltages and currents as continuous functions of time. *Logic simulators* are concerned with the discrete responses of logic blocks built out of these components. They operate on signals with a finite, and usually small, number of values and strengths: typically 0s, 1s, and some intermediate or unknown states. Many assume that logic blocks either respond to changes in their inputs instantaneously (zero-delay models) or after some unspecified constant time that is the same for all blocks (unit-delay models). If they adopt a more sophisticated model of time, in which blocks can have delays of different lengths specified in realistic time units such as nanoseconds, they are often called *timing simulators*. The above programs are concerned with predicting the behavior of hardware that correctly implements a specified design; *fault simulators*, on the other hand, are like logic simulators but are concerned with simulating the effects of assumed hardware defects on the behavior of the system.

All of the above types of programs have one major characteristic in common: they are concerned with predicting the specific response of a system to a specific set of completely defined inputs. Circuit simulators and timing simulators can detect many common timing problems with a proposed design, but only if the problem is triggered by the specific set of input waveforms specified by the user. Because there are an exponential number of possible input patterns, it is generally not practical to exhaustively examine all possible inputs. Thus it is possible that timing errors will go undetected by these programs because they were not triggered by the set of input patterns chosen.

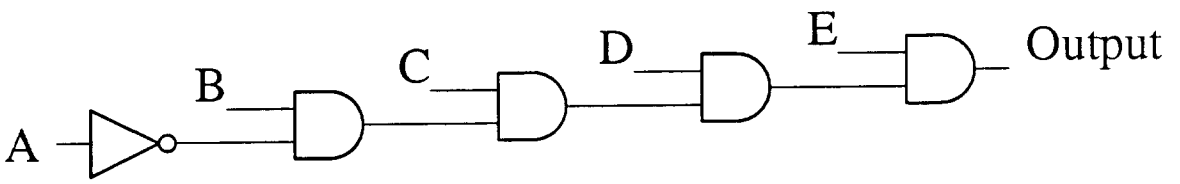


Figure 1-1: Logic example - an inverter and a series of AND gates. The long path delay from A to the output will only be observed by simulators if all the inputs B through E have the value 1 when a change in A occurs. Timing verifiers look for the worst possible delay path independently of specific data values.

This problem is illustrated by the logic circuit shown in Figure 1-1. Assuming that all the inputs arrive at about the same time, the worst-case delay path to the output will run from input A to the output, passing through one inverter and four AND gates. This path will only be observed, however, if the other four inputs either are already 1 or are becoming 1 when input A changes state. Thus if each gate took 20ns to compute its output and the overall output had to be available 90ns after the inputs arrived, there would be a timing error along this path that would only be observed by a timing simulator if these conditions occurred in the input data. None of the input values shown in Figure 1-2a produce this worst-case delay; in each case, the change in input A is masked by the change in some other input.

Timing verifiers attempt to address this problem by examining the timing behavior of the design independently of specific data values. They focus on *when* the signals stabilize rather than *what* specific values they have. They assume a worst-case setting for all unspecified signals, assuming that any change on the inputs of a block will propagate to its output.

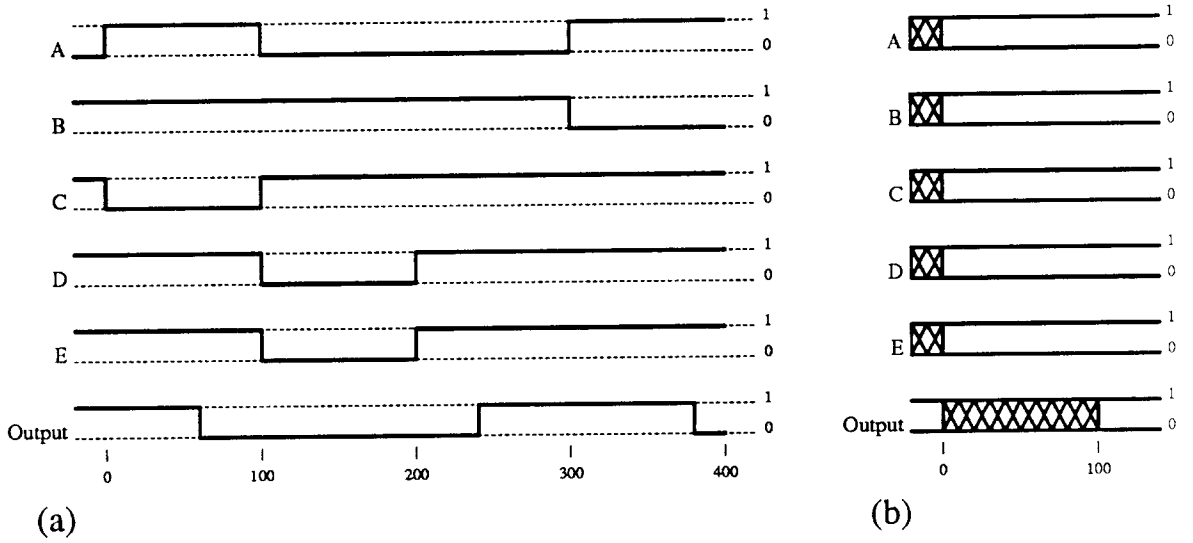


Figure 1-2: Results of timing simulation (a) and timing verification (b) of the example in Figure 1-1, assuming a 20ns delay through each gate. Timing simulation, looking at the response to specific inputs, does not observe the worst case delay to the output for these inputs. Timing verification, looking only at the stable vs. changing behavior of the signals, notes that the output can continue to change up to 100ns after the inputs stop changing.

Jouppi [Jou84] distinguishes between timing verifiers and timing analyzers: timing verifiers examine a design and a proposed clocking to see if any errors result; timing analyzers examine the design and determine how long the clock phases need to be to prevent errors (by computing critical path lengths). Thus timing verification is essentially a decision problem, and timing analysis is the related optimization problem.

Although many existing programs operate in only one of these two modes, the basic operations required of a program are the same in each case; I will show that both modes can be supported in the same program. In this report, I will generally use timing verification as a generic term to include both timing verification and timing analysis. Jouppi's usage is not universal in the literature; indeed, both "timing analyzer" and "timing verifier" have been used in the literature to describe programs that I would classify among the "circuit analyzers," because they operate on continuous voltage waveforms and require that all inputs have their values completely specified.

Experience has shown the need for automatic analysis of critical timing paths. The designers of the RISC I chip* found that the actual chip ran five times slower than predicted because of a transistor sizing problem[FVP82]. The MIPS chip developed at Stanford had a similar problem: the worst case paths were four times slower than predicted [Jou84, p. 3]. In response to these problems, the timing verifiers Crystal and TV were developed to analyze MOS chip designs. These timing verifiers aided in catching timing problems with a chip design before the chip was fabricated, and proved useful in identifying and fixing minor timing problems with subsequent designs. However, both programs require a completed design to be run, and both can only analyze the MOS circuitry on a single chip.

Both these limitations proved to be problems on the SOAR† project. Because the timing verifier Crystal could not be run until the layout of the chip was completed, it was only then that a major timing error was discovered, which required a redesign of the chip microarchitecture to fix [Pen85, pp. 273-283]. This cost a six month schedule slip (one person-year) in the completion of the chip. Another timing problem, involving the interaction of the chip with surrounding circuits on the board, was not discovered until the chip had been fabricated and was mounted and tested on the board [Ung87, pp. 52-53]. This problem increased the effective cycle time of the design by 25%.

The common theme to all the above problems is that in each case, the actual critical path of the design was not what the designers thought it would be. This is readily understandable. Those blocks a designer thinks will form a critical path get plenty of attention throughout the design process. From the beginning of the design, the designers are thinking about how these blocks will limit the speed of the design, how they can be sped up, and how they can be measured as the design evolves. These are the paths that are extensively simulated by a circuit simulator, such as SPICE [NaP73]. By the time the design is implemented, there are very few surprises left along this path. It is the unexpected path, the one path out of thousands or millions that has a timing problem no one suspects, that waits to snare the unwary designer. It is imperative to identify these paths as early in the design cycle as possible, so that

*Developed at U.C. Berkeley.
†Smalltalk On A RISC, a microprocessor developed at Berkeley.

the designers can spend their time thinking about them, rather than paths that will not ultimately limit the speed of the design. The earlier in the design cycle such paths can be identified, the easier it is to fix the problem. Had the SOAR microarchitecture problem been found before the control section was laid out, much wasted effort on layout that was ultimately discarded could have been avoided.

The SOAR experience shows that it is important to have a timing verification tool that can run early in the design cycle, using whatever information is available, and that it is important to be able to verify MOS chips together with surrounding TTL circuitry on a board. Although both the chip and the board circuitry were ultimately verified by two different timing verifiers, it was not possible to verify both in the same environment, or to relate the information provided by the two verifiers in order to identify the critical path that involved both. The design of ATV, the Abstract Timing Verifier that is the subject of this report, grew out of efforts to provide a common timing verification framework that could be used both for VLSI chips and their surrounding circuitry, that could be used early in the design cycle but could track the design as it progresses, that could be used to perform both timing verification and timing analysis, and that could address clock constraints passing through transparent latches.

Previous timing verifiers have used many different representations for time and delays. Some have represented time as a single number, where every event occurs at a specific time. Others have represented time as a range: as a min-max pair or a min-typ-max triplet. Others have used statistical descriptions. Many have tried to represent distinct delays for rising and falling transitions. Whatever the representation chosen, it has typically been built in to each timing verifier at a fundamental level, wedded to the design of the algorithm. This specificity has created artificial barriers to integrated timing analysis: programs designed to model TTL chips from data sheet information are unable to accurately model MOS chips where signal drive strength is crucial; programs designed to process MOS transistor structures are unable to analyze portions of a design that are not built out of such structures.

In ATV, I have instead defined the algorithm in terms of an Abstract Timing Model that defines several operations, but leaves implementation of these operations up to the particular model used. This means that any timing model that can define the operations of the abstract model can be plugged into the

program and run. The set of timing models is deliberately left open-ended, so that users can develop their own models to meet specific needs. The basic data structure of the program is intentionally generic: a dependency graph whose arcs may represent high-level blocks whose details are unimportant or not yet defined, or very low-level structures for a more detailed analysis. The intent is to provide a wide-ranging flexibility previously unavailable; a generic tool that could be wrapped in many different interfaces to meet different needs.

2. ORGANIZATION OF THIS REPORT

Chapter 2 discusses previous work in timing verification, including many of the generic issues common to different timing verifiers. It starts with a generic discussion of the longest path problem and continues by examining different previous timing verifiers, grouped by the representation they used for time and delay. An important sub-problem is how to verify designs that include transparent latches; this is discussed in a separate section at the end of the chapter.

Chapter 3 presents the Abstract Timing Model that is at the core of the Abstract Timing Verifier. The basic operations are defined, and many examples of different timing models are given, with discussion of how each model would implement the basic operations.

Chapter 4 discusses the implementation of the Abstract Timing Verifier, showing how the theory developed in the previous chapter is reduced to practice. I show how, by specifying input events in different *reference frames* that can be translated relative to one another, the same algorithm can be used for both timing verification and timing analysis.

Chapter 5 presents results of using the Abstract Timing Verifier. I present the results of running ATV on a large example for several different timing models and compare the critical paths that result. I also examine the observed performance of ATV and note ways in which the efficiency of the program could be improved.

Chapter 6 discusses future extensions of this work. These fall into three categories: enhancements to ATV, issues of hierarchical timing verification, and issues concerning the verification of asynchronous

and self-timed systems.

Finally, Chapter 7 contains the overall summary and conclusions.

Chapter 2: Previous Work

1. INTRODUCTION - LOOKING FOR LONGEST PATHS

The problem of static timing verification is essentially one of looking for the longest path through a directed graph.* It is well known that the longest-path problem is NP–complete in the general case [GaJ79, p. 213], but can be solved efficiently if the graph is acyclic [Law76, pp. 68-69]. The costs in the graph problem are derived from the delays through circuit elements or interconnections in the circuit to be analyzed. Although some authors [Hit82, KiC66] have formulated the problem with delays identified with nodes of the graph (Figure 2-1a), there is no particular advantage in doing so, since these delays can just as easily be attached to the output arcs of these nodes (Figure 2-1b).† The converse, however, is not true: problems formulated in the classical PERT style, with delays attached to arcs, cannot always be reformulated with the delays attached to the nodes without revising the graph.‡ I will therefore present examples in the latter style, with delays attached to the arcs.

To find the longest path through such a graph, there are three basic approaches. I refer to them as *breadth-first*, *depth-first*, and *depth-first with pruning*, referring to the classical search strategies of those names. Hitchcock [Hit82] refers to breadth-first and depth-first as *block-oriented* and *path-enumeration* strategies, respectively. These strategies will be illustrated by the example in Figure 2-2, taken from Lawler's discussion of PERT graphs [Law76, p. 62].

The depth-first strategy traces all possible paths through the graph, summing the total delay along each path. Thus it might start by tracing the path START-A-C-FINISH, with a total cost of 12, back up to node A and trace the path START-A-D-FINISH, with a cost of 18, and finally back all the way up to

*Shortest paths are also often of interest, but this is an easier problem with well-known efficient solutions [Law76, Chapter 3]. Often the desired shortest paths can be calculated by a trivial variation of the method being used to compute the longest paths.

† If block delays really are identical for all pins, there may be an efficiency gain from only having to represent the block delay once.

‡ If revising the graph is permitted, you can introduce a new node in the middle of each arc that carries the delay for that arc, making the original nodes all of delay zero. This doubles the number of arcs in the graph, although simplification may be possible in some cases.

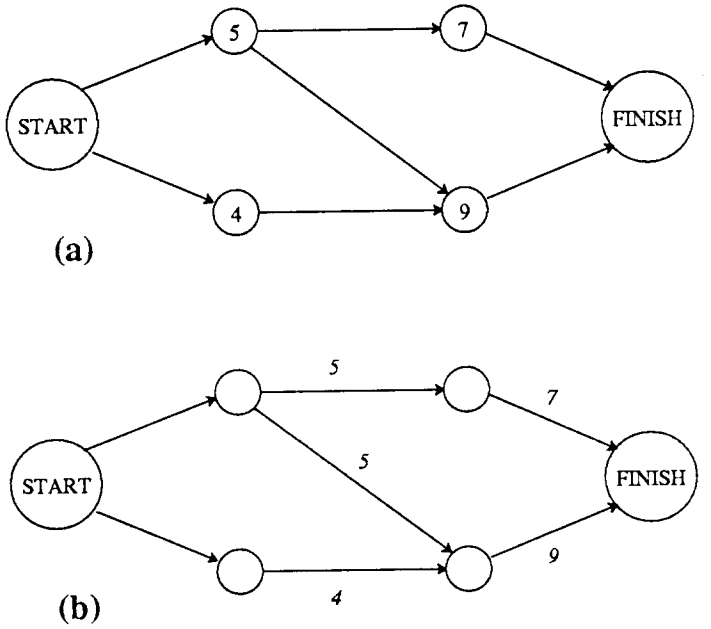


Figure 2-1: PERT-type problem. (a): With delays attached to nodes; (b): Reformulated with delays attached to arcs.

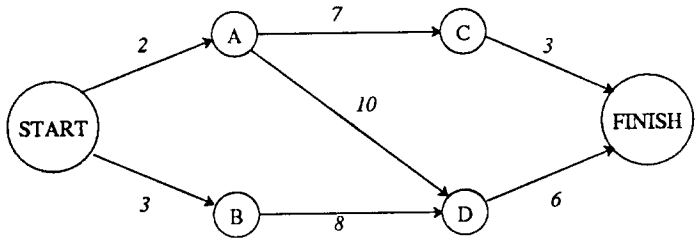


Figure 2-2: Classical PERT problem, with arc delays shown.

the START node and trace the path START-B-D-FINISH, with a cost of 17.

The depth-first with pruning strategy adds an optimization: whenever a node is revisited in the course of the search, the new path is only explored if the new arrival time at the node is greater than the previous worst time to that node. Thus in this example, when it reached node D the second time (from node B), it would note that D had previously been reached with a cost of 12, and the cost from node B is only 11. Thus it would terminate the search without exploring this new path all the way to the end.

In the breadth-first strategy, a node is not evaluated until all of its possible predecessors have been evaluated. The worst case arrival times from each predecessor produce a worst case time at the node that is then propagated to its successors. This strategy requires an acyclic graph. In the example, both nodes A and B would have to be evaluated before node D could be evaluated. Then the algorithm would compute the worst-case path to node D (from node A). Node C could be evaluated any time after node A was evaluated, either before or after B. After both D and C were evaluated, the FINISH node would be evaluated, finding the worst-case path from START-A-D-FINISH.

The reader should note that the strategies I describe as depth-first and breadth-first are not identical with the classical search strategies of the same names: classical depth-first search would always terminate the search when a previously visited node was reached, and classical breadth-first search would always explore the nodes adjacent to the least-recently explored node with unexplored neighbors before proceeding to other nodes [HoS78, pp. 263-269]. Thus it might be less confusing to adopt new terminology, such as Hitchcock's, to describe the longest-path search strategies. On the other hand, the essence of the two strategies is immediately suggested by my terminology, by analogy with the classical strategies: depth-first plunges deeply into the graph along a single path, only backing up to explore other paths once the goal has been reached, while breadth-first proceeds more methodically in a series of wavefronts from the source. Certainly this terminology is familiar to virtually all computer scientists (and has been used by several previous authors without comment), whereas Hitchcock's terminology, being original, is familiar only to those who have studied the timing verification literature. To avoid unnecessary confusion, I will refer to the classical strategies as classical depth-first and classical breadth-first hereafter.

What is the time-complexity of the three search strategies, for a graph with V nodes and E edges? Breadth-first search can be implemented to be linear in the size of the graph, O(V+E). Depth-first search and depth-first search with pruning, on the other hand, can be exponential in the size of the graph. One example created to demonstrate this point is the "Ladder Graph" of Figure 2-3. It has $2n$ nodes and $4n$ arcs. Because there is a choice between two paths at each stage, there are 2^n distinct paths from START to FINISH. (The arc costs have been selected such that each path has a unique integral length from 0 to

2^n-1). Thus depth-first search, which explores all the paths, will require $O(2^n)$ time.* Further, if the upper path is always explored first at every stage, the arrival times at each node will always be strictly increasing (the paths will be enumerated in order of increasing length). Thus depth-first search with pruning will not be able to prune any paths, so it too will require $O(2^n)$ time. (Depth-first with pruning can be modified to reduce its time complexity on acyclic graphs with a single output by keeping track of the best path found to the output from each node; but this analysis applies to the more general algorithm found in the literature [Ous85].) Although the relative sizes of the delays in this example may seem unrealistic, the result also applies to any monotonic linear transform of these delays. In particular, transforming the delay d to $d' = 1 + d/2^n$ makes all the delays between 1 and 2, but preserves the character of the problem.

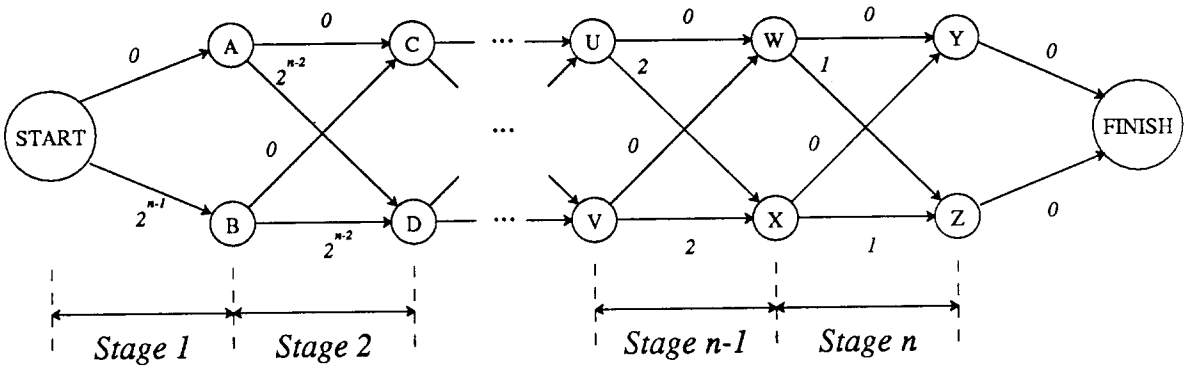


Figure 2-3: Ladder Graph: This example requires exponential time in the number of stages for both depth-first search and depth-first search with pruning (if the upper path is always explored first).

On the other hand, both breadth-first search and depth-first with pruning can miss potential critical paths when there are feedback loops in the graph. Breadth-first search requires that such loops be broken by cutting one of the arcs involved before the search begins, to make the graph acyclic. Depth-first search with pruning does not cut such arcs statically, but it does stop the search when a path loops back

*Although not every path will require the same time to explore, exploring a path must at least touch the FINISH node. Thus the FINISH node will be touched at least 2^n times, providing the required bound.

on itself (as does straight depth-first search). When combined with the pruning technique, this can cause the algorithm to miss critical paths, as shown in Figure 2-4. Figure 2-4a shows the graph, which includes a feedback loop that might be derived from a pair of cross-coupled NAND gates. Figure 2-4b shows the critical path to node E, while Figure 2-4c shows the critical path to node F. Because these two critical paths cover all the arcs of the loop, any arc that is cut to break the loop will eliminate one of the paths. Depth-first search with pruning also fails to find both paths. If path b is explored first, it will store an arrival time of 18 at node A, which will prune path c before it can be explored. Likewise, if path c is explored first, it will store an arrival time of 16 at node B, which will block path b. Only a full depth-first search will find both paths.

A different approach to problems such as that of Figure 2-4 was developed as part of the LEADOUT timing verifier [Szy86]. LEADOUT identifies strongly connected components of its dependency graphs and treats them as a unit, a kind of super-node. Thus in Figure 2-4a, LEADOUT would tread nodes A, B, C, and D as a unit, processing them separately.

1.1. Global and Local Slacks

Once worst-case events have been calculated for a graph such as Figure 2-2, *slacks* can be computed. Jouppi [Jou84] discusses the traditional definitions of *arc slacks* and *node slacks*. The *local slack*, also known as *arc slack*[Jou84] or *activity float*[MPD83, p. 79], of an input to some arc is the amount by which that event could be additionally delayed before it would delay its immediate successor. The only case where a local slack can be non-zero is when there are multiple input arcs to a node. Local slack is clearly computed on a per-arc basis. From this definition, it is clear that the worst input to a node has zero local slack.

The *node slack* (Jouppi) or *path float* (Moder) of an event is the additional amount by which it could be delayed without increasing the critical path length (or violating the final schedule constraint, if this is different from the path length). I define the *global slack* of an event with respect to some overall constraint or final destination event to be the minimum sum of local slacks from the initial event to the

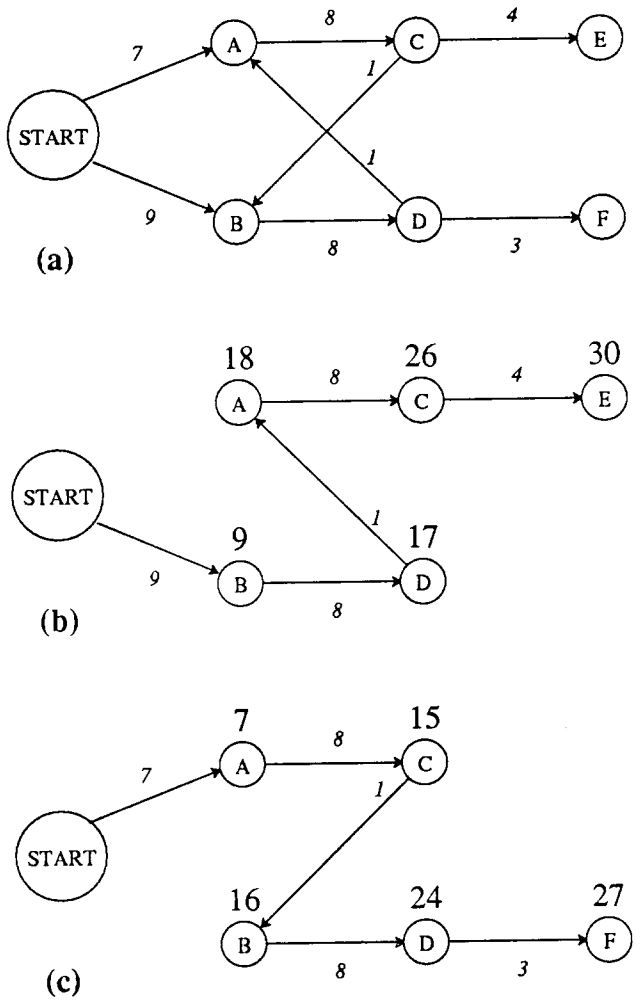


Figure 2-4: Static loop-breaking can miss critical paths. (a): Dependency graph. (b): Critical path to node E. (c): Critical path to node F. When using breadth-first search, any static loop-breaking will cut an arc in either path (b) or (c); when using depth-first search with pruning, whichever of (b) or (c) is explored first will cause the other to be pruned.

final event, plus whatever slack the final event may have with respect to the ultimate constraint. If the slack of the final event is 0 (what Moder refers to as the *zero-slack convention*), it is apparent that nodes on an overall critical path will have zero global slack. I believe that the two concepts are equivalent.

Figure 2-5 shows how slacks are calculated. Local slacks are shown at each node with multiple predecessors. If the zero-slack convention holds (global slack of FINISH is 0), then nodes START, A, D, and FINISH have global slack of 0, node C has a global slack of 6, and node B has a global slack of 1. If there was a constraint that the FINISH arrival time had to occur on or before time 22, then START, A, D,

and FINISH would have a global slack of 4 with respect to this constraint, node C would have a global slack of 10, and node B a global slack of 5. If there was an additional constraint that the arrival time at node C had to occur by time 10 (Moder does not address slack for multiple constraints), then the global slack for nodes START, A, and C would drop to 1, while the other global slacks remained unchanged.

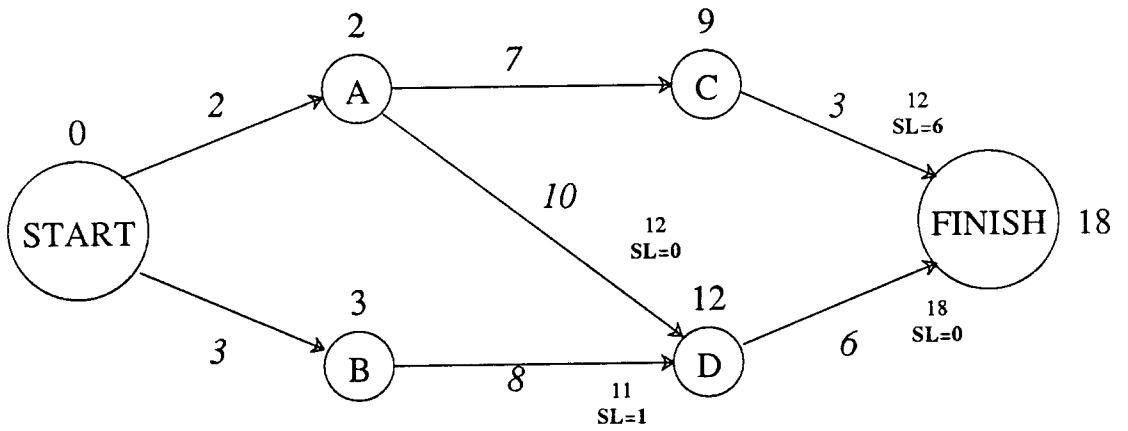


Figure 2-5: Slack computation. Arc delays are shown in italics, node arrival times are shown at each node, local slacks (and input arrival times) are shown where multiple arcs come together at a node (at node D and FINISH).

2. STATISTICAL TIMING VERIFICATION

One of the major distinctions between previous timing verifiers is between those that adopt a statistical approach to the analysis and those whose approach is more deterministic. Much of the work on statistical methods has come from IBM. Perhaps this is due to the early development of automated statistical methods there, which may have generated a large community of designers with experience using statistically-based analysis tools, and therefore more accepting of such tools than the design community at large. The timing verifiers in this section all use statistically based descriptions of events.

2.1. PERT

Although informal analysis of timing requirements is probably as old as logic design itself, the first automatic timing verifier reported in the literature was an application of a PERT chart analysis program to the problem of logic design [KiC66]. Kirkpatrick and Clark realized that by interpreting logic functions as jobs to be performed and the delays through each logic element as the time to perform the corresponding job, they could cast the problem of finding the length of a critical path through a logic network as a PERT problem of finding the minimum time required to complete a series of interdependent project tasks. Because they were dissatisfied with the overly conservative nature of pure worst-case analysis, they turned to statistical methods as a way of expressing the relatively independent variation in delays between the individual discrete components they were studying.

Delay values for each block were input as three numbers: a minimum time, a, a typical time, m, and a maximum time, b. Two such triplets of numbers were supplied for each block, representing the delay for a rising output and a falling output. These were converted to a mean, μ, and variance, σ^2, via the equations:

$$\mu = \frac{a+4m+b}{6}$$

$$\sigma^2 = \left[\frac{b-a}{6}\right]^2$$

These were then used as the elements to be summed in the resulting computation. This is one of the few early examples of coercion between two different delay formats, a concept that is used extensively in ATV. It appears that the analysis was examining paths one at a time, although Hitchcock *et al.* [HSC82] cite PERT as an example of the block-oriented algorithms. The program is indeed described as computing a value for each block, but this value represented the delay through the block, not the signal arrival time there.*

*Ironically enough, the association of delays with arcs is probably more appropriate for circuit delays, where the delay from each pin to the output of a gate may be distinct; associating delays with nodes is more appropriate for project management, where the delay represents the time to complete a specific task that may have many predecessors and successors. Yet the first automated PERT tools for project management were strictly arc-based [MPD83, p. 38], while this first paper on the ap-

Once the arrival times at each output had been calculated (as a mean and variance), the program worked backwards from specified required times at the outputs to compute the acceptable slacks for each input. These slacks were also computed as both means and variances; with the variance interpreted as the maximum allowable variance in the input arrival time if it arrived with mean delayed by the maximum slack.

2.2. TA

The statistical approach taken by PERT was significantly extended by a new timing verifier - the IBM TA program developed by Hitchcock *et al.*[HSC82, Hit82]. TA used a breadth-first "block-oriented" algorithm (levelizing the network, and processing it in order so that the inputs to each block were calculated before the block was evaluated). It represented block delays as a mean and standard deviation for both rising and falling delays. It also stored the inverting properties of the logic function associated with each block so the delay could be applied properly. When multiple inputs arrived at a block, each was delayed by the block delay, and then the result with the largest value of $\mu+\beta\sigma$ * was selected as the output time of the block. When looking for shortest paths, the program would instead choose the result with the smallest value of $\mu-\beta\sigma$. Rising and falling results were selected independently.

The TA program was used to analyze designs that were designed to LSSD† standards for testability [EiW77]. This simplified the task of the verifier by enforcing restrictions on clock signals. Input to an analysis specified one clock that was to be used for propagation purposes, and another that was to be used for comparisons. Paths were checked starting from the propagation clock, propagating through the logic network, and being compared against the comparison phase of the compare clock. The scheduled arrival time of the comparison clock was translated into requirements that could be propagated backwards to calculate the global slack for nodes in the design. An interactive analysis program was included to help the

plication of PERT to logic design describes how they altered this standard practice in order to associate the delays with the nodes!

*For example, the user could choose $\beta=3$ to select the result with the largest 3σ point.

†Level-Sensitive Scan Design

user analyze the results.

2.3. Statistical Refinements

The difficulty with the way TA computes the latest output arrival time at a block is that it does not accurately reflect the interactions between the input distributions. For example, if there are five independent normally distributed inputs to a block all with mean 0, any individual input will arrive before time 0 half the time, but the probability that the latest one of them arrives before time 0 is only 1/32. The worst-case distribution (for maximal length paths) is the distribution of the maximum of two random variables, one drawn from each input distribution. This distribution is usually different from any of the input distributions (see Figure 2-6).

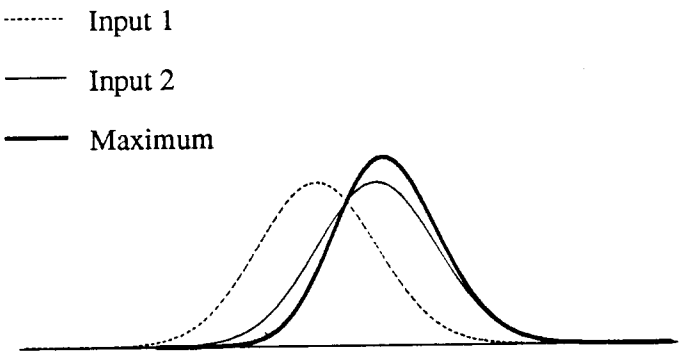


Figure 2-6: Two independent normal density functions and the density of their maximum. The maximum is not normally distributed (although it is approximately so). The TA program would choose Input 2 to represent the worst-case time; Shelly and Tryon correct for the error between this and the actual maximum.

This problem was discussed by Shelly and Tryon [ShT83], who described the difference between the 3-sigma point computed by T.A. and the *d-point* (the point whose cumulative distribution is equal to that of the 3-sigma point of a normal distribution, .9986501) of the actual distribution. The method for calculating the d-point of the actual distribution is described in [BCS84]. A mathematical justification for statistical methods of delay calculation is offered in [TAR84].

2.4. Probabilistic Pert

A more radical generalization of the PERT technique was developed by Arthur Nádas [Nad79, Nad80]. He analyzed PERT-like dependency graphs where each delay was a random variable, X_i, with an arbitrary distribution function, $F_i(t)=\Pr(X_i \le t)$. He observed that the calculation of the exact distribution of the longest path through the network appears intractable, except in a few very special cases. Even when the delays are all assumed to be independent, the corresponding event times may not be, due to reconvergent fan-outs. He recast the problem as a parametric linear programming problem, seeking a worst-case bound over all possible joint probability distributions corresponding to the given marginal distributions.

2.5. SCAT

Another unique statistical description is the two-point discrete probability distribution used in the SCAT timing verifier [GSS86, Ste85]. SCAT models delays with three numbers: a, b, and p; the interpretation is that the delay has value b with probability p, and value a with probability $q=1-p$. This model combines some of the features of min-max models with statistical descriptions. It will be discussed in more detail in Chapter 3.

2.6. Other Statistically-Based Timing Verifiers

Other timing verifiers have used statistical descriptions for delay. These include the GRASP system from Honeywell [Wol78]. DIGSIM [Mag77] represented delays as normally-distributed random variables for purposes of simulation; this program was a *timing simulator*, rather than a value-independent timing verifier.

3. STATE-SEQUENCE BASED TIMING VERIFICATION

3.1. The SCALD Timing Verifier

Another important class of timing verifiers represents events as sequences of states. For example, the SCALD timing verifier [McW80a, McW80b] used a total of seven different states that signals could assume: 0, 1, S (stable), C (changing), R (rising), F (falling), or U (unknown). The value of a signal would be represented as a sequence of these states, with transition times specified as single numbers; thus a clock signal might have the value: "0:0 R:1 1:4 F:12 0:15" (0 at time 0, rising from 1-4, high from time 4-12, falling from 12-15, 0 after 15), while a data signal might have the value: "S:0 C:7 S:9 C:15 S:18" (changing from times 7-9 and 15-18, stable otherwise). System inputs (including clocks) could have their values described by the user in this form; system outputs could also be specified in this way (although for outputs the stable and changing states were interpreted as assertions to be checked against the calculated values). Each state sequence also had an associated *skew* value, an numeric value that indicated the specified transitions could actually happen up to the skew value later than the specified transition times.

The timing description of a design was built out of a small set of low-level primitive components, such as AND, OR, and NOT. The effects of combining inputs of different states were defined in a table for each component: for example, the AND of a 0 and a S is a 0, but the AND of a 1 and a S is a S. Higher-level components could be specified in terms of these components, and would be expanded as macros into the primitive components before the analysis was performed [McW78a, McW78b]. Delays were specified as a min and a max time for each component; these were component delays which applied equally to all inputs of the component.

The basic algorithm used was to initialize known clocks and inputs to the sequences of states specified by the user, and all other signals to U (unknown) initially. The system then proceeded to calculate values of signals using an event-driven scheduler. The inputs to a primitive component would be combined according to the rules for that component, as if the component had zero delay, with each

transition on any input potentially generating a corresponding transition on the output. The result would then be delayed by the minimum delay of the component, with the difference between the maximum delay and minimum delay added to the skew of the signal.* If the inputs had different associated skews, the skews would be added back in to the state sequences (introducing rising or falling states, or extending the changing state) before combining. Whenever this calculation resulted in a new set of states for some signal, the components driven by that signal were scheduled to be reevaluated. The calculation continued until no further changes in signal states were computed.

Constraints were specified by special primitive components that were part of the logic design: pulse-width checkers that would ensure that clock widths did not diminish below a specified value, and set-up and hold checkers that checked the stability of one signal around a specified edge of another. After values for all signals were computed, the checkers would be invoked to ensure that the calculated signal values met the required constraints, and output signals were checked against their assertions. Any violations were reported to the user, but no critical path information was available that would identify the source of the violation. Instead, the user would have to trace the violation by hand by examining the computed state sequences for all the nets working backwards from the violation to its source. Another problem with this type of checking is that signals could only be checked to see if they were stable; there was no guarantee that the stable value was valid. Thus if the input to a register missed its required arrival time by a small amount, it could be flagged as missing a set-up constraint because it was still changing in the window where it was required to be stable. However if it was sufficiently late, it could miss being reported because it was still stable (with the old value) throughout the set-up and hold window, not starting to change until after the required hold time.

One of the problems with value-independent analysis is that it may be too pessimistic, calculating delays through paths that are not actually achievable. SCALD addressed this problem by providing a means for *case analysis*: the user, on discovering a false critical path, could specify that the design was to

*The purpose of this separate skew value was to ensure that clock pulse widths were not reduced by going through a buffer with uncertain delay.

be analyzed first assuming that a particular signal had the value 0, and then reanalyzed assuming that it was 1, during its stable period. The specified value would be propagated as far as possible through the design, using logic simulation. The example given by McWilliams is shown in Figure 2-7. Worst-case analysis would report the worst-case path through the two muxes as 40ns, but in fact the two 20ns paths are mutually exclusive, so the true path is only 30ns long. Case analysis could be performed on the select input to the muxes, revealing that the critical path was only 30ns long for each of the two possible values of the select line.

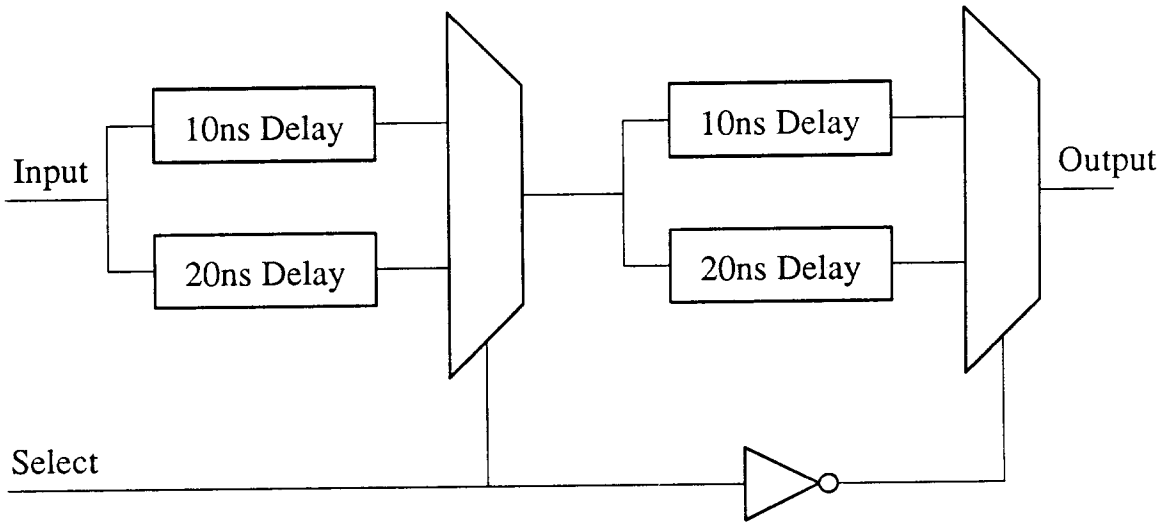


Figure 2-7: The need for case analysis. The true path through the two muxes is only 30ns long, but a value independent analysis identifies the worst path as 40ns. Case analysis on the select input can resolve this error.

3.2. RDV

Another timing verifier that uses state sequences to represent events is RDV, developed by Sumit Ghosh [Gho87]. RDV uses only five states, eliminating the rising and falling states used in SCALD. Primitive elements are modeled by Ada concurrent routines, using the concurrent scheduling provided by the Ada language environment to do the scheduling. Like SCALD, delays are represented as a min and a max delay. Unlike SCALD, RDV can be used to analyze asynchronous designs by a form of Monte

Carlo simulation: asynchronous clocks are generated using a random-number generator to assign actual times to their edges, and the resulting schedule is analyzed for errors. This will have a good chance of detecting gross errors, although errors that are sensitive to the exact relationship between two asynchronous signals will not be detected unless the randomly generated times for these signals happen to be in the correct relationship.

3.3. Other State-Sequence Work

The Daisy timing verifier [CaL82] is similar to the SCALD timing verifier, but extends the number of states by attaching one of three signal strengths to each of the seven SCALD states to analyze MOS designs. In effect, this increases the number of states to 21.

An unusual technique combining breadth-first and depth-first search was developed by Muraoka *et al.* in the ACTAS timing verifier [MIK85]. Block delays were specified as a min and max delay for both rising and falling output transitions. They used a breadth-first SCALD-like algorithm to search for timing errors. When an error was found, they would then use the information from the initial search to define bounds for a limited depth-first search for the critical path that caused the error.

The issue of defining timing verifier states was addressed by Mezzalama *et al.* [MPV81], who presented an analysis technique for automatically generating a large number of states, starting from a user-selected set of basic values, each of which represents some combination of signal strength and value (current/voltage combinations). They then combine these in various ways to express timing verifier states in terms of unions of sequences of the basic states. For example, if the basic states are 1 and 0, they could create a state that represents the union of states 0 and 010; the interpretation of this state is that the signal starts and remains at 0, but may make an optional single transition to the 1 state in the middle of this composite state, i.e., the signal has a possible hazard during this state. The difference between this state and a state consisting solely of the sequence 010 is that the latter asserts that the signal does make such a transition during this state. There are two additional states: G representing a signal that may make more than a user-specified number of transitions, and U, the unknown state. The seven SCALD states

can be built out of basic states 0 and 1, using sequences with no more than 1 transition:

$0 = \{0\}$

$1 = \{1\}$

$R = \{01\}$

$F = \{10\}$

$S = \{0\} \cup \{1\}$

$C = \{0\} \cup \{1\} \cup \{01\} \cup \{10\} \cup G$

$U = U$

(Note that the changing state does not require that the signal value change, it merely permits it.) They give rules for combining these states in a SCALD-like timing verifier, using min-max delay specifications. One of the advantages of their approach is that the system can automatically deduce the rules for combining these timing verifier states for a given primitive, given the rules for combining the basic states. Thus where SCALD requires a 7x7 table for each 2-input primitive element, their system would only require the boolean truth table for the same element, and could derive the rest of the table from the logic function. For example, given the truth table for AND, it would deduce that the AND of 0 and S was $\{0\} \cup \{0\}$, or $\{0\}$, while the AND of 1 and S was $\{0\} \cup \{1\}$, or S. This ability in turn allows them to expand the number of timing verifier states far beyond the seven available in SCALD, which may provide more information about the actual legal transitions a signal may undergo.

The earlier work of Bose and Szygenda [BoS77] was much more limited, calculating the rules for combining states in two fixed simulation algebras (five state and nine state) from the boolean truth table for a gate. Their work was oriented towards hazard detection in simulations, rather than timing verification proper.

4. SWITCH-LEVEL TIMING VERIFICATION

The growth of custom MOS chip development has led to the development of timing verifiers oriented towards MOS designs, where the basic element being analyzed may be as small as an individual transistor. Because MOS transistors basically act like switches, with a gate node that controls current flow through the other two terminals, these timing verifiers are known as switch-level verifiers. Three such switch-level timing verifiers are Crystal, TV, and LEADOUT. All three programs take a MOS transistor netlist as their basic input, and analyze the delays between groups of transistors. Although these programs are well-suited to analyzing complete MOS designs, their tight coupling with the MOS transistor representation makes it difficult or impossible to analyze designs including incomplete or non-MOS components, such as the surrounding circuitry on a board.

4.1. Crystal

Crystal [Ous83, Ous84] groups transistors into *stages* for purposes of timing verification. Although this grouping could in principle be done in advance, Crystal constructs the stages dynamically during the analysis. Stage construction starts at the gate of a transistor (the *trigger* transistor) and proceeds outwards from the source and drain nodes of that transistor, through any intermediate transistors that are connected via source or drain, looking for attached gates of target transistors and sources of 1 or 0 (Vdd or Gnd). A stage consists of those transistors in a direct source-to-drain path between a given target transistor and a source of 1 or 0, that either includes the original trigger transistor or connects to a weak signal source opposite in polarity to a stronger signal source connected through the trigger transistor. The former case represents the potential for the target to be driven to the source value when the trigger transistor switches on, connecting the target and the signal source; the latter case represents the potential for the target to be driven to the weak source value when the trigger transistor switches off, isolating the stronger opposite source from the target. Figure 2-8 shows these two kinds of stages: one originating at the Gnd node, passing through the trigger transistor and the transistor marked X to get to the target, and the other being the weak path to Vdd that also passes through transistor X to get to the target. If these

transistors were later analyzed with transistor X as the trigger the same two stages would be found, but now they would both be the first type of stage, activated when X turns on.

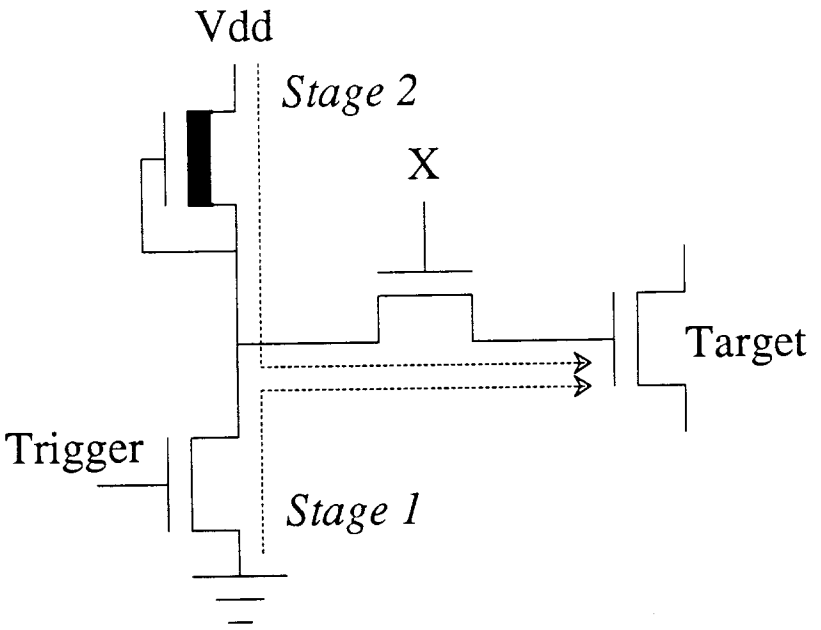


Figure 2-8: Crystal stages. There are two stages from the trigger transistor to the target, both passing through the transistor marked X.

Crystal uses depth-first search with pruning to analyze a transistor network, starting from user-specified input events. It calculates the delay for each stage on the fly as a single number, using one of several different ways to compute the delay from the transistor structure [Ous84]. (Although Ousterhout refers to these different computations as delay models, they are not delay models in the sense I use the term here, but rather different coercions to convert a transistor structure into a single-number delay.) The basis for these calculations is the Penfield-Rubenstein timing model for RC networks [LiM84, PcR81, RPH83]. Although this model calculates bounds on the possible delay through the network, Crystal converts these bounds to a single number. Crystal assumes that the trigger transistor in each stage is the last transistor to turn on (or off) and hence this is the event that will control the delay to the target. Crystal keeps track of the direction of the transitions it is calculating, computing worst-case times for rising and falling transitions separately. In most cases, Crystal can determine the direction of signal flow through each stage, though there are some structures, like barrel shifters, that give it trouble.

In such cases, Crystal relies on user labels in the design to disambiguate the direction of signal flow (the user would add these after analyzing the output from a previous run of Crystal).

4.1.1. ELogic with Crystal

ELogic is a switch-level simulator that models signal voltage levels by a user-selectable number of discrete states [KKS84]. It models MOS devices by characterizing for each set of input and output states which adjacent state (if any) the output will be driven to, and how long it will take to achieve that state. ELogic has been used as a delay calculator for Crystal to compute the delays through each stage [HKN86]. Unlike the usual Crystal delay models, this combination preserves not merely the single-number time of a transition, but also the (discretized) waveform, which is then used as input to the next stage. In this way, greater accuracy can be achieved than with the usual Crystal models, since MOS circuit delays tend to be sensitive to the shape of the input waveform.

A very similar implementation was outlined by Neto and Vidigal [NeV86]. Dagenais and Rumin [DaR86] went one step further, computing min and max waveforms for both rising and falling transitions based on the envelope of waveforms computed by a circuit simulator. They coupled this with a detailed analysis of the series-parallel structure of each gate.

4.2. TV

TV [Jou83a, Jou83b, Jou84, Jou87], like Crystal, analyzes groups of transistors. Unlike Crystal, it relies on breadth-first search, on direction-finding heuristics rather than user labels to find the direction of signal flow through complicated transistor structures, and on algorithmic analysis of clock phase lengths for transparent latches, rather than user specification. The breadth-first search is preceded by a quick "predecessor counting" phase, which uses a classical depth-first search from known active inputs to count the number of active predecessors to each node in a given phase. Any loops are broken in this phase. Once the number of active predecessors is known for each node, the breadth-first delay computation can proceed, with each node that is evaluated checking each of its successors, and scheduling them

for evaluation if it is the last of their predecessors needing evaluation.

4.3. LEADOUT

LEADOUT [Szy86] uses a unique interpretation of its timing graph: rather than just representing individual points in the circuit, nodes in the graph represent specific transitions at those points, per clock phase. Thus there would be one node representing a rising transition at some point "A" in phase 1, and another distinct node representing a falling transition at A in the same phase. Arcs represent causal relations between these transitions. The program derives delay equations relating these transitions. LEADOUT handles feedback loops by identifying strongly connected components* of the graph and treating them as a unit for purposes of generating delay equations. For a fixed input clock schedule, LEADOUT solves the equations by a relaxation algorithm. No procedure is given for solving these equations when the initial clock schedule is unknown.

4.4. Other Switch-Level Verifiers

Another switch-level verifier for MOS designs was developed at American Microsystems, Inc. [NG81]. This program also used an asymmetric rise/fall model with single-number components.

5. OTHER TIMING VERIFIERS

One of the earliest efforts at timing verification was reported by Harrison and Olson [HaO71]. Unlike most subsequent timing verifiers, they were primarily concerned with avoiding races at the clock inputs of flip-flops that could cause hazards at those inputs. They started at the clock input of the flip-flop, and worked backwards through the network to the original clock. Delays were represented as a min and a max, but the analysis could also be done in statistical terms. Paths were traced for both rising and falling transitions, but it is not clear if different delays were used; the distinction may simply have

*A strongly connected component of a directed graph is one in which every node is reachable from every other node of the component.

affected which gates were traced back through (NANDs vs. NORs, etc.).

The NELTAS system [NST82, SYA81] used both min and max delays. It attempted to develop simplified timing models of components for use in verifying higher levels. They were aided in this by the clocking scheme used: apparently all storage elements were edge-triggered registers clocked by a single clock. Thus register-to-register paths within a block could be deleted entirely and replaced with a constraint on the clock width. The primary analysis mode was depth-first search. Hitchcock [Hit82] thinks the critical path mode was block-oriented (breadth-first), but Sasaki *et al.* [SYA81] do not seem to have stated this clearly one way or the other.

Another depth-first system was the Critical Path Delay Check System [KYC81]. This system used asymmetric rise/fall delays which were probably single numbers, though there is one hint that statistical information was used.

Another hierarchical system was described by Tamura, Ogawa and Nakano [TON83]. This program used a breadth-first search to generate rough timing information, and then used a depth-first algorithm to refine the data in the vicinity of the critical path suggested by the breadth-first algorithm. This program used an asymmetric rise-fall model.

Yet another hierarchical system was developed by Reddi and Chen at Sperry [ReC86]. This system used both min-max and statistical descriptions for delay, though the paper does not describe how they interacted. The TRACE timing verifier, developed by Bening *et al.* [BLA82], used an asymmetric min-max model for delay.

6. TIMING VERIFICATION WITH TRANSPARENT LATCHES

Transparent latches* complicate the timing verification of designs that include them. Edge-triggered or master-slave storage elements interrupt the flow of data through them such that critical paths will never extend through them. If all the storage elements in a design are of this type, we only need to

*A transparent latch is a storage element whose output changes directly in response to changes in its input during some state of its controlling clock (rather than waiting for some clock transition to occur).

look at paths through combinational logic between storage elements. But with transparent latches, critical paths can potentially extend over multiple cycles, passing through several intermediate latches at any time during their respective open periods. The simplest solution is to ignore the nature of transparent latches, treating them as if they were edge-triggered by the closing edge of the controlling clock. This is a very conservative solution that prevents the logic designer from exploiting the existence of the transparent latch. Various other solutions to this problem have been adopted by the MOTIS timing verifier, Crystal, TV, and SCAT. These solutions are discussed below.

6.1. MOTIS Timing Verification

The timing verifier incorporated in the MOTIS system [Agr82] shares many features with the switch-level timing verifiers I discussed previously. It verifies MOS designs, it represents delays through logic elements as a rising delay and a falling delay, both of which are single numbers, it allows multiple non-overlapping clocks, and it analyzes paths extending through transparent latches for multiple clock phases. In analyzing multiple-phase paths, it requires all clock edges to be fixed and specified with respect to a normalized machine cycle. Thus the only parameter it varies is the length of the machine cycle (actually it varies the clock frequency, the reciprocal of the cycle length) stretching or shrinking all clock lengths and spacings uniformly with the cycle length. The program uses depth-first search to trace all clocks forwards from user-specified origins, adding the delays for both initially rising and initially falling edges until it reaches a latch controlled by that clock. It identifies for each type of latch an opening and closing edge in terms of the input clock signal. Transparent latches have different opening and closing edges; edge-triggered registers have opening and closing edges that are the same.

Once each latch has been marked with its opening and closing clock edges and the rising and falling skew (i.e., the total rising/falling delay) from the controlling clock to the latch, the main analysis can begin. The program uses depth-first search from each clocked latch to examine all paths originating from that point. Paths extend through transparent latches that are clocked by phases other than the clock phase of the origin latch. Paths terminate when they hit an edge-triggered register, reach a user-specified search

depth,* or hit a latch with the same closing clock edge as the original latch. This last restriction means that no paths longer than one machine cycle are examined. This restriction does not appear to be vital to the algorithm, and could probably be removed. Like the user-specified search depth, however, it appears to have been introduced to limit the otherwise exponential run-time of the algorithm.

Each path discovered is considered to begin on the arrival of the closing edge of the original latch. The arrival time of the signal at each intermediate latch is compared with the arrival of the opening and closing edges at that latch (for an initially specified operating frequency). If the signal arrives before the opening edge, it is replaced by the arrival time of the opening edge for the rest of the path. If the signal arrives after the closing edge, it is considered an error. The program then re-analyzes the path for different clock frequencies, seeking the maximum frequency that does not cause the path to fail. After the program analyzes all the paths it finds, the user can request a plot of the number of failing paths vs. clock frequency.

This work has several limitations. The most serious is the limitation on the search depth, required by the exponential nature of the algorithm. Unfortunately, this means that paths that go through a greater number of logic levels than expected will not be discovered - yet if such paths exist, they are most likely to be the critical paths in the design. Perhaps the user can identify the need to extend the search depth from the output of the program, but this seems like a likely source of error in using the program. The exponential nature of the algorithm and the repeated iterations to solve for the maximum operating frequency of each path suggest that run time may become an issue. The results reported in the paper indicate that run-times become excessive above a search depth of about 15 on a Harris computer and about 20 on a Cray. The assumption that all clocks scale perfectly with the cycle time may be excessively rigid for many design environments. In spite of these limitations, the Agrawal algorithm [Agr82] is an important seminal influence on the basic analysis algorithm of ATV, which overcomes most of these limitations.

*The default limit is 15 logic levels

6.2. Crystal

Crystal provides only loose support for analyzing designs that use *transparent latches* (also known as *level-sensitive latches*). The user analyzes the design one clock phase at a time, specifying values for all the inactive clocks in the current phase. As in case analysis, these are propagated through the design as far as possible by logic simulation. The user then specifies the active clock or clocks for this phase as an input, and the system computes the latest settling time for all nodes. All the activity associated with a given clock edge is implicitly assumed to take place during that clock phase. If this is not the case, it is up to the user to specify manually how much of this activity is to be allocated to the current phase, and how much is to be carried over into the analysis of the next phase.

6.3. TV

TV also includes a "cycle borrowing" algorithm [Jou84, pp. 29-36] * that attempts to reduce the cycle time of a design by exploiting the existence of transparent latches. Because critical paths can pass through such latches in the middle of their open phase, the overall critical path need not arrive at all such latches at the beginning of that phase. TV begins by computing the required cycle time if all latches were edge-triggered registers where the critical path did need to arrive by the beginning of the phase, and then attempts to reduce this cycle time by "borrowing" time from adjacent cycles for the current critical path or paths, pushing their arrival times into the open phase of the next consecutive transparent latches.

Figure 2-9, adapted from Jouppi's thesis, illustrates the cycle borrowing algorithm. Figure 2-9a shows the example to be studied, with delays for the arcs as shown. Figure 2-9b shows the result of the initial analysis, assuming that all latches are edge-triggered. Jouppi's discussion assumes that the clock duration and spacing must be the same for all three phases. Hence there is 5ns of extra slack during phase B, and 25ns of extra slack during phase C. Because the path from A to B determines the cycle

*The discussion of cycle borrowing in [Jou84] is difficult to follow. The example on p. 30, discussion in the main text, and pseudo-code on p. 32 are mutually inconsistent, possibly due to multiple inadvertent typographical errors. The example I give here (Figure 2-9) is based on Jouppi's example, but simplified and modified to conform to the discussion in his main text.

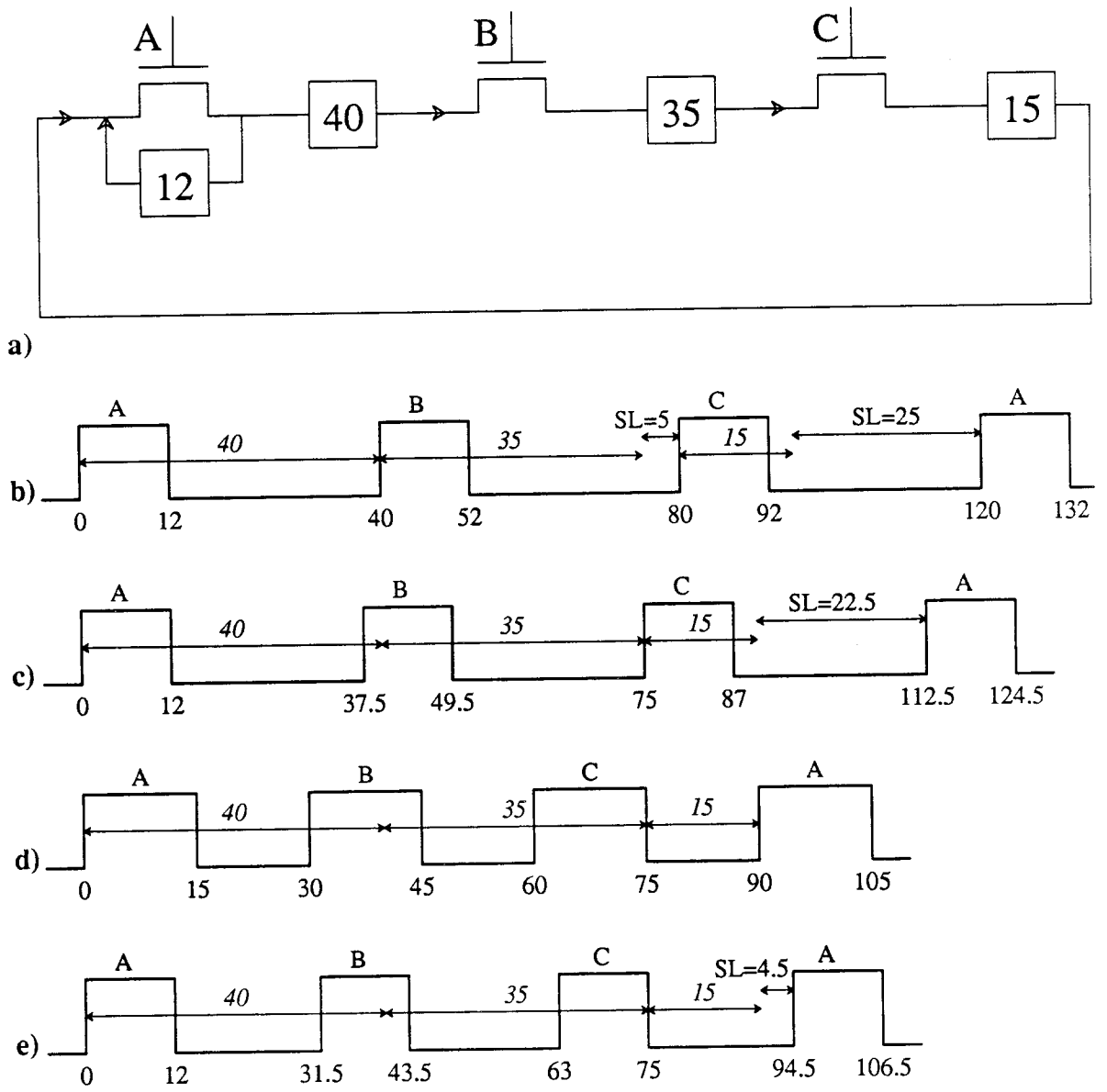


Figure 2-9: Cycle Borrowing. (a): Example with delays shown; (b): Initial cycle calculation; (c): After borrowing from phase B; (d): Desired borrowing from phase C; (e): Feasible borrowing from phase C. The cycle time is equal to the rising edge time for phase B in all cases.

length, the algorithm attempts to reduce the cycle length by borrowing slack from phase B. It takes the 5ns of slack from phase B and distributes it evenly over phases A and B, producing the schedule shown in Figure 2-9c. Now the paths in phases A and B are both critical. The algorithm would like to borrow the remaining 22.5ns of slack from phase C and distribute it to all three phases, producing the minimal

clock schedule shown in Figure 2-9d. Unfortunately, this would require that the high phase of the clock be lengthened to 15ns so the data can arrive at C by the falling edge of the clock, and the short path from A to itself restricts the maximum clock length to 12ns so new data from this path is not latched until the subsequent phase. Thus the best the algorithm can do, subject to this constraint, is to only borrow 18ns of slack from phase C, producing the final optimal schedule shown in Figure 2-9e.

Jouppi observes that this algorithm has an exponential worst-case running time, as in general one needs to borrow slack from multiple fanouts of each node, and this process can be recursive. To avoid this cost, and because he observes that it is rare to borrow slack more than one clock cycle ahead, he uses an approximation algorithm that only looks at cycle borrowing from the next adjacent cycle. In the example of Figure 2-9, this approximation algorithm would stop after the first borrowing, yielding the schedule shown in Figure 2-9c. (He notes that this example was constructed to be non-optimal for the approximation algorithm).

6.4. SCAT

The SCAT timing verifier also includes an algorithm to solve for the minimum clock period for paths that include multiple transparent latches. This algorithm assumes that all clocks have the same width. It also does not appear to account for different clock skews to different latches, or to allow for any underlap between consecutive clocks (i.e., it assumes that the falling edge of one clock occurs simultaneously with the rising edge of the next). For the example shown in Figure 2-10 (which is based on Jouppi's example), a clock period of 25ns will be computed by this algorithm, subject to these assumptions. This clock width is just sufficient for an event starting on the rising edge of clock A to get through latch C before its clock falls. Note that any required underlaps will add to this period, since the clock width cannot be shortened without either lengthening the period or causing the signal to miss the falling edge at latch C. This solution would be unacceptable for Jouppi's original problem, as the clock high times were limited to 12ns.

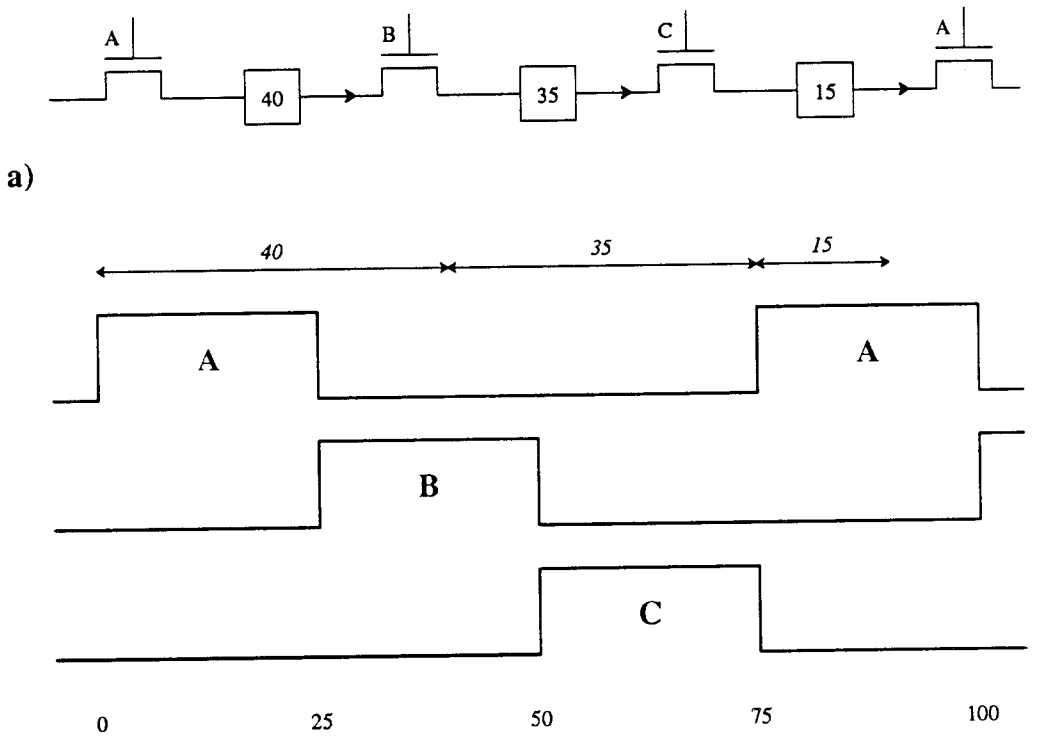


Figure 2-10: SCAT clock period computation. (a): Example with delays; (b): Computed clocks.

7. OTHER TIMING VERIFICATION ISSUES

Two other problems related to timing verification are how to verify designs that are too large to fit in memory at one time, and how to detect possible false paths (other than by letting the user identify them manually). Each of these is discussed briefly below.

7.1. Verifying Large Designs

One problem with verifying large systems is that the entire design may be too large to verify in a single run. The usual solution, adopted by the SCALD timing verifier, is to require the user to partition the system manually into units to be separately verified, and to supply timing assertions about all the signals crossing unit boundaries. Frisiani and Roth [FrR80, FrR81, FrR83, RoF80], however, proposed a

way to perform the analysis of paths between units (which they call T-units) automatically, only specifying assertions for primary inputs and outputs of the overall system. The basic technique they used is to verify each T-unit with whatever inputs had been specified or calculated to date, setting the arrival time of all other inputs to "unknown." The system would then calculate times for all outputs that could be calculated from the available information and move on to the next T-unit, iterating until the process converged.

7.2. False Path Detection

Benkoski *et al.* [BMC87] have recently proposed a new approach to detecting and eliminating "false paths," by attempting to sensitize the path. Unfortunately, not every non-statically-sensitizable path is a false one,* hence their approach is flawed. The example in Figure 2-11 illustrates the problem. The and-or-invert gate shown is being used as a mux. Suppose that inputs A and B are computed by complicated logic functions that just happen to be identical. In that case, neither path from SEL to the output will be statically sensitizable, since this would require that A and B assume different values.† However, because of the different delay through the two paths (through/ not through the inverter), if A and B are both high, there will be a momentary glitch in the output when SEL goes from high to low. If the data were sampled during this glitch, it would be incorrect. Thus this is really a true path that must be accounted for in computing system timing, even though it is not sensitizable.

*My thanks to Albert Wang for this observation.

†In order to sensitize the upper path, A must be set to its non-controlling value of 1, but the lower input to the OR must also be set to its non-controlling value of 0, which requires that B be set to 0. Similarly A must be set to 0 and B to 1 to sensitize the lower path.

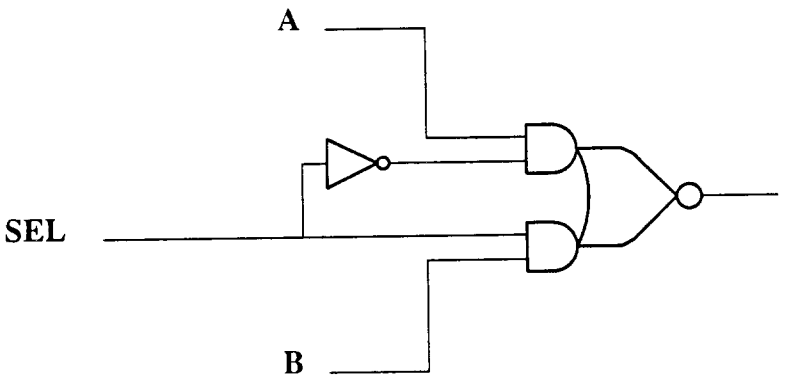


Figure 2-11: The path from SEL to the output is not sensitizable if A=B, but it may still be critical.

8. SUMMARY

Previous timing verifiers have used many representations for time and delay. Times when events occur have been represented as single numbers, as min-max ranges, as statistical descriptions (either mean and standard deviation, or other probability distributions), and as electrical waveforms. Orthogonally to this choice of representation, most representations have been used both directly and as separate rise and fall times in different timing verifiers. Delays too have taken many forms; generally similar to the representation used for time, but sometimes quite different (such as min-max delays in SCALD, or transistor structures in the switch-level timing verifiers). Whatever the representation of time and delay used, however, it has generally been built into the program at a deep level. Although a few programs have offered both min-max and statistical representations, these have not generally been provided as genuine alternatives. Rather, either one form has been coerced into the other for analysis (PERT) or one has been used in a purely supplemental role, such as performing statistical computations on a few paths identified through a min-max analysis.

None of the previous algorithms for analyzing clock phase lengths in the presence of transparent latches are fully satisfactory. They have exponential time complexity (MOTIS, TV), require extensive manual intervention (Crystal), or restrict the potential degrees of freedom in the clock schedule (MOTIS, SCAT).

In ATV, many representations can be expressed in terms of a single abstract model, allowing the user to choose one of several different timing models to be plugged in to a common framework. A new algorithm for transparent latch analysis is used that has polynomial time complexity (effectively linear in the size of the graph), allows each clock phase length and underlap to vary independently, and (with the addition of a subsequent linear programming step) will not require significant user intervention. The abstract timing model and the implementation, including the analysis algorithm, will be presented in the next two chapters.

Chapter 3 - The Abstract Timing Model

1. THE ABSTRACT TIMING MODEL

The abstract timing model operates on a dependency graph derived from the logic design to be verified, as shown in Figure 3-1. Each node in this directed graph is a logical node with no internal delays -- if the interconnect delay between two or more points in the logic diagram is significant, each such point must be modeled as a separate node in the dependency graph. Arcs run from each node to those it can directly affect. Each arc may have an associated delay. The format used to represent delays is defined by the particular timing model used.

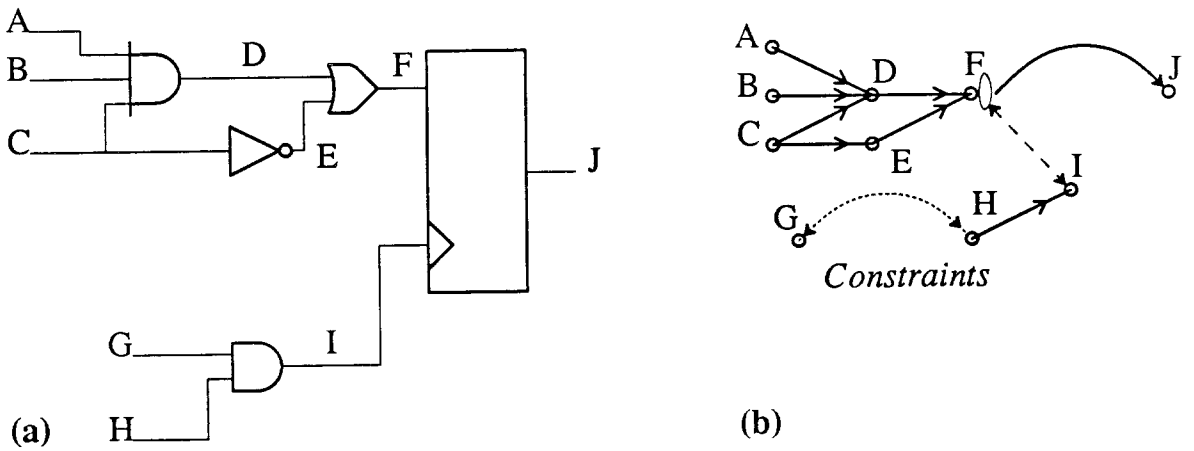


Figure 3-1: (a): Logic diagram. (b): Corresponding dependency graph.

This description of the abstract timing model leaves the precise details of how the dependency graph and its associated delays are derived from the logic design free as an implementation issue. Thus in Figure 1, if the wire labeled "D" had a significant delay associated with it, there would be several options for representing this delay in the graph. First, the entire delay could be included in the arc from D to F, identifying node D with the beginning of the wire. Second, the entire delay could be included in each of the arcs from A, B, and C to D, identifying node D with the end of the wire. Third, the delay could be split between these groups of arcs, identifying node D with a point along the wire. Fourth, node D could be split into two nodes, D1 and D2, identified with opposite ends of the wire, thus allowing the

interconnect delay to be represented explicitly in the graph by an arc from D1 to D2, rather than being combined with gate delays. Finally, at the other extreme, if the internal structure between inputs A, B, and C and node F is unimportant, this entire section of the graph could be reduced to three arcs, one from each input to node F, eliminating node D from explicit consideration.

Most existing systems use one of three approaches to computing worst-case signal arrival times: a) depth-first search without pruning, b) depth-first search with pruning, or c) breadth-first search with pruning. Depth-first search without pruning, also known as *path enumeration*, traces all possible paths through the graph, summing the total delay along each path. Depth-first search with pruning adds an optimization: whenever a node is revisited in the course of the search, the new arrival time is only propagated to the child nodes if it is worse than the previous worst arrival time at that node. This approach is used in Crystal [Ous85]. Both of these approaches can have exponential time complexity in the number of circuit components, although the worst-case graphs do not often arise in practice. Breadth-first search with pruning, also known as the *block-oriented* [Hit82] or *critical path* method [BLA82], requires an acyclic graph. Here a node is not evaluated until all of its possible predecessors have been evaluated. The worst case arrival times from each predecessor produce a worst case time that is then propagated to the children of the node. This approach has linear time complexity. The complexity of these algorithms does not depend on the specific timing model used.

The abstract timing model is based upon a) abstract *event times*, which are descriptions of *when* events happen at nodes of the dependency graph, b) abstract *delays*, associated with arcs of the graph, and c) four operations (discussed below) that are performed on event times: *translating*, *delaying*, *merging*, and *comparing*. An event time generally consists of some numbers describing when an event occurs and the uncertainty (of various kinds) surrounding such an event, plus additional information that may contribute to delay calculations. Such additional information may include information about the history of the signal, allowing for non-local events to affect the delay of the signal. Alternatively, an event time may consist of a stream of values, as in SCALD, where signals are described by a series of timing verifier states, such as *stable* or *changing* [McW80a], and a time associated with each state that describes when

the signal assumes that value. This representation may cause difficulties for the compare operation, and it may be better to model such a series of values explicitly as a series of events.

Every timing model defines a minimum of four operations that apply to event-times in that model. These are:

(1) *translating* an event-time forwards or backwards in time by some real number,

(2) *delaying* an event-time by some delay along an arc of the timing graph,

(3) *merging* multiple event-times at a node to produce a "worst-case" event-time at that node (when there are multiple inputs to the node), and

(4) *comparing* two event-times.

1.1. Translating

The translation operation takes an event-time $\{E\}$ and shifts it in time by some real number t, producing a new event-time $\{E\}+t$. Unlike the delay operation, it does not transform the internal structure of the event-time, and is always invertible (translating by $-t$). Operations 2-4 above are all time-invariant with respect to the translation operation: if the inputs to a series of delays, merges, and comparisons are all translated by the same amount, t, the result will be the same as performing the operations on the untranslated inputs and then translating any output results by t. No such assertions can be made about the generic delay operation: there exist reasonable examples of timing models for which these properties do not hold for the delay operation. Without the translation operation it would be difficult or impossible to define critical paths and slacks in the general case, or to interpret constraints on events deep within the timing graph in terms of the original inputs. Thus we decided to include translation in the fundamental operations of our abstract model.

The notation $E+a$, for scalar a and event time E, will be used to represent the translation of E by a.

1.2. Delaying

The generalized delaying operation takes an input event time, E_i, and delays it by some delay D to produce an output event time, E_o. In discussions of the delaying operation, the subscripts i, o, and d will refer to elements of the input time, output time, and delay, respectively. A notation often used for this delay operation will be:

$$E_o = E_i \oplus D$$

This delay operation is left-associative:

$$E \oplus D_1 \oplus D_2 \triangleq (E \oplus D_1) \oplus D_2$$

The delaying operation is time-invariant with respect to the translation operation; we always have:

$$(E_i + t) \oplus D = (E_i \oplus D) + t$$

for any event time E_i, delay D, and real number $t \in \mathbb{R}$.

In principle, an abstract delay can be a pointer to almost anything (an arbitrary data structure, a pointer into a transistor network, data mingled with executable code) as long as the delaying routine can apply it to event times. Most timing models use a simple structure of numbers to describe delays. For many models, the representation for delays is the same as for event times. Many previous authors who worked with such models did not distinguish between the two concepts. Event times are associated with the nodes of the dependency graph; delays with the arcs. The two representations need not be the same.

The general rule for timing models is that the type of the event-time determines the model to use. Thus the delay operation for a given type of event-time will attempt to coerce whatever delay information is available into the appropriate form. There may be more than one delay format specified for a given arc. The general idea is to allow the delay to be specified in its natural form, and to use coercions to obtain the format needed by a particular timing model.

1.3. Merging

The merge operation supports algorithms that include pruning. It applies when a node in the dependency graph has two or more potentially active predecessors. It produces an event time describing the "worst case" among its input event times. For single transitions, this means taking two or more event times that could affect the output and producing an event time that describes the latest (or occasionally the earliest) event that could affect the output. Sometimes we can do both at the same time (see discussion of min-max model). For simplicity, the effects of delay have been separated from the effects of merging. This allows for different delays from each input to a given output, but does mean that each delay is independent of what may be happening to other inputs. In this paper, the formal definition of the merge operation is based on only two inputs. Merging of three or more inputs can be done two at a time, as long as there is no dependency between inputs. If the merge operation is defined to be commutative and associative, the order in which inputs are combined is irrelevant. In an actual system, the abstract merge operation may be defined on an arbitrary number of inputs, allowing models to consider more complex interactions between inputs than pairwise combination would allow. The merge of several event times will often be represented as $M(E_1, E_2, ...)$.

1.4. Comparing

Delaying and merging produce a description of what is happening in the system over time. Comparing is the operation that tests the resulting timing behavior against constraints or expectations. Node event times may be compared to each other directly (signal A must arrive before signal B), to each other after some delay (such as checking set-up and hold times at latches), or to prior expectations (requirements on system outputs). In general, there are several possible relations that could hold between event times:

$=$ The equal relation means that the two event times are identical.

$\approx$ The fuzzy equal relation means that the two event times are equivalent within some user-specified tolerance. It is used in the generic definition of critical inputs.

< $\{E_1\}<\{E_2\}$ means that an event occurring at event time $\{E_1\}$ is "guaranteed" to occur before an event occurring at event time $\{E_2\}$ (within the model-dependent interpretation of what "guaranteed" means). If $\{E_1<\{E_2\}$, then $\{E_1\}<\{E_2\}+t$, for all $t \geq 0$.

≤ $\{E_1\}\leq\{E_2\}$ means that an event occurring at event time $\{E_1\}$ is "guaranteed" to occur no later than an event occurring at event time $\{E_2\}$ (within the model-dependent interpretation of what "guaranteed" means). If $\{E_1\leq\{E_2\}$, then $\{E_1\}\leq\{E_2\}+t$, for all $t \geq 0$.

> $\{E_1\}>\{E_2\}$ means that an event occurring at event time $\{E_1\}$ is "guaranteed" to occur after an event occurring at event time $\{E_2\}$ (within the model-dependent interpretation of what "guaranteed" means). If $\{E_1>\{E_2\}$, then $\{E_1\}>\{E_2\}+t$, for all $t \leq 0$.

≥ $\{E_1\}\geq\{E_2\}$ means that an event occurring at event time $\{E_1\}$ is "guaranteed" to occur no sooner than an event occurring at event time $\{E_2\}$ (within the model-dependent interpretation of what "guaranteed" means). If $\{E_1\geq\{E_2\}$, then $\{E_1\}\geq\{E_2\}+t$, for all $t \leq 0$.

Incomparable

Two event times are *incomparable* if no more specific relation holds between them.

Because many of the above relations overlap (e.g., < and ≤), it is not possible to ask for the unique relation that holds between two event times. Instead, a function called *Satisfies-p* is defined, which tests whether a specified relation holds between two different event times. The current implementation of ATV really only requires the results ≤, ≥, **Incomparable**, and possibly ≃: the others are defined above for completeness and for possible use by future versions of the program.

The above relations may be further extended by individual models, e.g., by adding an indication of how likely one event time is to exceed another, for probabilistic models.

2. SINGLE-NUMBER TIMING MODEL

This is the first of many timing models to be discussed in detail. It has the simplest structure of all the timing models to be discussed.

2.1. Model Data Definitions

$$E=[t]$$

$$D=[d]$$

The time number represents the time at which an event happens. The delay number represents the delay of a block. The notation $E[t]$ is used below to mean "the event time at time t," the notation $D[d]$ is used to mean "the delay with value d."

2.2. Translation Operation

The translation operation for single numbers is just real-number addition applied to the event times:

$$E[e]+t=E[e+t]$$

2.3. Delay Operation

The subscripts i, o, and d refer to elements of the input time, output time, and delay, respectively. The delay operation for single numbers is identical to the translation operation, except for the type of the second operand (delay vs. real number).

$$E[e]+D[d]=E[e+d]$$

2.4. Merge Operation

The definition of the merge operation is also straightforward for this model. The earliest time an output transition can occur is the earlier of the input times; the latest an output transition can occur is the latest of the input times:

$$t_o=\max(t_1,t_2)$$

2.5. Compare Operation

The compare operation compares two event times, E_1 and E_2. For this model, real number ordering is used on the respective event times.

3. [MIN, MAX] TIMING MODELS

This is one of several families of timing models to be discussed in detail. All members of the "min-max" family use the same mathematical structure for the timing model, but may differ in its interpretation.

3.1. Model Data Definitions

$$T=[\min_t, \max_t]$$

$$D=[\min_d, \max_d]$$

The time numbers represent the minimum and maximum times at which an event can happen. The delay numbers represent the minimum and maximum delays of a block.

3.2. Delay Operation

The subscripts i, o, and d refer to elements of the input time, output time, and delay, respectively. The delay operation is defined in the expected way:

$$\min_o = \min_i + \min_d$$

$$\max_o = \max_i + \max_d$$

3.3. Merge Operation

The definition of the merge operation is also straightforward for this model. The earliest time an output transition can occur is the earlier of the two minimum input times; the latest an output transition can occur is the later of the two maximum input times:

$$\min_o = \min(\min_1, \min_2)$$

$$\max_o = \max(\max_1, \max_2)$$

3.4. Compare Operation

The compare operation compares two event times, T_1 and T_2. The proper definition of this operation is less obvious than the others. The description below defines two intervals to be comparable only if they are either totally disjoint or completely equal. Thus as one interval shifts relative to another of equal length, the result of the comparison will go through the sequence: <, incomparable, =, incomparable, >. The resulting discontinuities are undesirable.

$$\text{compare}(T_1,T_2) = \begin{cases} < & \text{if } \max_1 < \min_2 \\ = & \text{if } \min_1 = \min_2 \text{ and } \max_1 = \max_2 \\ > & \text{if } \min_1 > \max_2 \\ \text{incomparable} & \text{otherwise} \end{cases}$$

Another approach would be to define results ≤ and ≥, to cover comparisons that do not involve one interval completely covering another. This avoids the above discontinuities, but extends the interpretation of the compare command in a way that may be model-dependent.

3.5. Data Interpretation

The usual interpretation of event times in the min-max model is that there are two underlying logic states, 0 and 1, and a single logic threshold between them. The minimum represents the earliest time that a node can make a transition between states; the maximum represents the latest time. An alternate

interpretation assumes that there are three logic states: 0, 1, and X, with two logic thresholds. Signals going from one stable state (0 or 1) to the other must pass through the intermediate X state first. In this interpretation, the minimum represents the earliest time that the node may make a transition from a stable state to the X state if a transition is going to occur; the maximum represents the latest time that the node can make a transition out of the X state into one of the two stable states. A pictorial representation of the two interpretations is given in Figure 3-2.

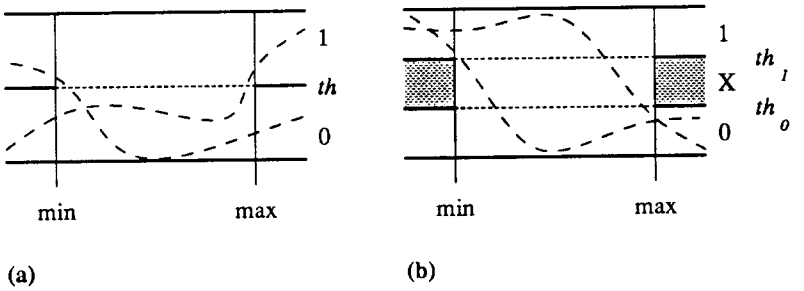


Figure 3-2: Interpretations of event times in the min-max model. Solid lines and shaded areas are barriers the signal may not cross. (a): two-state interpretation; (b): three-state interpretation

Both interpretations produce the same desired mathematical structure for the delaying, merging, and comparing operations; hence both are equally valid interpretations of the model. However, the delay values computed for a particular logic block under the two interpretations will be different and will depend on the choice of logic thresholds. The two-state interpretation is a degenerate case of the three-state interpretation, where $th_1=th_0=th$.

3.6. An Example: Deriving Delays for an Inverter

How are the delay parameters for this model derived from underlying circuit simulation data? As an example, consider deriving delay parameters for a single inverter from SPICE [NaP73] simulation data in the above three-state interpretation. The uncertainty to be modeled is the uncertainty about the initial logic state and the input waveform. For simplicity, we will assume that there is no uncertainty about the device parameters, the output loading, or the internal structure of the inverter. To model these

uncertainties as well, we would make best-case (fastest response) assumptions when calculating the minimum delay, and worst-case (slowest response) assumptions when calculating the maximum.

Since the initial and final states of the inverter are unknown, we compute the delay separately for a rising output and a falling output. These delays will either be merged together to form the overall delay, or maintained separately for asymmetric models as discussed in Section 7. Only the case of a falling output will be considered here; the other case is similar. To keep the modeling effort reasonable, we set the input event time to (0, 0), and determine the range of output event times generated by all possible inputs that meet the input event time specifications. The extremes of the range of output times are the min and max values for the delay. Figure 3-3 shows the input waveforms producing the minimum and maximum delays for the falling output case and the corresponding min/max bounds.

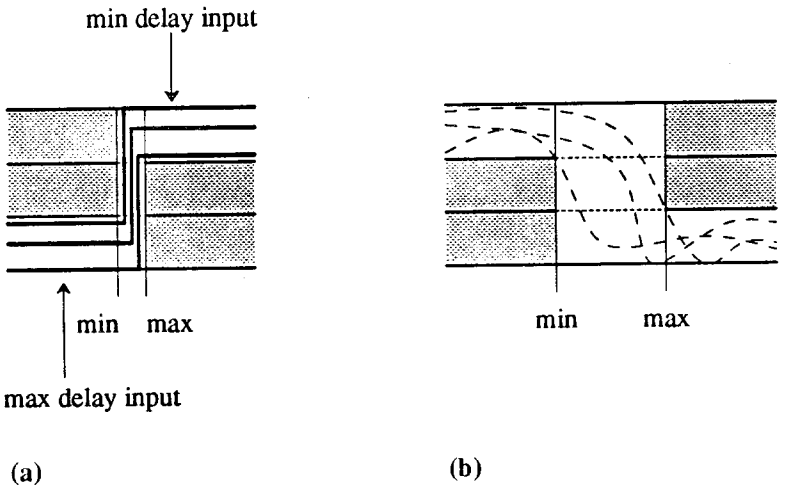


Figure 3-3: Diagrams of input and output signals for an inverter with a falling output. Three-state interpretation of (a): Input event time; (b): Output event time.

Table 3-1:
Inverter Delay Bounds for Different Logic Thresholds (ns)

Thresholds (v)		Rising Output		Falling Output	
th_0	th_1	$\min_d$	$\max_d$	$\min_d$	$\max_d$
2.19	2.20	2	805	1	1065
2.1	2.3	26	353	5	372
1.5	3.5	50	218	8	60
1.0	4.0	29	212	6	50
0.5	4.5	9	269	4	51
0.25	4.75	1	333	2	73

Table 3-1 shows some SPICE results for the delays of an nMOS inverter with different logic thresholds. The inversion point was at 2.1965 volts. For a two state model no time bounds can be determined, since the input signal could cross the inverter threshold infinitely slowly, producing infinite delay in the output. The three-state model forces the signal to cross the inverter threshold with explicit margins. As these margins increase, the observed delays get shorter. However as th_1 and th_0 approach the potentials of the supply voltages, the max delay increases again, because the output signal will only asymptotically approach the supply line voltages.

It is apparent from the table that the minimum and maximum delays are at least an order of magnitude apart. Uncertainty about device parameters, output loading, or internal structure of the inverter would only increase this difference even further. Without making more restrictive assumptions about the possible shapes of input waveforms, the min-max model will not yield better than an order of magnitude estimate of the delays along some path.

4. [MEAN, STANDARD DEVIATION] TIMING MODELS

Although the min-max model may suffice if only gross bounds on uncertainty are known or desired, a probabilistic model is more appropriate if more detailed information is available about the distribution of delays within their possible range. If two numbers must suffice to represent such a distribution, the mean and standard deviation are generally chosen, because they have properties that are independent of the actual distribution. Most previous work [Hit82, KiC66] has assumed that delays and event times are normally distributed. Members of this family of timing models have a common delay

operation, but may have different merge and compare operations.

4.1. Model Data Definitions

$$T=(\mu_t, \sigma_t)$$

$$D=(\mu_d, \sigma_d)$$

Both event times and delays are represented statistically by a mean, μ, and a standard deviation, σ. In addition, different models may have various global parameters. A model like that used in the TA program [Hit82] has three global parameters. The first parameter, β, represents the desired confidence level for merge and compare operations. It is the number of standard deviations away from the mean used to characterize the distribution. Thus when $\beta=3$, distributions are merged and compared based on their 3-sigma points. This parameter is strictly non-negative. The second parameter, p, specifies whether minimal length paths ($p=-1$) or maximal length paths ($p=1$) are being examined. The third parameter, ρ, is an assumed correlation between delays. For the two-number models, ρ can only have the values 0 or 1.

4.2. Delay Operation

For all probabilistic models,

$$\mu_o=\mu_i+\mu_d.$$

If all delays have correlation ρ, the standard deviation of an event time along a path is given by:

$$\sigma_o=\sqrt{\sigma_i^2+2\rho s_i \sigma_d+\sigma_d^2},$$

where s_i is the sum of all standard deviations along the path up to the input event time. This would lead to a three number model with event times being modeled as

$$T=(\mu_t, \sigma_t, s_t).$$

In two special cases, the s_t term may be dispensed with: when $\rho=0$, where the value of s_t is irrelevant, and when $\rho=1$, where $s_t=\sigma_t$. For these cases, we have the simpler form:

$$\sigma_o = \sqrt{\sigma_i^2 + 2\rho\sigma_i\sigma_d + \sigma_d^2}.$$

This form is used for two number models.

4.3. Compare Operation

The TA-like model performs comparisons based on one point in each distribution (often a 3-sigma point).

$$\text{compare}(T_1, T_2) = \begin{cases} < & \text{if } \mu_1 + p\,\beta\sigma_1 < \mu_2 + p\,\beta\sigma_2 \\ = & \text{if } \mu_1 + p\,\beta\sigma_1 = \mu_2 + p\,\beta\sigma_2 \\ > & \text{if } \mu_1 + p\,\beta\sigma_1 > \mu_2 + p\,\beta\sigma_2 \end{cases}$$

With this definition, comparing is simple to evaluate, but differs from the intuitive notion of comparison described in Section 2.3. For example, for β=3 and p=1, the distribution (0.0, 2.0) will be reported as "greater" than the distribution (5.9999, 0.0), even though a random point drawn from the first distribution has more than a 99% probability of being less than a random point drawn from the second. This definition also implicitly assumes that the distributions are normal. A more general definition of comparison would be that one distribution is greater than another if with some probability c (the desired confidence level), a number randomly drawn from the first is greater than a number randomly drawn from the second. This definition allows distributions other than the normal distribution to be used.

4.4. Merge Operation

In the TA-like model, the merge operation chooses one of the two input event times based on the type of path search and on the results of a comparison between the event times:

$$\text{merge}(T_1, T_2) = \begin{cases} T_1 & \text{if } T_1 \geq T_2 \text{ and } p = 1 \text{ or if } T_1 \leq T_2 \text{ and } p = -1 \\ T_2 & \text{if } T_1 < T_2 \text{ and } p = 1 \text{ or if } T_1 > T_2 \text{ and } p = -1 \end{cases}$$

There are several difficulties with this definition of the merge operation. First, small changes in one input can produce large discontinuities in the output. For example, for β=3 and p=1, the merge of (0.0, 2.0) with (5.9999, 0.0) is (0.0, 2.0), but the merge of (0.0, 2.0) with (6.0001, 0.0) is (6.0001, 0.0).

Here a small change in the mean of one distribution results in a major change in both the mean and standard deviation of the result. Given two distributions such as those in Figure 3-4, either one could be the merged result, depending on the choice of comparison points.

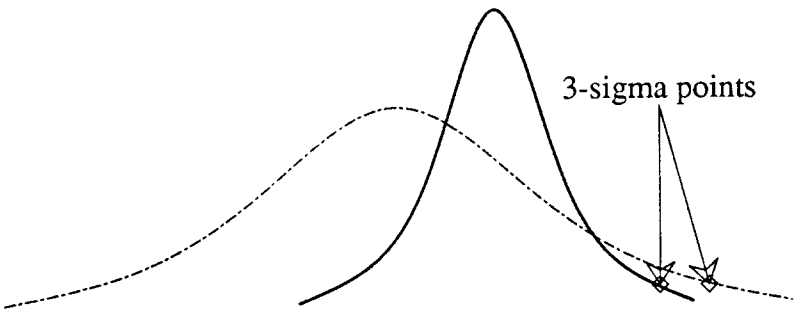


Figure 3-4: Two input distributions to be merged or compared.

Another problem is that this definition of the merge operation does not accurately describe the distribution of the "worst-case" input merge time. The worst-case distribution (for maximal length paths) is the distribution of the maximum of two random variables, one drawn from each input distribution. This problem was discussed by Shelly and Tryon [ShT83], who described the difference between the 3-sigma point computed by T.A. and the *d-point* (the point whose cumulative distribution is equal to that of the 3-sigma point of a normal distribution, .9986501) of the actual distribution. The method for calculating the d-point of the actual distribution is described in [BCS84].

Because the maximum of two normal distributions is typically not normal, there are several ways to compute the mean and standard deviation of the implied normal distribution representing the merged event time. One approach is to compute the mean and standard deviation of the actual distribution. Another approach is to compute the "best fitting" normal distribution. A third approach is to constrain the result so that the 3-sigma point of this distribution is equal to the d-point of the actual distribution, and select among all such distributions as in one of the two previous approaches. Tables 3-2 to 3-4 (which are extensions of tables presented in Shelly and Tryon) compare the results of the three approaches described above (called "moments," "best fit," and "d-point," respectively) with the results given by

the simple TA model. Table 3-2 shows how the maximum is affected by the number of input distributions: it shows the results of merging n identical independent input distributions together, for values of n ranging from 1 to 15. Table 3-3 shows how the maximum is affected by the correlation between distributions: it shows how the merge of two identical input distributions is affected as the correlation between them ranges from 0.0 (independent distributions) to 1.0 (perfectly correlated). Table 3-4 shows how the maximum is affected by the skew between distributions: it shows how the merge varies for two independent input distributions with the same standard deviation with six different time offsets ranging from 0.0 to 1.0 standard deviations.

Table 3-2: Distribution of max of n N(0,1) Independent Random Variables

n	μ	σ	3σ point	Fit Index	method
1	0.000	1.000	3.000	1.000	TA approx.
	0.000	1.000	3.000	1.000	best fit
	0.000	1.000	3.000	1.000	moments
			3.000		d-point
2	0.000	1.000	3.000	.750	TA approx.
	0.535	0.819	2.991	.987	best fit
	0.564	0.826	3.041	.982	moments
			3.205		d-point
3	0.000	1.000	3.000	.616	TA approx.
	0.803	0.739	3.020	.980	best fit
	0.846	0.748	3.090	.973	moments
			3.320		d-point
4	0.000	1.000	3.000	.528	TA approx.
	0.982	0.692	3.059	.975	best fit
	1.029	0.701	3.133	.966	moments
			3.399		d-point
5	0.000	1.000	3.000	.465	TA approx.
	1.107	0.657	3.077	.972	best fit
	1.162	0.669	3.170	.962	moments
			3.460		d-point
10	0.000	1.000	3.000	.303	TA approx.
	1.474	0.574	3.196	.962	best fit
	1.538	0.587	3.299	.948	moments
			3.642		d-point
15	0.000	1.000	3.000	.231	TA approx.
	1.666	0.524	3.238	.957	best fit
	1.735	0.549	3.382	.941	moments
			3.745		d-point

Table 3-3: Distribution of max of 2 N(0,1) Correlated Random Variables

ρ	μ	σ	3σ point	Fit Index	method
0.0	0.000	1.000	3.000	.750	TA approx.
	0.535	0.819	2.991	.987	best fit
	0.564	0.826	3.041	.982	moments
			3.205		d-point
0.2	0.000	1.000	3.000	.782	TA approx.
	0.486	0.861	3.067	.992	best fit
	0.504	0.863	3.095	.989	moments
			3.204		d-point
0.4	0.000	1.000	3.000	.816	TA approx.
	0.425	0.898	3.118	.995	best fit
	0.437	0.809	3.135	.949	moments
			3.202		d-point
0.6	0.000	1.000	3.000	.853	TA approx.
	0.350	0.934	3.152	.998	best fit
	0.356	0.873	3.159	.967	moments
			3.193		d-point
0.8	0.000	1.000	3.000	.898	TA approx.
	0.250	0.968	3.152	.999	best fit
	0.252	0.968	3.155	.999	moments
			3.166		d-point

Table 3-4: Distribution of max of a N(0,1) and an N(DIFF,1) Random Variable

DIFF	μ	σ	3σ point	Fit Index	method
0.0	0.000	1.000	3.000	0.750	TA approx.
	0.535	0.819	2.991	0.987	best fit
	0.564	0.826	3.041	0.982	moments
			3.205		d-point
0.2	0.200	1.000	3.200	0.789	TA approx.
	0.640	0.821	3.102	0.987	best fit
	0.670	0.828	3.153	0.982	moments
			3.321		d-point
0.4	0.400	1.000	3.400	0.823	TA approx.
	0.755	0.826	3.233	0.986	best fit
	0.786	0.834	3.289	0.981	moments
			3.466		d-point
0.6	0.600	1.000	3.600	0.854	TA approx.
	0.876	0.838	3.389	0.985	best fit
	0.914	0.844	3.447	0.979	moments
			3.633		d-point
0.8	0.800	1.000	3.800	0.882	TA approx.
	1.015	0.850	3.566	0.983	best fit
	1.052	0.857	3.624	0.978	moments
			3.816		d-point
1.0	1.000	1.000	4.000	0.905	TA approx.
	1.156	0.866	3.755	0.982	best fit
	1.199	0.872	3.815	0.977	moments
			4.007		d-point

Only the 3-sigma point is given for the d-point method, because this is really a family of methods with the common property that the 3-sigma point is equal to the d-point of the exact distribution. The *fit index* is the area of overlap between the density function of the true distribution and the density function of the approximation. It ranges from 0 (for no overlap at all) to 1 (for a perfect fit). This is the function maximized by the best fit method. Both the mean and standard deviation of the TA approximation differ significantly from those of the actual distribution (shown by the moments method), but the direction of the errors differ, so they tend to cancel out in the computation of the 3-sigma point. In Table 3-4, the 3-sigma point of the TA approximation is often closer to the d-point of the actual distribution than either the best fit or the moments approximations, but this is not true in general (compare Tables 3-2 and 3-3, for example).

Although the TA approach introduces significant distortions, it is easier to work with than any of the other approaches. Both the basic merge operation and *local slack computation* along sub-critical

paths are easier to compute in the TA model. Because the TA approach effectively reduces distributions to a single point for merging and comparing, it is easy to compute local slack as the difference between two event times. In the more accurate approaches, both input event times influence the final result (except when one event time is much larger than the other). Thus it is more difficult to compute slack in these cases. This will be discussed more fully in the section on slack and critical paths.

4.5. Data Interpretation

The interpretation of the delays is similar to the 3-level interpretation of the [min, max] model. The delay is the time measured from when the input enters its final state to when the output enters its final state.

To derive accurate parameters for the statistical model, the user can either rely on measurements of actual designs, or on monte-carlo simulation. In the former case, the user would start with a sufficiently large number of blocks similar to the one being modeled, and compute statistics. In the latter, the user would start with assumed distributions for the underlying parameters, and then run a series of simulations with parameters chosen at random from these distributions.

4.6. Correlation Classes

In general, to add two normal distributions together, we need to know the correlation between them. The general rules are:

$$\mu_{A+B}=\mu_A+\mu_B$$

$$\sigma^2_{A+B}=\sigma^2_A+2\sigma_A\,\sigma_B\,\rho_{AB}+\sigma^2_B,$$

where ρ_{AB} is the correlation between A and B. This expression generalizes like a multinomial expansion, thus if $X=\sum_{i=1}^{n} x_i$, μ_i is the mean of x_i, σ_i is the standard deviation of x_i, and ρ_{ij} is the correlation between x_i and x_j, then

$$\mu_X=\sum_{i=1}^{n}\mu_i$$

$$\sigma_X^2=\sum_{i=1}^{n}\sum_{j=1}^{n}\sigma_i\sigma_j\rho_{ij}, \qquad \text{(I)}$$

where of course, $\rho_{ii}=1$ for all i. Thus if we knew the correlation between any two delays in a path, we could calculate the overall standard deviation along that path.

Now it is clearly impractical to specify a correlation between every pair of delays in a given design, since there are $O(n^2)$ such pairs, where n is the number of delays. To reduce the number of correlations that have to be specified, we define a fixed number of *correlation classes*. A correlation coefficient is specified for each pair of correlation classes; if the correlation between classes A and B is ρ_{AB}, then for any pair of delays with one a member of class A and the other a member of class B, ρ_{AB} is the correlation between this pair of delays. Note that ρ_{AA} need not be 1; however, it is still true that the correlation between a delay and itself is still 1. This is the one exception to the rule about correlations between delays. The correlation matrix is symmetric: $\rho_{AB}=\rho_{BA}$.

Depending on the number of correlation classes we choose, we can have everything between a global correlation for all delays (only one correlation class) at one extreme, to individual correlations between all delays at the other extreme (a correlation class for every delay). Most applications will define just a few correlation classes, corresponding, say, to individual chips or regions on a chip.

Now in this correlation class model, the individual x_i's correspond to the individual delays along a path, while random variables such as X and Y correspond to event times that are summing the individual delays along that path. In order to define the delay operation for these correlation classes, we need to first derive the standard deviation of the sum of n x_i's (the output event time) in terms of the standard deviation of the sum of the first $n-1$ x_i's (the input event time), the standard deviation of the x_n (the delay), and some miscellaneous cross terms (other information that also needs to be attached to the input event time).

Because the correlation of any x_i with itself is 1, and all other correlations are symmetric, equation I can be rewritten:

$$\sigma_X^2=\sum_{i=1}^{n}\sigma_i^2+\sum_{i=1}^{n-1}\sum_{j=i+1}^{n}2\sigma_i\sigma_j\rho_{ij}. \qquad \text{(II)}$$

If $Y=\sum_{i=1}^{k} x_i$ and $Z=\sum_{i=k+1}^{n} x_i$ represent a partition of the sum of the random variables x_i into the first k and the remaining $n-k$, then equation II can be regrouped to refer only to the standard deviations of each partition plus the cross terms between the partitions:

$$\sigma_X^2=\sigma_{Y+Z}^2=\sigma_Y^2+\sigma_Z^2+\sum_{i=1}^{k}\sum_{j=k+1}^{n} 2\sigma_i\sigma_j\rho_{ij}. \tag{III}$$

In particular, when $k=n-1$ (so $Z=x_n$):

$$\sigma_X^2=\sigma_{Y+Z}^2=\sigma_Y^2+\sigma_n^2+\sum_{i=1}^{n-1} 2\sigma_i\sigma_n\rho_{in}. \tag{IV}$$

Now if there are a total of m correlation classes, C_1 through C_m, and $C(i)$ represents the correlation class of x_i, then we can regroup the last set of terms to collect terms from the same correlation class together:

$$\sigma_X^2=\sigma_{Y+Z}^2=\sigma_Y^2+\sigma_n^2+\sum_{l=1}^{m}\sum_{\substack{x_i\in C_l \\ 1\le i<n}} 2\sigma_i\sigma_n\rho_{C_lC(n)} \tag{V}$$

$$=\sigma_Y^2+\sigma_n^2+\sum_{l=1}^{m} 2\sigma_n\rho_{C_lC(n)}\sum_{\substack{x_i\in C_l \\ 1\le i<n}}\sigma_i,$$

where $\rho_{C_lC(n)}$ is the correlation between correlation class l and the correlation class of x_n. Now σ_Y^2 is the variance of the input event time, and σ_n^2 is the variance of the delay, so equation (V) demonstrates that in order to perform the delay operation, all we need is the standard deviation of the input event time together with a term for each correlation class consisting of the sum of the standard deviations of all the constituent delays that fall in that class. Using τ_i to represent this sum for correlation class i, we can now define the correlation class model.

4.6.1. Model Data Definitions

$$E=(\mu_e, \sigma_e, \tau_1,\ldots, \tau_m)$$

$$D=(\mu_d, \sigma_d, k)$$

Both event times and delays are represented statistically by a mean, μ, and a standard deviation, σ. Event times have m terms for the m correlation classes; delays have one variable k, $1 \le k \le m$, the index of the correlation class for this delay. There is a m by m correlation matrix, ρ_{ij}. Other global parameters are the same as for the simple mean, standard deviation model.

4.6.2. Translation Operation

As in the simple mean, standard deviation model, the translation operation just adds the translation amount to the mean of the event time:

$$\mu_o = \mu_i + t$$

4.6.3. Delay Operation

$$\mu_o = \mu_i + \mu_d .$$

$$\sigma_o^2 = \sigma_i^2 + 2\sum_{i=1}^{m} \rho_{ik}\tau_i\sigma_d + \sigma_d^2$$

$${}_o\tau_k = \tau_k + \sigma_d$$

$${}_o\tau_i = \tau_i , \; i \neq k ,$$

where ${}_o\tau_i$ is the ith cross-term of the output event time, τ_i is the ith cross-term of the input event time, and k is the index of the correlation class of the delay.

4.6.4. Compare Operation

The compare operation is like that of the TA-like model.

4.6.5. Merge Operation

The merge operation is like that of the TA-like model.

4.7. Weighted Discrete Probability Distribution

Roland Steck has proposed yet another probabilistic model that combines features of the min-max model and the probabilistic models [GSS86, Ste85]. He proposes a two point discrete probability distribution for delays, characterized by three numbers: a minimum value, a, a maximum value, b, and a probability p. The interpretation is that the delay takes the value b with probability p, and takes the value a with probability $q=1-p$. The variables a, b, and p are themselves derived from the more usual min, typical, and max by taking a and b to be the min and max, respectively, and deriving p from these values together with the typical value, m, via the linear interpolation formula:

$$p = \frac{m-a}{b-a}$$

(i.e., p is chosen to keep the mean of the resulting distribution equal to the specified typical value, m).

When adding two delays (which corresponds to my delaying operation), the mean and standard deviation are calculated for both input delays. With the input delay components subscripted by 1 and 2, and the output result components subscripted by s, Steck gives the following equations for calculating the appropriate parameters for the sum [Ste85]:

$$m_s = m_1 + m_2$$

$$a_s = m_s + s \cdot (a_1 + a_2 - m_s)$$

$$b_s = m_s + s \cdot (b_1 + b_2 - m_s),$$

where the "shrink factor" s referred to above is calculated from the following equations:

$$\sigma_1^2 = (b_1 - m_1) \cdot (m_1 - a_1)$$

$$\sigma_2^2 = (b_2 - m_2) \cdot (m_2 - a_2)$$

$$s^2 = \frac{\sigma_1^2 + \sigma_2^2 + 2\rho_{12}\sigma_1\sigma_2}{(\sigma_1 + \sigma_2)^2}$$

Steck inverts the signs of the right-hand sides of the first two equations above, but this would imply a negative variance, since $a_1 \le m_1 \le b_1$. Since $-1 \le \rho_{12} \le 1$ and $\sigma_1, \sigma_2 \ge 0$, the right-hand side of the last equation above takes on its maximum value when $\rho_{12} = 1$, and its minimum value when $\rho_{12} = -1$. Hence:

$$0 \le \frac{(\sigma_1 - \sigma_2)^2}{(\sigma_1 + \sigma_2)^2} \le s^2 \le \frac{(\sigma_1 + \sigma_2)^2}{(\sigma_1 + \sigma_2)^2} = 1,$$

so $0 \le s \le 1$. Intuitively, Steck is first computing a_s, m_s, and b_s as if this were a pure min-max model, with an implied correlation of 1 ($s=1$), and then shrinking a_s and b_s towards m_s so as to preserve m_s as the mean, but make the standard deviation σ_s equal to the standard deviation under a mean, standard-deviation model when the correlation is in fact ρ_{12}.

Steck gives some additional equations for the calculation of the statistical maximum of two such distributions (corresponding to my merge operation).

Steck's equations for merging (a_1, b_1, p_1) and (a_2, b_2, p_2), in the case where $b_1 > b_2$ (the other case is symmetrical), fall into three sub-cases.

Case I ($b_2 < a_1$):

$$a_m = a_1$$

$$m_m = m_1$$

$$b_m = b_1$$

Case II ($b_2 > a_1$ and $a_2 < a_1$):

$$a_m = a_1$$

$$m_m = m_1 + q_1 p_2 (b_2 - a_1)$$

$$b_m = b_1$$

Case III ($b_2 > a_1$ and $a_2 > a_1$):

$$a_m = a_2$$

$$m_m = m_1 + q_1(m_2 - a_1)$$

$$b_m = b_1$$

Steck does not seem to discuss the equivalent of my comparison operation, though there are several ways this could be done (i.e., use the comparison from the min-max model, or the TA-based model, or some appropriate combination of the two). The translation operation for the twopoint model should be self-evident.

5. ARBITRARY PROBABILITY DISTRIBUTIONS

Normal distributions are only reasonable approximations to delay and event times if the underlying expected distribution is unimodal (note that this is not a sufficient condition). However, there are circumstances when this is not an appropriate assumption. For example, suppose that we are trying to model the delay through a yet-to-be-designed adder circuit, and that there are two competing designs, corresponding to two different carry-bypass schemes. We expect one scheme to have an average delay of around 15ns, and the other to have an average delay around 25ns. If part of the uncertainty we are trying to model is the design uncertainty about which design will be chosen, it would obviously be inappropriate to model this with a single normal delay with mean 20ns: the true probability distribution is bimodal, with peaks at 15 and 25ns whose relative heights are weighted by the probability of choosing one design vs. the other. For such purposes, it might be useful to be able to model delays and event times with arbitrary probability distributions. This section describes such a model, where both delays and event times are represented as arbitrary probability density functions, assumed independent. Such a model could be implemented on top of a symbolic math manipulation system such as MACSYMA [MAC77], or based on some numerical discrete approximation representation. The basic equations would also hold for specific subsets, such as piecewise polynomial density functions, provided that such a subset is closed under the following operations, or under some approximation of these operations. Choosing the subset consisting of normal distributions, with appropriate approximations to provide closure, yields the more

advanced mean-standard-deviation model discussed earlier.

5.1. Model Data Definitions

$$E=[f(t)]$$

$$D=[\delta(t)]$$

The function $f(t)$ is the density function of the event time, with corresponding distribution function $F(t)=\int_{-\infty}^{t} f(x)\,dx$. $F(t)$ is the probability that the corresponding event occurs on or before time t. The delay function $\delta(t)$ is the density function of the delay through an arc, with corresponding distribution function $\Delta(t)=\int_{-\infty}^{t} \delta(x)\,dx$. $\Delta(t)$ is the probability that the delay through the arc is $\leq t$.

5.2. Translation Operation

$$E[f(t)]+a=E[f_o(t)]\,,\quad f_o(t)=f(t-a)$$

5.3. Delay Operation

For the delay operation, we want the density function of $e_o=e_i+d$, where e_i is a random variable from the input event time distribution, and d is a random variable from the delay distribution. This density function is given by a convolution of the density functions for the input event time and delay:

$$f(t)\oplus\delta(t)=\int_{-\infty}^{\infty} f(x)\delta(t-x)\,dx$$

5.4. Merge Operation

For long-path merges, $E_m=M(E_1,E_2)$ is the density of the random variable $e_m=\max(e_1,e_2)$. This is readily calculated via the distribution functions:

$$F_m(t)=\text{Prob}(e_1\leq t \wedge e_2\leq t)$$

$$=F_1(t)\bullet F_2(t),$$

because of the assumption that the two event time distributions are independent. Therefore

$$f_m(t)=\frac{d}{dt}F_1(t)\bullet F_2(t)$$

$$=f_1(t)\bullet F_2(t)+F_1(t)\bullet f_2(t)$$

The equations for short-path merges are analogous: here $E_m=M(E_1,E_2)$ is the density of the random variable $e_m=\min(e_1,e_2)$. Again, this can be calculated via the distribution functions:

$$F_m(t)=\text{Prob}(e_1\le t \vee e_2\le t)$$

$$=F_1(t)+F_2(t)-F_1(t)\bullet F_2(t),$$

because of the assumption that the two event time distributions are independent. Therefore

$$f_m(t)=\frac{d}{dt}F_1(t)+F_2(t)-F_1(t)\bullet F_2(t)$$

$$=f_1(t)+f_2(t)-f_1(t)\bullet F_2(t)+F_1(t)\bullet f_2(t)$$

$$=f_1(t)\bullet(1-F_2(t))+(1-F_1(t))\bullet f_2(t)$$

5.5. Compare Operation

The compare operation compares two event times, E_1 and E_2. For this model, one distribution is considered greater than another if with some probability c (the desired confidence level), a number randomly drawn from the first is greater than a number randomly drawn from the second. More formally:

$$E_1\le E_2 \text{ if and only if } \int_{-\infty}^{\infty} f_2(t)\bullet F_1(t)\,dt\ge c.$$

5.5.1. Examples

For one example of the above equation, let E_1 and E_2 both be the uniform distribution on the interval [0,1]. Intuitively, since the two distributions are the same and the distribution functions F_i are continuous, the probability that a point drawn from the first is ≥ a point drawn from the second must be .5, by symmetry and the fact that the probability the two points are equal is 0. Working through the equations, we have:

$$f_1(t)=f_2(t)=\begin{cases}1 \text{ if } 0\le t\le 1\\ 0 \text{ otherwise}\end{cases}$$

$$F_1(t)=F_2(t)=\begin{cases}0 \text{ if } t<0\\ t \text{ if } 0\le t\le 1\\ 1 \text{ if } t>1\end{cases}$$

So:

$$\int_{-\infty}^{\infty} f_2(t)\cdot F_1(t)\,dt=\int_0^1 t\,dt=\frac{t^2}{2}\Big|_0^1=\frac{1}{2},$$

as expected. So $E_1\le E_2$ (and symmetrically) if and only if $c\le.5$ (which is quite low for a confidence level).

A more interesting case has E_1 as above, but E_2 is the uniform distribution on the interval [0.5,1.5]. Then:

$$\int_{-\infty}^{\infty} f_2(t)\cdot F_1(t)\,dt=\int_{0.5}^1 t\,dt+\int_1^{1.5} dt=\frac{t^2}{2}\Big|_{0.5}^1+t\Big|_1^{1.5}=.375+.5=.875.$$

So in this case, we have $E_1\le E_2$ with a higher confidence level.

6. [TIME, SLOPE] TIMING MODELS

The delay of a given circuit may be strongly affected by the strength of the input signal. This observation suggests a third family of two-number models, where one number is used to represent the time of an event, and the other number describes the signal strength of the associated event. Experience

with Crystal [Ous84] suggests that the slope of the input signal can be used for improved delay estimates. The model presented here is one example of what such a timing model might look like when restricted to a two-number model.

6.1. Model Data Definitions

$$T=(t, r)$$

$$D=(A, K)$$

This model represents event times as t, a time when an input voltage is presumed to reach some *midpoint*, e.g., the switching threshold, and r, the rise time required for the signal to go from "OFF" to "ON" (thus $r=0$ would represent a step input). This rise time is the reciprocal of the input slope. Practical measures for this rise time may be the 10% to 90% rise time or the absolute value of the reciprocal slope at the midpoint. Delays also have two parts: a) The attenuation, A, is the reciprocal of an assumed gain. It multiplies the input rise time. b) K is the RC time constant to which this presumed input is being applied (See Figure 3-5). As in the previous model, there is a global path search variable, p, equal to 1 if maximal length paths are being examined, and equal to -1 if minimal length paths are being examined.

6.2. Delay Operation

The following discussion computes the output event time as the time at which the output signal crosses the geometric midpoint, and the reciprocal slope at that point. To delay the event $T_i=(t_i, r_i)$ by a delay $D=(A, K)$, first form:

$$r=Ar_i$$

To simplify computation, we translate the input time, t_i to the origin, solve for the output time in these coordinates, t_*, and then translate back to get the actual output time, t_o. The output rise time, r_o, is computed directly, as it is not sensitive to the time translation. There are four cases in solving for t_*:

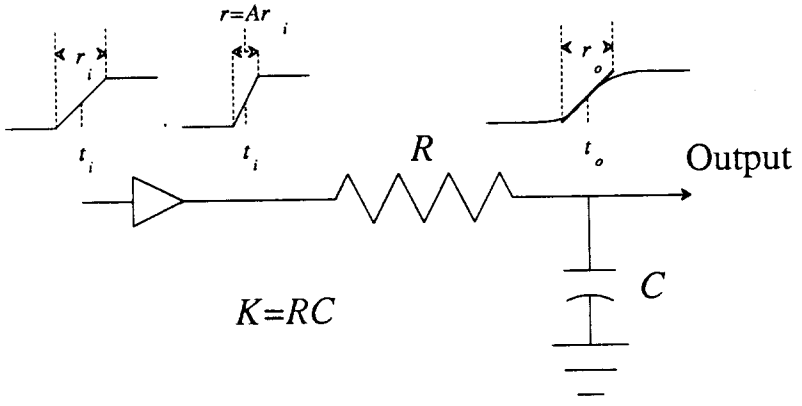


Figure 3-5: Delay operation for [time, slope] model

Case I: $r=0$ [step function].

$$t_* = K \ln 2$$

$$r_o = 2K$$

Case II: $K=0$ [no load].

$$t_* = \frac{r}{2}$$

$$r_o = r$$

Case III: $r>0$, but $t_* \geq r$ (as computed below). [input fast compared to output].

$$t_* = K \ln \frac{2K(e^{r/K}-1)}{r}$$

$$r_o = 2K$$

Case IV: $t_* < r$ (as computed above). [input slow compared to output]. In this case, we have an equation to be solved for t_*, rather than a closed form:

$$t_* + Ke^{-t_*/K} = K + \frac{r}{2}$$

$$r_o = \frac{r}{1 - e^{-t_*/K}}$$

Finally, we translate back to get t_o:

$$t_o = t_* + t_i - \frac{1}{2} r$$

6.3. Compare Operation

In this model, the comparison of two event times is primarily based on the relation between their time components. The rise times only come into play when the times are equal within practical margins.

$$\text{compare}(T_1, T_2) = \begin{cases} < & \text{if } t_1 < t_2 \text{ or if } t_1 \approx t_2 \text{ and } r_1 < r_2 \\ = & \text{if } t_1 \approx t_2 \text{ and } r_1 \approx r_2 \\ > & \text{if } t_1 > t_2 \text{ or if } t_1 \approx t_2 \text{ and } r_1 > r_2 \end{cases}$$

6.4. Merge Operation

As in the TA-like statistical model, the merge operation chooses one of the two input event times based on the type of path search being done and the results of a comparison between the two event times:

$$\text{merge}(T_1, T_2) = \begin{cases} T_1 & \text{if } T_1 \geq T_2 \text{ and } p = 1 \text{ or if } T_1 \leq T_2 \text{ and } p = -1 \\ T_2 & \text{if } T_1 < T_2 \text{ and } p = 1 \text{ or if } T_1 > T_2 \text{ and } p = -1 \end{cases}$$

6.5. Data Interpretation

This model was designed to describe small blocks, such as single stages in Crystal. The time constant, K, can be estimated for such blocks using some of the simpler Crystal models. The attenuation, A, describes the block's sensitivity to the shape of the input waveform.

6.6. Inverter Chain Example

One of the principal motivations for the [time, slope] model is to capture the effects of various drive strengths. As a test case, I studied the response of several chains of 10 nMOS inverters to a falling step input. The 10 inverters were paired into 5 stages, with each stage modeled by a single [time, slope] delay. Chains were composed of two types of stages: small load stages (where a 1pf load was added to the outputs of each of the inverters in the stage), and large load stages (where a 10pf load was used instead). A total of 5 chains were studied. Using the letter S to stand for a small load stage, and L to stand for a large load stage, these five chains can be represented as: 1) SSSSS (all small loads), 2) LSSSS (large load first), 3) SSLSS (large load in middle), 4) SSSSL (large load last), and 5) LLLLL (all large loads). Parameters for the small and large delays were derived by fitting the SPICE results from the SSSSS and LLLLL chains for individual stage delays. These parameters were: (A = 0.2, K = 230ns) for the small load stages, and (A = 0.2, K = 2300ns) for the large load stages.

Table 3-5 shows the results of using these parameters to model the LSSSS and SSLSS chains. These two chains are the best for observing the effects of variation in drive strength, because they each have a small stage immediately following a large stage. This stage (stage 2 for the LSSSS chain, stage 4 for the SSLSS chain) is driven more weakly than any of the other small stages, and hence has a longer delay. Table 3-5 shows how the [time, slope] model does reflect this effect for this stage. (The SPICE data also shows a slight "rebound effect" in the immediately following stage, which has a shorter delay than any of the others. This effect is not captured by the [time, slope] model at this level; perhaps it would be more accurately reflected if each inverter was modeled separately.)

Table 3-5: Delays at Each Stage (Midpoint-Midpoint, ns)				
Stage	LSSSS Chain		SSLSS Chain	
	Actual Delay (SPICE)	Modeled Delay	Actual Delay (SPICE)	Modeled Delay
1	1540	1590	150	160
2	**230**	**220**	160	160
3	140	170	1530	1590
4	160	160	**240**	**220**
5	160	160	140	170

7. ASYMMETRIC RISE AND FALL TIMES

Now that we have seen several basic timing models, we will consider how such models can be extended to represent asymmetric rise and fall times. All timing models can be extended in exactly the same way. In general, any timing model using m numbers to represent event times and n numbers to represent delays can be extended to a model with asymmetric rise and fall times. $2m$ numbers are then used to represent event times for rising and falling edges, while delays use $2n$ numbers to represent the delays for both cases and a flag which indicates how the signal polarity changes. Thus the two-number models discussed previously are extended to models with 4 numbers for event times and 5 numbers for delays. This extension is based on Hitchcock's observation [Hit82, HSC82] that analysis of worst-case arrival times only requires knowledge of the signal inversion properties of each block and its rising and falling delays, by maintaining separate values for rising and falling times and delays.

In the discussion below, data types and operations in the extended model are shown in bold type, while italic type is used for the underlying model. Starting with a timing model including event times E and delays D, we form times and delays for the asymmetric extension as:

$$\mathbf{E} = (E_r, E_f)$$

$$\mathbf{D} = (D_r, D_f, v),$$

where v is a flag with three possible values: *inverting*, *non-inverting*, or *unknown*. These values correspond to Bening's [BLA82] *positive unate*, *negative unate*, and *non-unate* behavior, respectively. A delay is inverting if any output transition must be opposite in direction from the input transition that

caused it, non-inverting if it must be in the same direction, and unknown if it could be either. Thus for boolean gates, AND and OR would have non-inverting delays, NOT, NAND and NOR would have inverting delays, and XOR would have unknown delays. Rising and falling delays are associated with output transitions of the indicated direction.

The delay, merge, and compare operations are defined in terms of the operations in the underlying model. For non-inverting delays,

$$\textbf{Delay(E, D)} = (Delay\,(E_r, D_r), Delay\,(E_f, D_f)).$$

For inverting delays,

$$\textbf{Delay(E, D)} = (Delay\,(E_f, D_r), Delay\,(E_r, D_f)).$$

Unknown delays are computed as the **Merge** of the two delay calculations above.

The **Merge** operation is defined as:

$$\textbf{Merge}(\mathbf{E_1}, \mathbf{E_2}) = (Merge\,(E_{r1}, E_{r2}), Merge\,(E_{f1}, E_{f2})).$$

When $Compare\,(E_{r1}, E_{r2})$ and $Compare\,(E_{f1}, E_{f2})$ yield the same result, $\textbf{Compare}(\mathbf{E_1}, \mathbf{E_2})$ is defined to be that result. If the two comparisons produce incompatible results, the overall event times are incomparable.

8. MORE COMPLEX ABSTRACT OPERATIONS

In addition to the basic operations defined above, other operations are usually implemented in each model to improve efficiency. Some of these are necessary operations that could, in principle, be synthesized out of the basic operations if they were not supplied. Others represent optional optimizations that do not affect the correctness of results, but do improve the efficiency of the program when they are provided.

Satisfy_constraint takes two event-times, $\{E_1\}$ and $\{E_2\}$, and returns the minimum translation amount t such that $\{E_1\} \le \{E_2\}+t$ (see Figure 3-6a). The semantics of $\le$ then guarantee that $\{E_1\} \le \{E_2\}+s$, for any $s \ge t$. *Merge_with_slacks* performs a merge on its input event-times, and also

returns the *local slack* for each input. The local slack of an input is the amount by which it could be translated in the direction of the merge (positively for long-path merges, negatively for short-path merges) before it becomes critical to its successor (see Figure 3-6b). In principle, satisfy_constraint could be synthesized out of the translation and comparison operations, and merge_with_slacks out of the merge, translation, and comparison operations, by trying various translations (e.g., by binary search) until the condition holds, but most models will be able to calculate the desired value more directly.

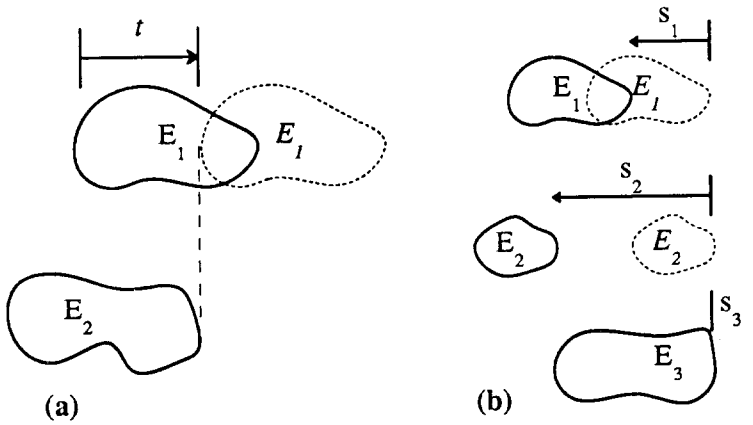


Figure 3-6: (a): Satisfy_constraint; (b): Merge_with_slacks

Sum-delays takes two delays and tries to compute a delay that will have the same effect as applying each of the input delays in series, based on an assumed default timing model for the particular delay format. This function is used to simplify the graph where possible. However not every model can define this operation, and the resulting delay is not guaranteed to produce the same effect as the original delays when used with timing models other than the assumed default. The application of sum-delays to graph simplification will be discussed in more detail in Chapter 4.

8.1. Critical Path Analysis and Slacks

One of the most important functions of a timing verifier is to identify critical paths in a design. These are the paths which need to be improved to speed up the design. A critical path from an event at some node is a succession of preceding events, each of which is critical to its successor. To use this

definition, we need to be able to identify which of several inputs to a merge is critical to its successor.

Intuitively, an input is critical to a merge if the result of the merge depends on that input. Every merge must have some critical input (it may have more than one).

There are several intuitive definitions of critical inputs that are perfectly good for one or more models, but fail in general. One attempt would be to define a critical input as one that would alter the output of the merge if it were omitted. This fails for the single-number model when all inputs to a merge are equal: under this definition, there would be no critical input. A second attempt would be to define a critical input as one that is equal to the output of the merge. This succeeds for models that merge by selecting among their inputs, but fails for models where the output of a merge is a combination of the inputs, such as the more complex mean & standard deviation model. A third attempt would say that an input a to a long-path merge is critical if there is no other input b to that merge such that $a < b$. This works for the above two cases, but fails for the min-max model when a=(3, 15) and b=(7, 10). Here the two inputs are incomparable, so both would be critical under this definition, but it is clear that the output of the merge is just a - input b is unimportant to the result.

The general definition of critical inputs used in the Abstract Timing Model is based on the translation and comparison operations. The intuitive idea is that if you "wiggle" an input to the merge and the output wiggles in response, the input was critical; otherwise not. More formally, we say that an event time E_i is *critical* to a long-path merge if we do *not* find that:

$$\text{Merge}(E_1, ..., E_i, ..., E_n) \approx \text{Merge}(E_1, ..., E_i + \varepsilon, ..., E_n),$$

for some small $\varepsilon > 0$ (e.g., a minimum time step). Similarly, for short-path merges, the wiggle is in the other direction, so the test is that we do *not* have:

$$\text{Merge}(E_1, ..., E_i, ..., E_n) \approx \text{Merge}(E_1, ..., E_i - \varepsilon, ..., E_n).$$

Given the definition of critical inputs, we can now define *slack* for all inputs: the *local slack* of an input is the amount by which it could be translated in the direction of the merge (positively for long-path merges, negatively for short-path merges) before it becomes critical to its successor. From this definition,

it is clear that critical inputs have zero local slack.

The *global slack* of an event with respect to some overall constraint or final destination, is just the minimum sum of local slacks from that event to the ultimate destination. From this, it is apparent that nodes on an overall critical path will have zero global slack.

8.1.1. Merge-with-slacks

Although I have defined slack in terms of critical inputs to a merge, it is evident from the preceding discussion that one can reverse this direction: if you know the local slack for each input to a merge, you can derive the critical paths from this by looking for local slacks equal to zero. This is the way ATV actually implements critical paths. Instead of the bare merge operation, most models supply a variant called merge-with-slacks, which returns both the merged event time and a list of local slacks in the merge, one for each input. If not supplied in a given model, this operation could be synthesized from the merge, translation, and comparison operations, but it would be inefficient to do so in general. By storing the local slack information with each event, along with the input events that formed it, the program can derive the global slacks to a system output or overall constraint by doing a shortest-path search in the backwards direction, using the local slacks as the distances between nodes. The n shortest paths back to source events (those with no predecessors) will identify the most critical and near-critical paths to the output.

8.2. Delay Format Coercion

As described previously, the type of the input event time to the delaying operation determines the type of delay used, if more than one is available. In addition, the type of delay necessary for a given event time may not be directly available. Both of these issues are addressed through a coercion mechanism for delay formats. Each event time format knows what type of delay it requires, and its implementation of the delaying operation attempts to coerce whatever delay is supplied into the required format. If multiple delay formats are available, they are collected into a special class of delay called a *delay-union*,

which can be coerced into any of its member classes or to any class they in turn can be coerced into. The user can control what coercions are to be allowed by only loading coercion methods corresponding to the allowable coercions between delay formats.

Often the supplied delay information on data sheets and the like do not correspond exactly to the required form. For example, a data sheet may supply only nominal and worst-case delay information, when we want min-max or mean and standard deviation. Often what has happened in the past is that the person developing a library of timing descriptions from these data sheets derives estimated values of the missing parameters from the supplied information, and enters it directly into the library definition, from which it is subsequently treated as gospel. For example, given a nominal delay of 10 and a max of 15, the library designer might assume that the nominal was halfway between max and min, and enter a minimum delay of 5, which might or might not correspond to the actual minimum delay of the part. Entering this number as the minimum delay would leave no record in the library of how this minimum delay was derived.

The coercion mechanism in ATV offers a better solution to this problem. A delay format can be defined that corresponds to the actual information available from the source, and other formats can be derived from this by supplying appropriate coercions. In the above example, a nominal-worst case delay format could be defined with parameters n and w, and a coercion to min-max supplied that takes w as the max, and computes the min as the greater of zero and $2*n-w$ (i.e., the min is chosen to make n the average of min and max, except when this would make the min delay less than zero). This has the advantage that assumptions about derived numbers are made explicit, and if these assumptions are later changed, only the coercion needs to be changed, not every library definition.

Chapter 4 - Implementation of an Abstract Timing Verifier

1. ATV EXTENSIONS

ATV extends the notion of a timing graph developed in the last chapter. In ATV, arcs come in two flavors: *ordinary arcs*, corresponding to pure combinational elements, which always transmit data, and *gated arcs*, corresponding to sequential elements, which are controlled by some clock node, and only transmit data when the clock node is in the appropriate state (see Figure 4-1). Both types of arcs can have *delays* attached to them, which specify how an event time at the tail of the arc is transformed into the consequent event time at the head of the arc. Although specific logic functions are not represented in the graph (since the timing verifier is generally only concerned with the stable/changing behavior of signals), arcs do have an *arc type* attached, which specifies the inverting properties of the arc. This arc type can have one of three values: INVERTING, NON-INVERTING, or UNKNOWN. These are the same as the types of delays in the asymmetric rise-fall model, and if such a delay is present, the delay-type will be used as the default for the arc type. An arc is inverting if any output transition must be opposite in direction to the input transition that caused it, non-inverting if it must be in the same direction, and unknown if it could be either.

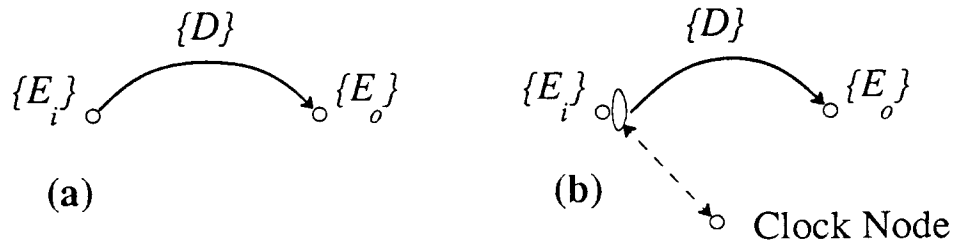


Figure 4-1: (a): Ordinary arc. (b): Gated arc.

Gated arcs act as if they had a small gate at the tail of the arc, controlled by the specified clock node. They only transmit data when the gate is open. Such arcs have both opening and closing edges specified from the set {R, F}. The gate is open from the time the specified opening edge occurs at the clock node until the specified closing edge occurs (which may be the same edge). If the opening and

closing edges are the same, the arc exhibits edge-triggered behavior; otherwise it exhibits level-sensitive behavior.

Thus a level-sensitive transparent latch open when the clock node is high would be modeled with a gated arc with OPEN=R, CLOSE=F, while a rising-edge triggered register would be modeled with a gated arc with OPEN=R, CLOSE=R. Clocks are distributed over ordinary arcs from user-specified origin nodes. For now, clocks must be distributed over pure fan-out trees from the origin nodes. If necessary, clock enable signals should be handled specially. In Figure 1 of the last chapter, signal G was a clock enable signal modeled by eliminating any direct arcs from it to the clock line, and instead establishing constraints between it and clock node H that will force G to be stable whenever H is high. A typical timing graph is shown in Figure 4-2.

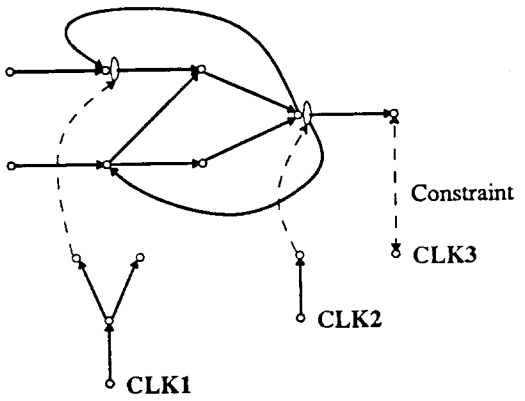


Figure 4-2: Typical timing graph.

1.1. Types of Analysis

There are several types of analysis possible with ATV. The piece of the timing graph being analyzed may involve purely combinational logic, or it may include sequential elements. If it includes sequential elements, the clock schedule may be known or unknown initially. To support analyses where the clock schedule is not known in advance, ATV supports the idea of multiple *reference frames*. Events in one reference frame have a known relationship to one another, but different reference frames have

different origins, which may have an unknown relationship to each other. One of the objectives of clock-phase length analysis is to determine what relationships must hold between the different reference frames.

1.2. Clock Edge Graphs in ATV

Regardless of how events are allocated to individual reference frames, ATV assumes that the general sequence of clock edges is known. This qualitative sequence is used to order the events being processed by the program, so that events associated with earlier clock edges can be processed before events associated with later clock edges. This sequence of clock edges is initially specified in the form of a *clock graph*, a (generally cyclic) directed graph which has a node for each clock edge during some user-defined *machine cycle*, and directed arcs from each clock edge to its direct successors.

In setting up for the analysis, this machine cycle is repeated a user-specified number of times to form the *unrolled graph*, a directed acyclic graph which has a distinct node for every clock edge in every machine cycle. This graph defines a partial order on individual clock events which will be observed by the program during the analysis. These graphs are completely distinct from the timing graph: they provide sequencing information only. Events must still be defined on the timing graph to define the actual arrival times and reference frames for the actual clock events.

An example showing the clock graph and unrolled graph for a simple 2-phase non-overlapped clock sequence is shown in Figure 4-3.

More complex clocking schemes involving overlapping clocks and partially asynchronous edges are also possible. For example, Figure 4-4 shows a clock ordering in which **clk1** goes high twice per cycle, and **clk2** goes high in between the first and second time that **clk1** goes high. There is an additional clock, **clk3**, which goes low between the first and second occurrence of **clk1**. The rising edge of **clk2** and the falling edge of **clk3** are asynchronous with respect to each other, but both occur before either of the following edges. Finally, there is an additional clock, **clk4**, which only goes low in between the alternate instances of **clk1** going high. Figure 4-5 shows the corresponding **Clock Graph**.

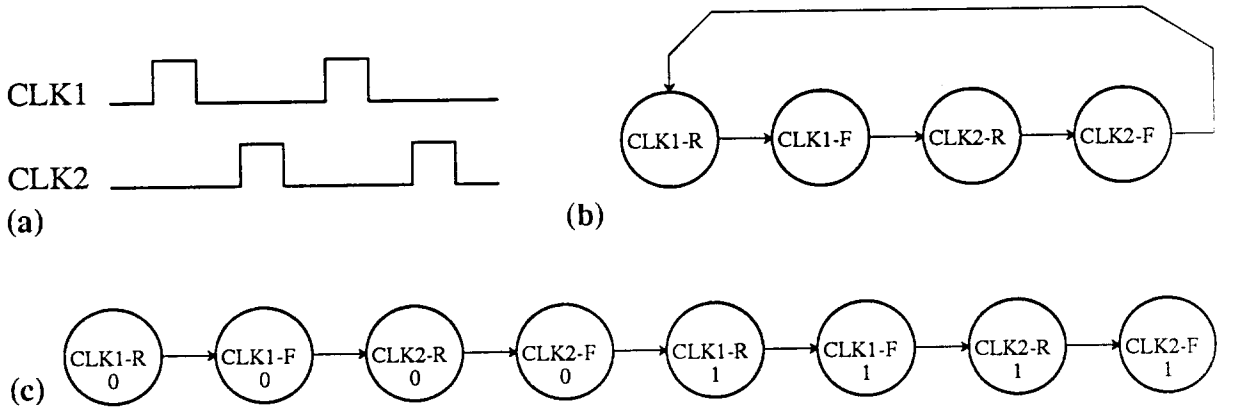


Figure 4-3: (a) Clock Schedule; (b) Corresponding Clock Graph; (c) Unrolled Graph for Two Machine Cycles.

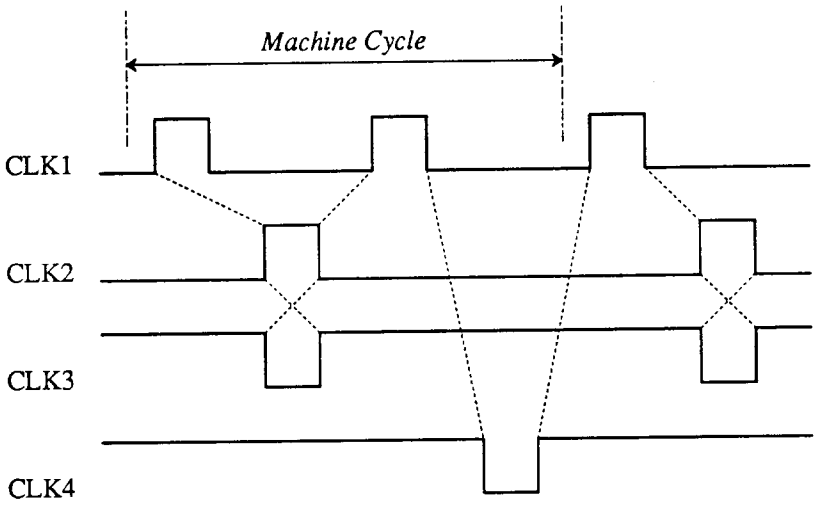


Figure 4-4: A more complex clock sequence.

It is also possible to define some clock events as aliases for others. For example, in the previous example, instead of **clk2** rising and **clk3** falling being separate events which are asynchronous with respect to each other, they could be defined as aliases, in which case they are regarded as occurring at the same time for purposes of sequencing.

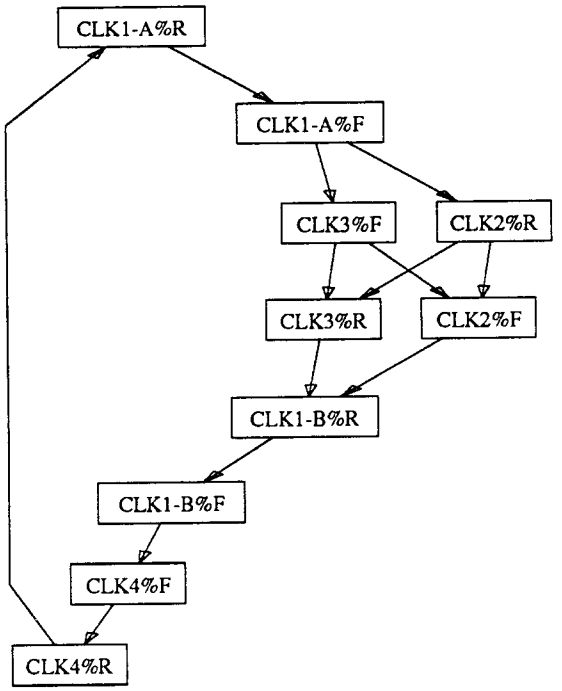


Figure 4-5: The Clock Graph corresponding to the sequence shown in Figure 4-4.

1.3. Event Representation

Events in ATV consist of:

1: A type of edge, one of {R, F, CH-S, or S-CH} (i.e., rising, falling, changing-to-stable, or stable-to-changing). This defines *what* happened.

2: An event time. This defines *when* it happened.

3: A reference frame. This defines a symbolic origin for the event time. It is possible to have events from several different reference frames whose origins are (initially) unknown relative to each other.

4: The last clock edge known to have occurred. This provides a coarse synchronization mechanism, which acts as a heuristic filter to restrict attention to interactions between events that occur close enough to be of concern. For example, we do not want to check a set-up constraint between a clock edge and the arrival of a data item that will not be generated until after the clock edge

arrives.

5: A list of predecessor events that gave rise to this event: to search for critical paths.

6: A list of slacks for those events.

7: The node or arc this event is associated with. This allows a critical path to be reported to the user.

1.4. Constraints

In general, a constraint specifies a comparison (≤,≥) required to hold between event-times of specified events (specified by edges) at two nodes of the timing graph. It includes a translation which is applied to the second event-time before the comparison is expected to hold. For example, an edge-triggered register with a set-up time of 3 and a hold time of 5 could have the set-up constraint specified as:

```
(constraint 'd 'ch-s '<= 'clk 'r -3)
```

(i.e. $D_{ch-s} \leq CLK_R - 3$), and hold constraint:

```
(constraint 'd 's-ch '>= 'clk 'r +5)
```

(i.e., $D_{s-ch} \geq CLK_R + 5$).

In general, a constraint only applies between two events if their last-clock-edges occur in the order specified by the comparison. Thus the set-up constraint above would only compare data ch-s edges with a last clock edge on or before the particular rising edge being compared against. An escape mechanism is provided so that this restriction can be either widened or narrowed, to include different last-clock-edges.

2. DERIVING THE GRAPH

Input to ATV can come from either of two sources: the timing graph can be derived from blocks with delay descriptions in the Oct database [HMS86], or the timing graph can be entered directly (which provides a hook for input from other sources). The following sections discuss the derivation of timing graphs from Oct library descriptions.

2.1. Specifying Models for Library Cells

A timing graph description of a block in Oct is specified as a set of arcs and a set of constraints. Both arcs and constraints operate between *terminal groups*: groups of zero or more terminals that act as a unit for purposes of timing verification (e.g., all 8 output terminals of an 8-bit latch could be considered identical). Terminal groups with zero terminals correspond to internal points of the timing graph description. Delays are specified as expressions attached to arcs, 1 per representation.

2.2. Merging the Graph

To get input from an Oct description of a design, the Oct description is first flattened down to the level of cells that have active timing graph descriptions. Next, an interface program called **oct2atv** is run, which extracts the timing graph representations of the flattened cells and glues them together into an overall timing graph (see Figure 4-6). This interface program has no knowledge of the particular delay models in use: delays are specified as expressions to be evaluated (in Lisp) by ATV.

The interface program glues the timing graph together by adding arcs corresponding to the nets that connect terminals of the individual blocks. All such arcs have NONINVERTING arc-type. By default, these net arcs are given null delays, corresponding to the assumption that the block delays include the associated interconnect delays. However, this default can be replaced with other defaults, or with individual arcs specified for each piece of interconnect. It is possible for each of these arcs to be specified individually. A modeling program could go through and insert each such arc individually. More general modeling of interconnect requires delays that can be parameterized in terms of variables such as net

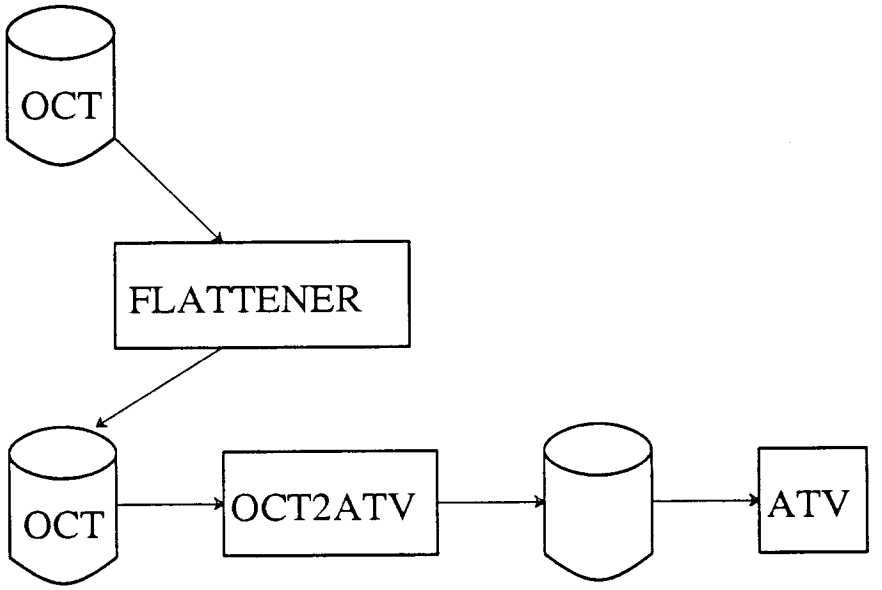


Figure 4-6: Interface from the Oct database: the design view is first flattened and then run through the interface program OCT2ATV to produce input to ATV.

length, which will be discussed in the next section.

2.3. Parameterizable Delays

Because delays are modeled with general Lisp expressions, they may include variables. The difficulty is how to provide a general mechanism for supplying values for these variables. In the model definition, there could be defined a set of needed Oct properties: the interface program could look for these properties on the attached net and pins of the net, sum them, and supply them as an evaluation context to ATV, so that when the delay expressions are evaluated, these parameters are bound to the appropriate values.

There are two principle difficulties with this approach: the resulting expressions require a Lisp interpreter (or subset) to evaluate them, which makes them difficult for other programs not written in Lisp to read, and the mechanism of variable substitution by a summed expression is not sufficiently powerful to capture all the possible forms of parameterization one might want. In particular, one might want fairly complex rules for deriving delays from the properties to be found in the database, rules of the form: if the

geometry of the net is completely specified, use rule1; if not, but it is partially specified, use rule 2; if no geometry is specified but net length estimates are available, use rule 3; and if no net length estimates are available, use the fan-out to estimate them via rule 4. The individual rules here could be much more than just simple parameter evaluations; they could be based on the entire topology of the net, such as modeling the net as a distributed RC-tree.

The best way to obtain this flexibility might be to integrate ATV more tightly with Oct, and make the rules for deriving delays part of the model definitions in ATV, giving the model routines direct access to the data from the Oct database; this, however, has the disadvantage of making the model definitions much more complex and making it more difficult to bring in input from non-Oct sources.

3. IMPLEMENTATION IN PORTABLE COMMON LOOPS

The implementation of an abstract timing verifier poses some special problems. Because the models are intended to be user-extensible, the range of possible types assumed by time and delays is unbounded and not known at the time most of the code is compiled. Consider the problem of defining the asymmetric-rise-fall-time model, where the components of delays and event times are themselves delays and event times in some other model: the fundamental operations of translation, delay, etc., are all defined in terms of the same operations in the component model, whatever that may be. Thus the code for delay must somehow know how to invoke the right delay operation for the component model based on the types of the component event times — types which may not even be defined at the time the code for the asymmetric-rise-fall delay operation is compiled. Such a problem seems to call for the facilities provided by object-oriented languages, where individual objects carry with them the information about how to perform operations on them.

Our ATV prototype is implemented in Portable Common Loops [BKK86], an object-oriented extension to Common Lisp [Ste84]. ATV makes use of two of the fundamental features provided by object-oriented systems such as PCL: *polymorphic function calls* — many functions with the same name but different argument types are loaded together, with the actual function called determined from the

run-time type of the arguments — and *inheritance* — new *classes* (i.e., types) can be defined from pre-existing classes, inheriting any *methods* (functions) defined on the pre-existing class. Unlike many other object-oriented systems, PCL allows methods to be selected based on the type of more than one argument (*multi-methods*). PCL was chosen over other object-oriented systems because (1): its method invocation syntax is identical to ordinary function calls, (2): it allows the user to write specialized versions of existing functions that apply to specified classes, and (3): it allows methods to be selected based on the type of more than one argument.

Delays and event-time definitions in ATV are PCL classes. They are based on two generic classes called **abstract-delay** and **abstract-event-time**, respectively, which supply default behavior for most functions. New representations can be loaded as needed, and will co-exist with the predefined representations. These representations can be based on existing representations. For example, many of the mean-standard deviation models discussed in the last chapter share certain operations, such as translation. Thus new mean-standard deviation models can be based on the simple mean-standard deviation model, with only those operations that differ redefined.

One example of the use of multi-methods is the delay operation. As described previously, the type of the input event time to the delaying operation determines the type of delay used, if more than one is available. In addition, the type of delay necessary for a given event time may not be directly available. Both of these issues are addressed through a coercion mechanism for delay formats. Each event time format knows what type of delay it requires, and its implementation of the delaying operation attempts to coerce whatever delay is supplied into the required format. If multiple delay formats are available, they are collected into a special class of delay called a **delay-union**, which can be coerced into any of its member classes or to any class they in turn can be coerced into. The user can control what coercions are to be allowed by only loading coercion methods corresponding to the allowable coercions between delay formats.

The standard methods implementing the delay operation are specialized on the class of the event-time argument, and take any abstract-delay as the delay argument (which will subsequently be coerced

into the appropriate type). For example, here is the delay method for the single number model:

```
(defmethod delay ((e single-number-et) (d abstract-delay))
  (let ((d1 (coerce-meth d 'single-number-delay)))
       (make-single-number-et :num (+ (num e) (num d1)))))
```

The first line defines delay as a function of two arguments, e and d, and defines this code as the method to be invoked when the first argument is a single-number event-time and the second is any abstract delay. The second line coerces d to the single-number delay format, setting the temporary variable d1 to the result of the coercion. The third line computes the result of the delay operation for this case. Any user-defined delay class based on abstract-delay will inherit this method whenever the first argument is a single-number event-time.

Other event time classes define the delay operation similarly. Thus the min-max delay definition starts:

```
(defmethod delay ((e min-max-et) (d abstract-delay))
  (let ((d1 (coerce-meth d 'min-max-delay)))
    ...
```

and the mean-standard deviation delay definition starts:

```
(defmethod delay ((e mean-stdDev1-et) (d abstract-delay))
  (let ((d1 (coerce-meth d 'mean-stdDev-delay)))
    ...
```

and if the user wishes to define a new event time class called foo-et that takes delays of the type bar-delay, the definition of the foo-et class would include a delay method that would start:

```
(defmethod delay ((e foo-et) (d abstract-delay))
  (let ((d1 (coerce-meth d 'bar-delay)))
    ...
```

which could then be loaded along with all the other delay method definitions, to be invoked whenever the first argument to delay was a foo-et and the second was any abstract delay.

In all the above cases, the selection of which delay method to run is primarily based on the class of the first argument. There are, however, a few cases where we want the delay operation to do something special. For example, it is useful to be able to define a special type of delay, called a *null-delay*, that always passes its input event-time through unchanged. Null-delays are used for arcs that only serve to connect other components of the graph, without contributing delays. For example, arcs corresponding to interconnect between blocks when the interconnect delays have already been incorporated into the block delays can be modeled with null delays. One way to implement this would be to have every delay method check to see if the second argument was a null-delay, but it is much simpler to add a new null-delay class and define a new method:

```
(defmethod delay ((e abstract-event-time) (d null-delay))
  e)
```

Because null-delay is defined as a separate class that is *not* a subclass of abstract-delay (unlike other delay classes), none of the other delay methods will apply when the second argument is a null-delay, so this method will override other delay methods when the second argument is a null-delay. (Note: this can be implemented more simply by using the PCL feature of specializing methods to individual instances, once this feature becomes available in PCL.) Any user-defined event-time model that is a subclass of abstract-event-time will automatically inherit this behavior, as desired.

The above idea can be extended to allow for coercion between types of event times, by defining special delay classes that will perform the coercions when they are involved in a delay operation. This could be used to verify a sub-circuit using a different timing model than used in the surrounding circuit, by making all incoming arcs to the sub-circuit coerce the input event times to the model used in the sub-circuit, and having all outgoing arcs from the sub-circuit perform the inverse coercion. For example, a designer might want to verify a MOS chip using a different timing model than that used for the surrounding TTL circuitry on a board.

Note how these examples exploit PCL's ability to discriminate on more than one argument for method selection: in some cases we are discriminating primarily on the class of the first argument, but in these latter cases, we are discriminating primarily on the class of the second argument. This feature would not be available in a "pure" (message-based) object-oriented system, where methods can only discriminate on the class of the receiver of a message (i.e., the first argument of a function).

4. BASIC ALGORITHM - TIMING VERIFICATION AND TRANSPARENT LATCHES

Transparent latches complicate the timing verification of designs that include them. Edge-triggered or master-slave storage elements interrupt the flow of data through them such that critical paths will never extend through them. If all the storage elements in a design are of this type, we only need to look at paths through combinational logic between storage elements. But with transparent latches, critical paths can potentially extend over multiple cycles, passing through several intermediate latches at any time during their respective open periods.

Previous timing verifiers have attempted to address this problem in various ways. Crystal [Ous85] operates one clock phase at a time, relying on the user to specify how available time should be split between phases. Jouppi's TV program [Jou84] relies on "cycle borrowing" to shift available time between consecutive clock phases. Glesner *et al.*[GSS86] describe an algebraic approach that can analyze paths extending over multiple machine cycles assuming that all clock phases have the same width. LEADOUT [Szy86] creates a series of equations relating all events in the networks by their respective delays derived from a dependency graph between events (not nodes, as in ATV); no procedure is given for solving these equations when the initial clock schedule is unknown.

In contrast, ATV examines all paths up to a user-specified number of machine cycles in length, allows clock phases to have non-uniform widths, allows for more complex clocking schemes than just p non-overlapped clock phases per machine cycle, and allows all, some, or none of the original clock schedule to be prespecified by the user.

Events in ATV include:

1) A signal_edge (RISE, FALL, CH-ST, or ST-CH),

2) an event_time (with model dependent values),

3) a reference_frame, and

4) the last_clock_edge known to have occurred.

The basic idea of the clock-phase length analysis is that the user will initially specify each clock edge whose position is to be calculated in its own reference frame (with arbitrary origin). Clock edges whose positions are known relative to each other will share a reference frame. There is some set of translations:

$$\mathbf{X}=(x_1, x_2, \cdots, x_n)$$

from each of these reference frames to the underlying "real" reference frame.

ATV computes bounds on the differences between pairs of translations:

$$x_i - x_j \geq b_{ij}$$

Such bounds could subsequently be solved as a linear program together with any other required constraints, to solve for the actual set of translations.

4.1. Specification of Clock Edge Event Times

To set up for the analysis, the user specifies:

1: The extent of the analysis (number of cycles, sequence of clock edges, which cycles initiate events).

2: An input event for each clock edge over the period of the analysis.

3: Input events for system inputs (each in the reference frame of some clock edge).

4: Output requirements specified as constraints between a system output and some clock edge.

4.2. Performing the Analysis

Consider the sample circuit shown in Figure 4-7, where latch L is a transparent latch, and the other storage elements are edge-triggered registers. For simplicity, assume we are using the single-number model, with delays as indicated, and that all setup and hold times are zero. Initially we assume that each clock edge occurs in its own reference frame at time 0. In preprocessing, the program traces forward from each clock origin node to determine what clock origin controls each gated arc and which phase of that clock opens the arc (by counting inversions along the clock distribution network). Thus it will mark arc B-C as controlled by CLK1, opening on its falling edge. Because it is important to be able to identify the controlling phase of each gated arc, special rules apply to the clock distribution network. Clocks must be distributed over pure fan-out trees with known inversion properties (i.e., no unknown arcs) from the clock origin nodes. Clock enable signals are handled specially as described earlier. This requirement is reasonable for most design styles, except for clock generation circuitry (before the clock origins), which must be handled separately.

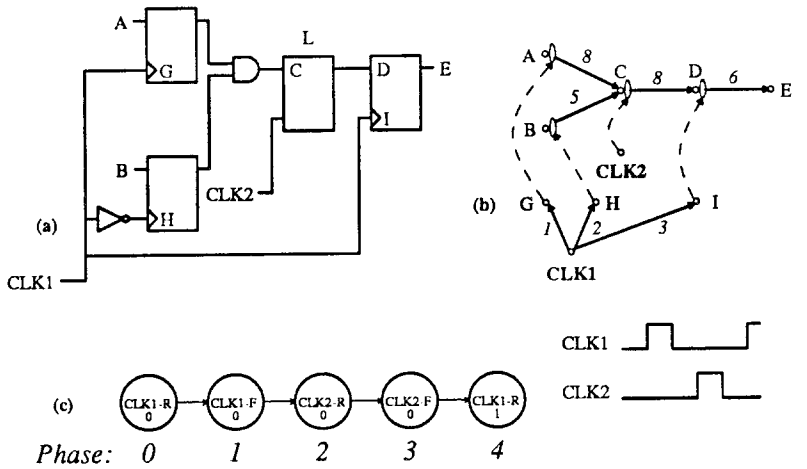


Figure 4-7: Transparent latch example. (a): Logic Diagram (L is the transparent latch), (b): Timing Graph, (c): Clock Schedule.

ATV proceeds in a series of phases, one for each clock edge in the unrolled clock graph, in an order compatible with the partial order defined by this graph. In each phase it initiates events starting in

this phase and propagates them through the network together with any other events waiting for this phase to occur before they can proceed. Each phase begins with Jouppi's predecessor counting [Jou84], a quick depth-first search to determine how many predecessors of each node are active in the current phase. It also detects and breaks any unclocked loops that may be found. Then the subsequent pass that actually calculates the events knows when all the active predecessors to a given node have been processed, so the node itself can be scheduled.

Thus in phase 0 (CLK1 rising), predecessor counting will note that node C has only one active predecessor (node G); thus it can be evaluated as soon as node G has been evaluated. Each node is then evaluated in a breadth-first manner as soon as all of its predecessors have been evaluated. The last_clock_edge of each event is equal to the phase in which it is calculated.

When a clock event reaches a clock node that gates some arc, it will initiate both a ST-CH and a CH-ST event on that arc with the same event-time as the arrival at the clock node.* This represents the (brief) possible transition at the opening clock edge if the input of the arc has previously reached a value different from the output. Clock events after the first machine cycle are typically disabled from initiating such events, as no new violations or constraints will be discovered unless the user is performing *case analysis* (selectively disabling impossible paths) in the first cycle. This prevents the same path from being identified as the critical path limiting phi1 rising in cycle 0 to phi2 falling in cycle 1, phi1 rising in cycle 1 to phi2 falling in cycle 2 and also phi1 rising in cycle 2 to phi2 falling in cycle 3. Events with the same last_clock_edge, edge_type, and reference_frame at a given node are merged together (CH-ST events merge for longest paths, ST-CH events merge for shortest paths). When events arrive at the input to a transparent-latch arc (i.e., with different opening and closing edges) that is open in the current phase, they propagate through the arc. If the arc is not currently open, they wait for the next phase that opens this arc, and update their last_clock_edges as they pass through. Relevant events calculated for the first

*One of the few cases where ATV cannot be completely blind to the actual model in use occurs here. If the event time at the clock node is an asymmetric rise-fall time, only one of the two components actually applies to the current event, corresponding to the clock edge (rising or falling) occurring at the clock node. In this case, ATV needs to extract the relevant component of the event time, create a new event-time with both components equal to this relevant component, and use this new event time for both the ST-CH and CH-ST events.

five phases of the example from Figure 4-7 are shown in Table 4-1 below.

Table 4-1: Selected Events from Transparent Latch Example

Phase	Node	Edge	Event-Time	Reference Frame	From
0	CLK1	R	0	0	-
	I	R	3	0	CLK1
	H	F	2	0	CLK1
	G	R	1	0	CLK1
	C	CH-ST	9	0	G
	C	ST-CH	9	0	G
1	CLK1	F	0	1	-
	I	F	3	1	CLK1
	H	R	2	1	CLK1
	G	F	1	1	CLK1
	C	CH-ST	7	1	H
	C	ST-CH	7	1	H
2	CLK2	R	0	2	-
	D	CH-ST	8	2	CLK2
	D	ST-CH	8	2	CLK2
	D	CH-ST	17	0	C
	D	ST-CH	17	0	C
	D	ST-CH	15	1	C
	D	CH-ST	15	1	C
3	CLK2	F	0	3	-
4	CLK1	R	0	4	-
	I	R	3	4	CLK1
	H	F	2	4	CLK1
	G	R	1	4	CLK1

After all events have propagated through the network, ATV examines all events that are involved in a possible constraint. Constraints specify two events to be checked, an optional translation to be applied to the second event, and a condition ($\leq$, $\geq$) that is expected to hold between them. In the above example, there are two relevant constraints: the data signals at C must stabilize before the closing edge of CLK2 occurs (CLK2 falling), and the data at node D must stabilize before the next closing event at node I (I rising). The last_clock_edges (phases) of the events at the two nodes are used to screen out incorrect comparisons: the default rule is that the last_clock_edges of events in a valid comparison must be in the same relation as the condition being checked. Thus the rising event at node I in phase 0 is not checked against the CH-ST events at node D in phase 2, because 0<2. If the constraint is specified as:

$$E_i \leq E_j + \Delta,$$

for some translation Δ, then E_i and $E_j+\Delta$ are passed as arguments to the *satisfy-constraint* routine. This will provide the minimum t such that

$$E_i \leq (E_j + \Delta) + t,$$

which produces a bound on the translation difference between the two reference frames:

$$x_j - x_i \geq t. \tag{I}$$

Because of the time-invariance of the delay, merge, and compare operations, this same bound applies to the allowable translation of the source events in each reference frame (see Figure 4-8).

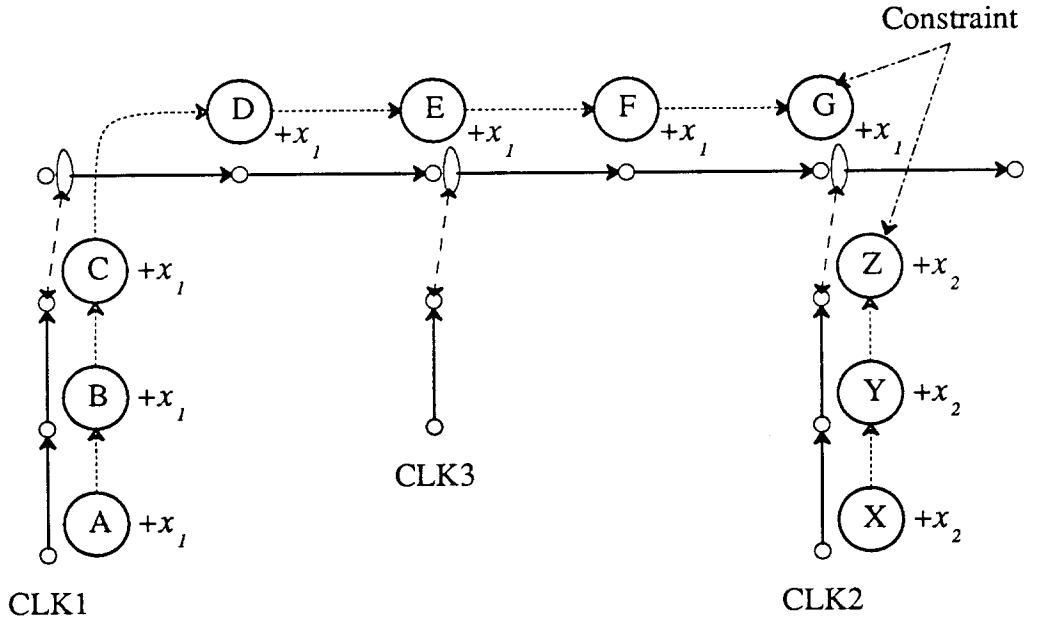


Figure 4-8: Constraint analysis. A constraint exists between events G and Z, which are in two different reference frames with translations x_1 and x_2, respectively. Any bound on the difference of these two translations applies to the translations of the original input events, A and X. Note: events F and G will have a last clock edge of CLK3 (whichever of rising or falling corresponds to the opening edge of the arc) because they have passed through an arc gated by CLK3. They are still specified in the original reference frame of event A, however.

From the two events involved in a constraint, we can derive the two critical paths (one short, one long) that produced this bound, by tracing backwards and looking for the preceding events with smallest slacks. After generating all the $x_j - x_i$ bounds, the largest t is the overall bound. If $i=j$, the left hand side of (I) is necessarily zero, so $t>0$ indicates an error.

Constraints for the reference frames in our example are shown in Figure 4-9, below. A sample set of translations, obeying the constraints with a minimum cycle time of 14 is shown in Figure 4-9. If the user desires, these translations could be applied to the original input event times (yielding event times

equal to the translations, since the original event times were all zero), and the analysis could be rerun with all events now in the same (absolute) reference frame. This would yield results similar to those shown in Table 4-1 with the appropriate translations, but now many of the events in the same phase can be merged together, since they are now in the same reference frame. This would allow the user to calculate times for all events in the absolute reference frame.

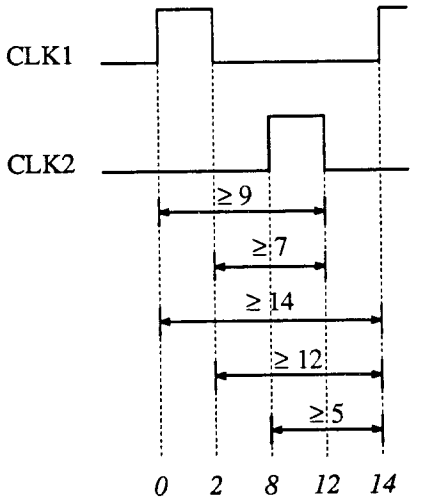


Figure 4-9: Constraints between reference frames, transparent latch example. Sample translation times obeying constraints are shown.

4.3. Derivation of Clocking (L. P. Problem)

Once a set of constraints has been derived from the analysis, they can be used, in principle, as input to a linear programming problem together with any other constraints that may hold. For example, in a simple two-phase clocking scheme, if ϕ_1 rising occurred in frame 1 in cycle 1, and in frame 5 in cycle 2, while ϕ_2 rising occurred in frame 3 in cycle 1 and in frame 7 in cycle 2, then two additional constraints could be:

$$x_5 - x_1 = c$$

$$x_7 - x_3 = c\,,$$

where c is an additional variable, the length of a machine cycle. This cycle must be the same length for both ϕ_1 and ϕ_2. Note: these equations assume that ϕ_1 rising was specified with the same event time in both frames 1 and 5, and likewise ϕ_2 rising in frames 7 and 3. If there is an offset between the specified input event times, then this would have to be taken into account in formulating the above equations. ATV does not presently include such a linear programming step, but it is a straightforward extension to the program.

Once the set of translations has been obtained, they can be applied to all the input event times to produce the event times for each input event in the absolute reference frame.

4.3.1. Deriving Clocking for Jouppi's Loop Example

I have already used Jouppi's loop example in Chapter 2 to illustrate both cycle borrowing (Figure 2-9) and the SCAT clock period algorithm (Figure 2-10). Now I will use it to show how we can derive the optimal clock schedule from the constraints computed by ATV. The example is shown in Figure 4-10. Each clock edge is specified to occur at time 0 in its own reference frame, as shown.

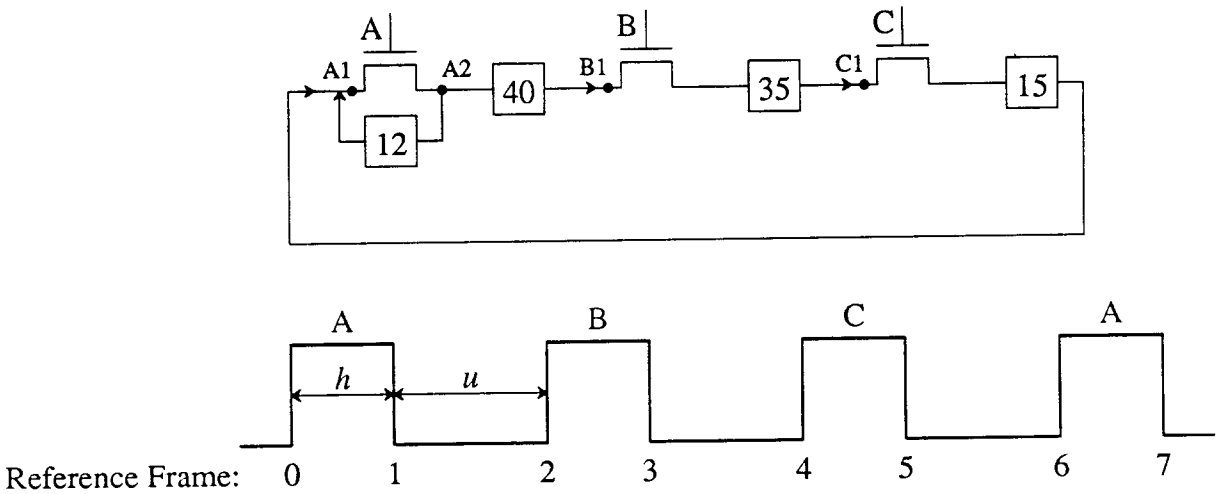


Figure 4-10: Jouppi's Loop Example. Reference frames for all clock edges are shown; the example assumes that both the clock high period, h, and the clock underlap, u, are the same for all three clocks. The points shown are the nodes used for the timing graph in ATV.

Jouppi's example makes assumptions different from the assumptions about legal design structures and their semantics that I used in designing ATV. In particular, the example has a loop that is only clocked by a single transparent latch (the short loop between A1 and A2). He assumes that this loop is computing a new value for latch A that must not arrive before the latch closes. In order to reflect this constraint in ATV, the path from A2 to A1 is defined as a multi-cycle path, and a constraint is added between the closing edge of A and the ST-CH event at A1 in the next machine cycle. If the path from A2 to A1 was not defined as a multi-cycle path, ATV would assume that this single-cycle loop was part of a storage element restoring the value at A1, and it would break the loop between A2 and A1.

Like the SCAT timing verifier, ATV by default assumes that the transparent latches it encounters occur in serial chains, not loops like the latches in Jouppi's example. By default, signals are only checked against the closing edge of each latch. This is the correct behavior for most events, but needs to be supplemented for latches in loops with a special check that an event originating on the opening edge of a latch in a loop arrives before the opening edge of that same latch in the next machine cycle. In general, this check would require a special loop analysis step that will be discussed under future work in Chapter 6. Lacking such a step, it is sufficient that at least one latch in every loop checks its input against its opening edge, rather than its closing edge. For this example, we can add a constraint that the value at A1 must stabilize before the opening edge of A occurs (rather than the default closing edge). This ensures that the loop is not too long and does not affect the solution for this example. In general, this constraint would be overly conservative because it also restricts paths originating from latches B or C.

The default constraints generated by ATV are:

$$x_3 - x_0 \geq 40 \tag{4.1}$$

$$x_5 - x_0 \geq 75 \tag{4.2}$$

$$x_5 - x_2 \geq 35 \tag{4.3}$$

$$x_7 - x_0 \geq 90 \tag{4.4}$$

$$x_7 - x_2 \geq 50 \tag{4.5}$$

$$x_7 - x_4 \geq 15 \tag{4.6}$$

The short loop constraint from A1 to the falling edge of A generates a constraint:

$$x_0 - x_1 \geq -12, \tag{4.7}$$

or rewritten,

$$x_1 - x_0 \leq 12. \tag{4.8}$$

The constraint that A1 must stabilize before the opening edge of A generates three constraints:

$$x_6 - x_0 \geq 90 \tag{4.9}$$

$$x_6 - x_2 \geq 50 \tag{4.10}$$

$$x_6 - x_4 \geq 15 \tag{4.11}$$

The SCAT algorithm did not take loops into account at all, and thus was solving a reduced version of the problem without the last two sets of constraints.

To these we add the constraints that define the length of the clock high period, h, and the clock underlap, u, for each clock. Because these are defined to be the same for every clock in this example, we can use the same variables h and u for each phase. If this were not the case, separate variables would be defined for each clock phase. These constraints are:

$$x_1 - x_0 - h = 0 \tag{4.12}$$

$$x_3 - x_2 - h = 0 \tag{4.13}$$

$$x_5 - x_4 - h = 0 \tag{4.14}$$

$$x_7 - x_6 - h = 0 \tag{4.15}$$

$$x_2 - x_1 - u = 0 \tag{4.16}$$

$$x_4 - x_3 - u = 0 \tag{4.17}$$

$$x_6 - x_5 - u = 0 \tag{4.18}$$

The problem is now to minimize the cycle time,

$$c = h + u \tag{4.19}$$

while observing all the constraints.

In solving this linear program manually, one can use the equalities to reduce the first set of constraints to linear constraints on h and u. The dominant inequalities from 4.1-4.6 are 4.1, 4.2, and 4.4, which become:

$$u \geq 40 - 2h \tag{4.20}$$

$$u \geq 37.5 - \frac{3h}{2} \tag{4.21}$$

$$u \geq 30 - \frac{4h}{3}, \tag{4.22}$$

respectively. The other three constraints are dominated by one of these (i.e., 4.3 reduces to $u \geq 35 - 2h$, which is clearly dominated by 4.20). The dominant constraint from the large loop (4.9-4.11) is 4.9, which reduces to:

$$h + u \geq 30 \tag{4.23}$$

and constraint 4.8 from the short loop reduces to:

$$h \leq 12 \tag{4.24}$$

Because this linear program only involves two variables, it can be solved graphically (Figure 4-11). Note that unlike cycle borrowing or the SCAT algorithm, additional linear constraints can be readily incorporated by including them in the linear program to be solved. Thus if the clock generation circuitry imposed minimum or maximum restrictions on h or u, these could be incorporated as additional constraints.

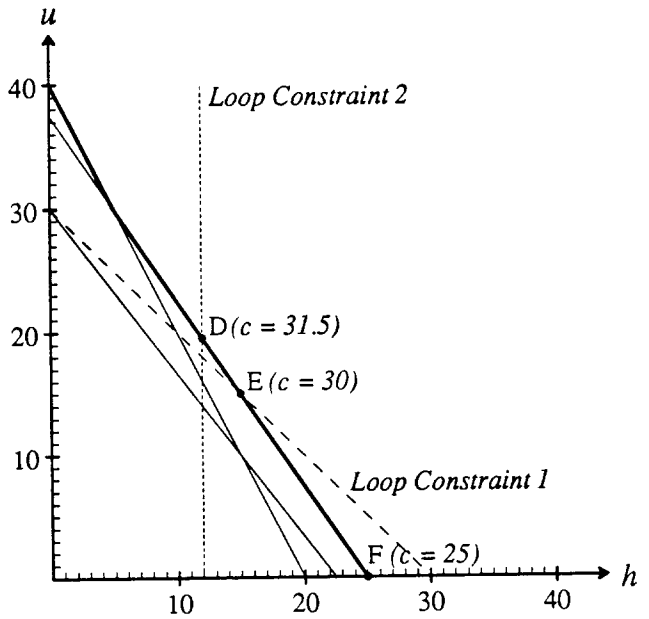


Figure 4-11: Graphical Solution to Jouppi's Loop Example. The default constraints calculated by ATV are the three solid lines; the solution must lie above them (bold line). The problem is to minimize c = u+h while observing all the constraints. The dotted line is the short loop constraint (4.24); the dashed line is the constraint from the large loop (4.23). Point D (h = 12, u = 19.5) is the solution when all constraints are observed; it corresponds to the schedule in Figure 2-9e. Point E (h = 15, u = 15) is the solution when the short loop constraint is disregarded; it corresponds to the desired schedule in Figure 2-9d. Point F (h = 25, u = 0) is the solution when both loop constraints are disregarded and there is no minimum underlap requirement; this is the solution found by the SCAT algorithm (Figure 2-10).

4.4. Critical Paths, Traceback

Recalling the definition of critical paths from Chapter 3, a critical path from some event is a succession of predecessor events, each of which has zero local slack with respect to its successor. Because each event in ATV stores both a list of all its predecessor events and the local slack for each one, the backwards search for critical paths is straightforward. ATV could use standard shortest path algorithms, such as Dijkstra's algorithm [Law76, pp. 70-73] to find the path with least total slack back to some input event; however, this case is simpler because every event should have some predecessor with slack zero (and all other predecessor slacks being non-negative). Thus to find the most critical path for any event, we need merely trace backwards looking for predecessor events with zero slack. A corollary of this is that for any event A with a direct or indirect predecessor event B, there exists a path from some input event I to A passing through B with total slack equal to the total slack from B to A.

Thus to find the n most critical paths from event A, we can start by finding the path with total slack 0, keeping track at each point along the path of what is the predecessor event with the second smallest slack, and storing these in a priority queue. If this event is E_1, with local slack s_1, then there is a path through E_1 to A with total slack s_1, and this is the path with the second smallest total slack. Likewise, when tracing this path back, we note at each step what the predecessor to each event with the second smallest slack is, and store this event in the priority queue with slack s_1+s_i, if s_i is the local slack of this event. In like manner, we can enumerate the n most critical paths, at each step storing events with local slack s_i on a path with slack s_p in the queue with slack s_i+s_p.

Ideally, every model with separately translatable components would identify separate slacks and critical paths for each such component. Using one slack for the entire event time, as currently used in ATV, while simpler, does lead to some anomalies.

For example, consider an asymmetric rise-fall model with single-number components, in which the four event times: A = (5, 17), B = (10, 12), C = (16, 5), and D = (5, 11) are to be merged for longest paths as shown in Figure 4-12. First A and B are merged to produce event time E = (10, 17). Then C and D are merged to produce event time F = (16, 11). Finally, E and F are merged to produce event time G = (16, 17). Now E and F are both critical to G, since neither can be translated forwards without changing the result. Likewise, A and B are both critical to E, and C and D to F. Thus all four of the paths A-E-G, B-E-G, C-F-G, and D-F-G are identified as equally critical to event G, with global slack zero.

However, B and D are in no way critical to event G. Translating either forwards will affect the immediately following event, but only in a component that disappears in the subsequent merge to produce G. Working with individual components would avoid this problem, as it would be clear that E is only critical to G in the falling component, so only those predecessors of E that are critical in this component are of interest in forming the overall critical path. Implementing this in the program would add considerable complexity, though, particularly since the delay operation (as here with inverting delays) can transform which components of earlier event times affect which components of their successors.

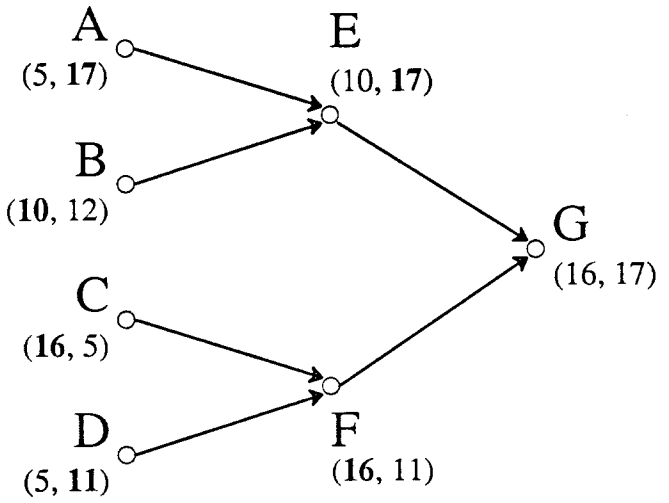


Figure 4-12: False critical paths due to use of a single slack for each event time in the asymmetric rise fall model. Event times are shown as (rise time, fall time). The critical component to each merge is shown in **bold**. All four paths are identified as equally critical, though events B and D do not affect event G.

4.5. Graph Simplification

Since the complexity of the analysis is a function of the size of the timing graph, the user may wish to simplify the graph before beginning the analysis, by consolidating nodes and arcs. Another reason is to reduce the reported length (in nodes) of critical paths that must be examined manually. If two consecutive delays can be replaced by a new delay that has the same effect as the original two delays applied in sequence, the intermediate node could be eliminated, simplifying the graph.

Graph simplification uses the *sum-delays* optional operation to consolidate consecutive delays. In general, if a node has m inputs and n outputs, and every input delay can be combined with every output delay, that node and the $m+n$ arcs attached to it can be replaced with mn arcs that link the inputs to the outputs directly. (See Figure 4-13). If $m=1$ or $n=1$, this will reduce both the number of nodes and the number of arcs by one.

Some restrictions apply to this transformation: if the node is significant for some reason (such as being the controlling node for a gated arc), it cannot be eliminated. Likewise, if any of the output arcs from the node are gated, they cannot be consolidated with previous arcs without changing the behavior of

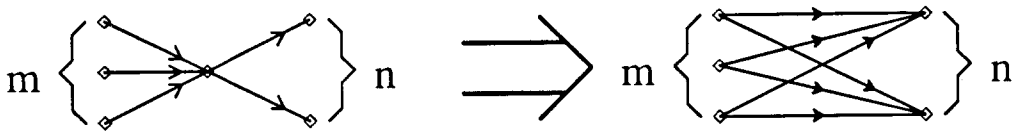


Figure 4-13: Node elimination by combining delays.

the system, since they would need to be replaced with arcs that were "gated in the middle" (see Figure 4-14). (A possible exception to this last would occur if all the input arcs had null delays.) Note, however, that if the gates are edge-triggered (rather than level-sensitive), it should be possible to decouple the gated arc from the input completely, as the only events that can affect an edge-triggered arc are those that occur at the controlling clock node. This may enable additional simplification to occur.

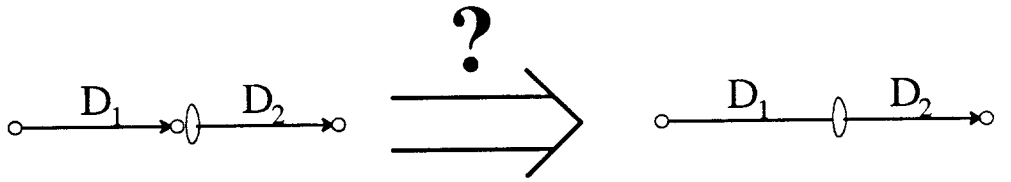


Figure 4-14: Trying to merge these arcs requires putting the resulting gate "in the middle" of the resulting arc.

The current version of graph simplification looks for non-essential nodes with a single ungated fan-out (n=1). Single fan-outs were selected rather than fan-ins because it is more common for nodes to have a single fan-out than a single fan-in; and because a node with an ungated fan-out can still be simplified if some of its fan-ins are gated, but not vice-versa. Each such node is examined in order: if all the input delays can be summed with the output delay, the arcs are merged and the node eliminated. A single pass through all the nodes of the graph suffices, since this simplification operation does not change the fan-out of any node. This operation alone can bring about a dramatic reduction in the depth of the graph: the graph shown in Figure 4-15, with 149 nodes, 199 arcs, and a maximum critical path length of 24, can be reduced to the graph shown in Figure 4-16, with 40 nodes, 90 arcs, and a maximum critical path length of 4.

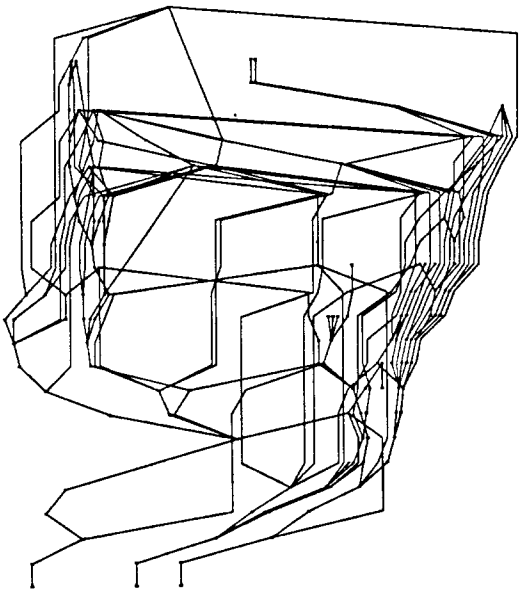


Figure 4-15: Dependency graph before simplification: 149 nodes, 199 arcs.

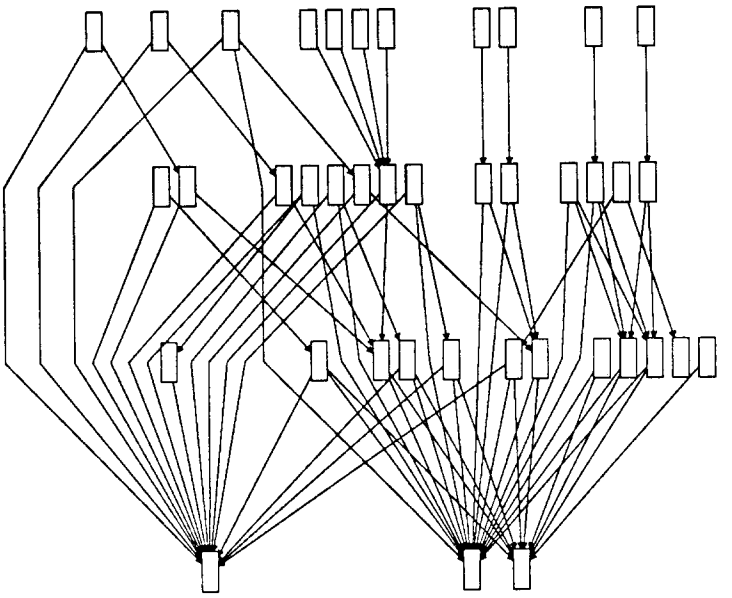


Figure 4-16: Dependency graph after simplification: 40 nodes, 90 arcs.

4.5.1. Limitations

Sum-delays is based on the "natural" timing model for each delay type. For input delays D_1, D_2, we seek an output delay, D_s such that for all input event times E_i, we have:

$$(E_i \oplus D_1) \oplus D_2 = E_i \oplus D_s$$

If such a delay exists, we can write:

$$D_s = D_1 \oplus D_2$$

In effect, this sum-delays operation, where defined, allows us to operate associatively on the delay operation, computing $E_i \oplus D_1 \oplus D_2$ to be computed as: $E_i \oplus (D_1 \oplus D_2)$.

If the resulting simplified graph is used for an analysis using a model with a different delay operation, it is unlikely that the results will be the same as before the simplification for this new model. This is particularly true if the delays are being coerced for the new model. For example, consider the min-max delay format, where the natural implementation of sum-delays is to add the mins together and the maxes together. If this delay format is used for a mean-standard-deviation timing model, taking the min and max to be the three-sigma points, the sum-delay operation will not yield a delay that yields the same results after the coercion as the original delays would after coercion. Thus if $C(D)$ represents the results of coercing D to the new form, in general we have:

$$(E_i \oplus C(D_1)) \oplus C(D_2) \neq E_i \oplus C(D_s)$$

Another issue is the treatment of the merge operation. If the delay operation is not distributive over the merge operation for some model, again graph simplification will not be transparent for this model. Thus, using $M(E_1, E_2)$ to represent the merge of E_1 and E_2, if

$$D_{s1} = D_1 \oplus D_3$$

$$D_{s2} = D_2 \oplus D_3$$

then for this graph simplification to be transparent, we must have:

$$M((E_1 \oplus D_1), (E_2 \oplus D_2)) \oplus D_3 = M((E_1 \oplus D_{s1}), (E_2 \oplus D_{s2})) = M((E_1 \oplus D_1 \oplus D_3), (E_2 \oplus D_2 \oplus D_3))$$

(see Figure 4-17). Since $E_1 \oplus D_1$ and $E_2 \oplus D_2$ may in general represent any arbitrary event times, this is clearly equivalent to requiring that any delay D_3 must distribute over the merge of 2 arbitrary event times - the extension to n event times is clear.

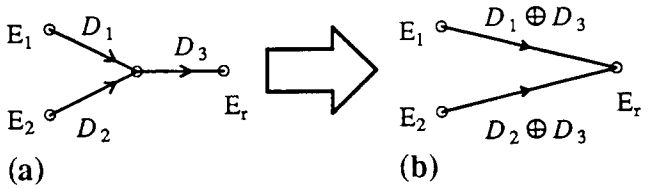


Figure 4-17: Summing delays in graph simplification. In order for the result E_r to be the same both before (a) and after (b) graph simplification, the delay D_3 must distribute over the merge operation.

This property does not hold for all timing models: in particular, it does not hold for the simple mean-standard-deviation model. For example, consider the event times E_1=(15.9,0) and E_2=(7.0,3), the delay D=(16,4), and the comparison point chosen to be the positive 3σ point. Then:

$$M(E_1, E_2) \oplus D = E_2 \oplus D = (23.0, 5),$$

because the 3σ points of E_1 and E_2 are 15.9 and 16.0, respectively, but

$$M(E_1 \oplus D, E_2 \oplus D) = E_1 \oplus D = (31.9, 4),$$

because $E_1 \oplus D$=(31.9,4), with a 3σ point of 43.9, while $E_2 \oplus D$=(23.0,5), with a 3σ point of 38.0.

Likewise in the (time, slope) model, a similar case can occur when a fast-rising input has a midpoint just slightly greater than a slow-rising input: the fast-rising input will be selected if they are merged directly, but if they are both put through a common delay first, the slow-rising input will dominate the results. For the delay (A = 0.2, K = 1.0), the input event time E_1=(1.0, 5.0) produces the output event time (1.73, 2.0), while the input event time E_2=(0.9, 50.0) produces the output event time (1.90, 10.0). Thus E_1 will be selected if the input event times are merged immediately, but E_2 will be selected if the inputs are delayed before the merge.

Both of these models have the properties that the merge operation has sharp discontinuities that are used in these examples. One might conjecture that for models with event times that are n-tuples,

$E \subset \mathbb{R}^n$, and delays that are m-tuples, $D \subset \mathbb{R}^m$, if both the delay operation $\oplus : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}^n$ and the merge operation $M : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}^n$ are continuous, then the delay operation would distribute over the merge operation. This conjecture, however, is false. A simple counter example would be a single-number model (with delay implemented as simple addition) in which the merge operation was defined as multiplication:

$$M(x, y) = xy.$$

Here both the delay and merge operations are continuous, but delay does not distribute over the merge operation:

$$ab + c \neq (a + c)(b + c),$$

in general. (Note that this definition of merge is not acceptable on other grounds: it is not time-invariant with respect to the translation operation. It does, however, suffice to show that continuity of the merge operation is not sufficient to force the delay operation to distribute over the merge.)

The converse is also false: if we were to modify the simple mean, standard deviation model by making the delay operation pass the standard deviation of the input event time through unchanged, this modified delay operation would distribute over the merge operation, even though the merge operation is still not continuous.

Even though delay does not distribute over merging for the simple mean-standard-deviation model or the (time, slope) model, there still exist individual delays for which this distributive property will hold. In particular, the null delay, N, will always distribute over any merge: since the definition of N is that $E \oplus N = E$, for all E, it must be true that:

$$M(E_1, E_2) \oplus N = M(E_1, E_2) = M(E_1 \oplus N, E_2 \oplus N)$$

Likewise, any delay similar to a null delay (e.g., the mean-standard-deviation delay: (0.0, 0.0)) will exhibit this property.

Another problem is that the required delay sum may not even exist in closed form, or may not be readily computable.

4.5.2. Defining Sum-Delays for Various Models

The sum-delays operation takes various forms for different models. For those models where delays and event times have the same format, sum-delays is often defined in the same manner as the delay operation for that model. This is the case, for example, for the single-number and min-max models. This would also be the case for mean-standard-deviation, but the non-transparent nature of sum-delays for the simple mean-standard-deviation model may make it better to leave sum-delays undefined in this case. The nominal-worst delay format acts like a min-max format (summing components separately).

For all delay models, the sum of any delay D with a null-delay N is defined as:

$$D \oplus N = N \oplus D = D$$

Two delay unions are summable if all of their respective components, taken in order, are summable. This restriction is to ensure that the result will be as transparent as possible.

Asymmetric delays can sometimes be summed. If neither delay is of UNKNOWN type, then the delays can be summed if the appropriate components of each can be summed, with the type of the resulting delay being INVERTING if one of the input delays is INVERTING, NONINVERTING otherwise. UNKNOWN delays can in general only be summed if they are the first delay in the sequence of delays to be summed. Recall the definition of the delay operation for the asymmetric delay model. For INVERTING delays:

$$(E_r, E_f) \oplus (D_r, D_f) = (E_f \oplus D_r, E_r \oplus D_f)$$

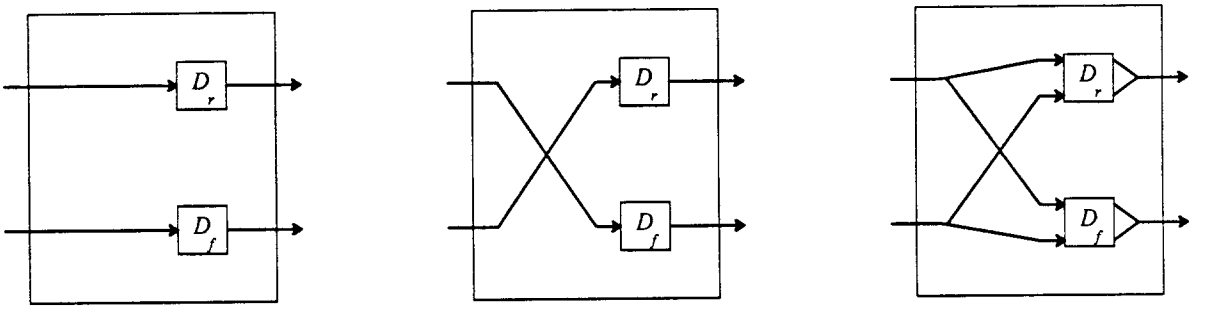
For NONINVERTING delays:

$$(E_r, E_f) \oplus (D_r, D_f) = (E_r \oplus D_r, E_f \oplus D_f)$$

For UNKNOWN delays:

$$(E_r, E_f) \oplus (D_r, D_f) = (M(E_f \oplus D_r, E_r \oplus D_r), M(E_r \oplus D_f, E_f \oplus D_f))$$

These three definitions are shown graphically in Figure 4-18.



NONINVERTING Delay INVERTING Delay UNKNOWN Delay

Figure 4-18: The three types of asymmetric delay. The upper input in each case is the input rising event time, the lower input is the input falling event time.

Now to sum two delays, we seek D_s such that

$$E \oplus D_1 \oplus D_2 = E \oplus D_s$$

for all E. Thus, to sum two INVERTING delays D_1 and D_2, we have:

$$(E_r, E_f) \oplus (D_{r1}, D_{f1}) \oplus (D_{r2}, D_{f2}) = (E_f \oplus D_{r1}, E_r \oplus D_{f1}) \oplus (D_{r2}, D_{f2})$$

$$= (E_r \oplus D_{f1} \oplus D_{r2}, E_f \oplus D_{r1} \oplus D_{f2})$$

$$= (E_r \oplus (D_{f1} \oplus D_{r2}), E_f \oplus (D_{r1} \oplus D_{f2})),$$

so that if the appropriate component delays are summable, we can form:

$$D_{rs} = (D_{f1} \oplus D_{r2}),$$

$$D_{fs} = (D_{r1} \oplus D_{f2}).$$

The resulting delay is evidently of NONINVERTING type, as determined by pattern-match with the rules for the three delay types (see Figure 4-19d). A similar calculation yields the rules for combining NONINVERTING with NONINVERTING, NONINVERTING with INVERTING, and INVERTING with NONINVERTING delay types.

Now suppose that D_1 is of UNKNOWN type, while D_2 is NONINVERTING. Then:

$$(E_r, E_f) \oplus (D_{r1}, D_{f1}) \oplus (D_{r2}, D_{f2}) = (M(E_f \oplus D_{r1}, E_r \oplus D_{r1}), M(E_r \oplus D_{f1}, E_f \oplus D_{f1})) \oplus (D_{r2}, D_{f2})$$

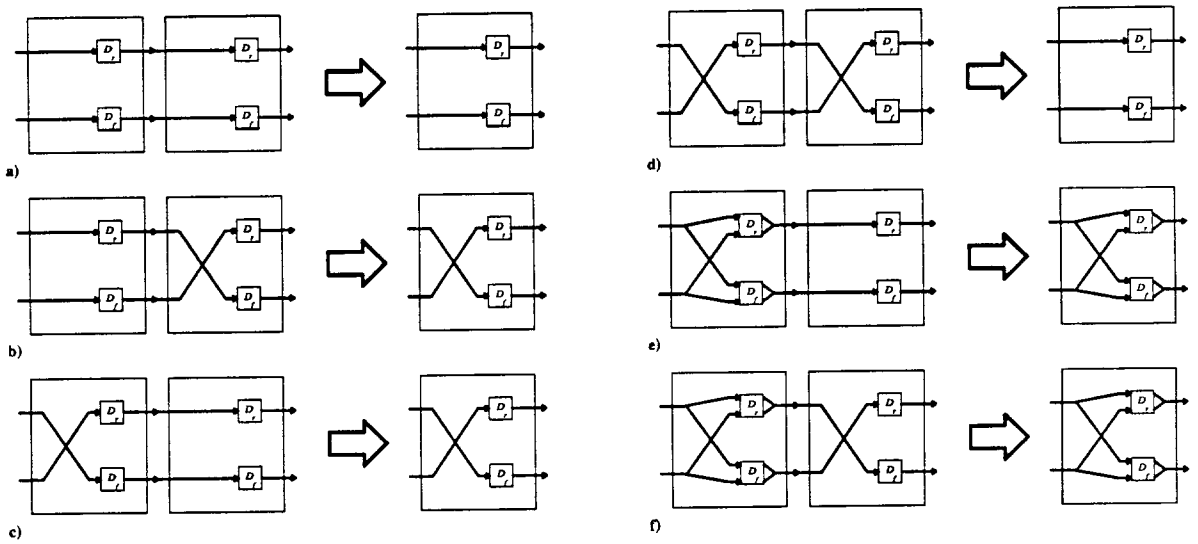


Figure 4-19: The six legal sums of two asymmetric delays. (a): NONINVERTING ⊕ NONINVERTING, (b): NONINVERTING ⊕ INVERTING, (c): INVERTING ⊕ NONINVERTING, (d): INVERTING ⊕ INVERTING, (e): UNKNOWN ⊕ NONINVERTING, (f): UNKNOWN ⊕ INVERTING. The latter two depend on the delay operation distributing over the merge operation for the component model.

$$= (M(E_f \oplus D_{r1}, E_r \oplus D_{r1}) \oplus D_{r2}, M(E_r \oplus D_{f1}, E_f \oplus D_{f1}) \oplus D_{f2})$$

If the delay operation for the components distributes over the merge operation, then this:

$$= (M(E_f \oplus D_{r1} \oplus D_{r2}, E_r \oplus D_{r1} \oplus D_{r2}) M(E_r \oplus D_{f1} \oplus D_{f2}, E_f \oplus D_{f1} \oplus D_{f2})$$

$$= (E_r, E_f) \oplus (D_{rs}, D_{fs}),$$

for UNKNOWN type delay D_s with components:

$$D_{rs} = (D_{r1} \oplus D_{r2}),$$

$$D_{fs} = (D_{f1} \oplus D_{f2})$$

(Figure 4-19e).

On the other hand, if D_1 is NONINVERTING and D_2 is UNKNOWN, then

$$(E_r, E_f) \oplus (D_{r1}, D_{f1}) \oplus (D_{r2}, D_{f2}) = (E_r \oplus D_{r1}, E_f \oplus D_{f1}) \oplus (D_{r2}, D_{f2})$$

$$= (M(E_f \oplus D_{f1} \oplus D_{r2}, E_r \oplus D_{r1} \oplus D_{r2}) M(E_r \oplus D_{r1} \oplus D_{f2}, E_f \oplus D_{f1} \oplus D_{f2}),$$

and since there are four different delay combinations here, there is no way to reduce this to one of the three basic types of asymmetric delay discussed above. (See Figure 4-20a). Similar problems occur if D_1 is INVERTING or UNKNOWN while D_2 is UNKNOWN; thus the sum-delays operation is undefined if the second operand is of UNKNOWN type.

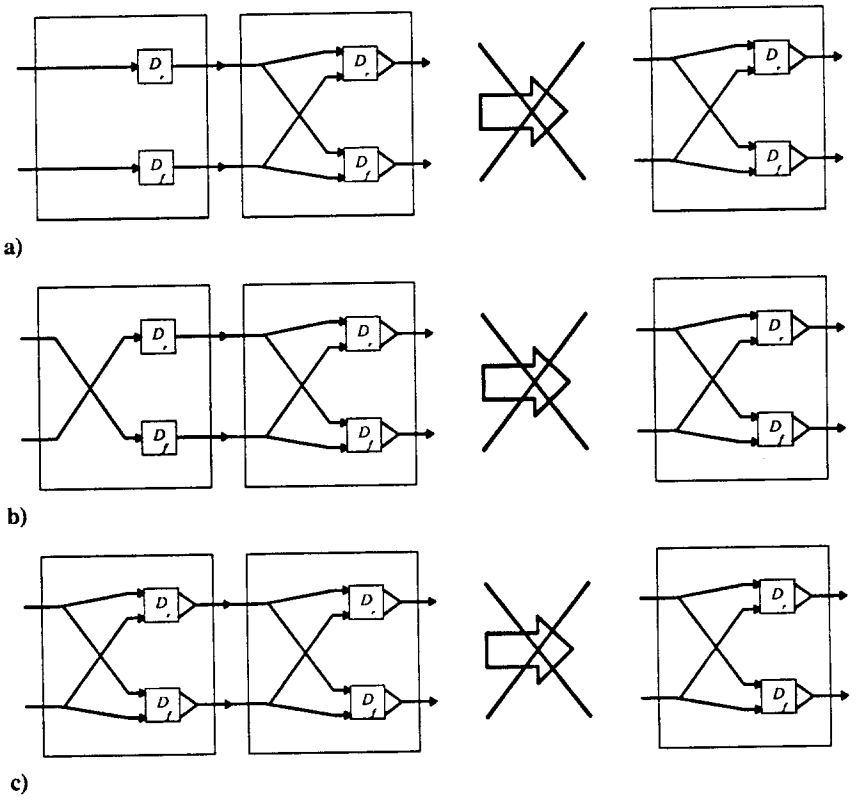


Figure 4-20: The three illegal sums of two asymmetric delays. (a): NONINVERTING ⊕ UNKNOWN, (b): INVERTING ⊕ UNKNOWN, (c): UNKNOWN ⊕ UNKNOWN. Any delay ⊕ UNKNOWN is undefined.

The six legal combinations of two asymmetric delays are shown in Figure 4-19. The complete rules for calculating the sum of two asymmetric delays, $D_1 \oplus D_2$ are:

1) If the type of D_2 is INVERTING, then the components of the sum, D_s are:

$$D_{rs} = (D_{f1} \oplus D_{r2}),$$

$$D_{fs} = (D_{r1} \oplus D_{f2}).$$

The type of the result is NONINVERTING if D_1 is also of type INVERTING, INVERTING if D_1 has type NONINVERTING, and UNKNOWN if D_1 has type UNKNOWN.

2) If the type of D_2 is NONINVERTING, then the components of the sum, D_s are:

$$D_{rs} = (D_{r1} \oplus D_{r2}),$$

$$D_{fs} = (D_{f1} \oplus D_{f2}).$$

The type of the result is the same as that of D_1.

3) If the type of D_2 is UNKNOWN, the result is undefined.

4.5.3. Other Forms of Simplification

Other forms of graph simplification may also be possible by defining additional operations. One operation that would be useful would be *merge-delay*, which would allow multiple arcs connecting the same pair of nodes to be collapsed into a single arc. Such nodes can occur naturally within a timing graph when nodes represent groups of terminals: e.g., all 32 inputs of a 32-bit latch grouped into a single node. There could then be multiple arcs representing the different interconnects on individual bits of the latches.

The requirement for merge-delay of two delays, D_1 and D_2, is that the resulting delay, D_m, should observe:

$$E \oplus D_m = M(E \oplus D_1, E \oplus D_2),$$

for all event-times E. Given this operation, parallel arcs with mergeable delays can be collapsed into a single arc.

This merge-delay operation would be useful in another context: coercing from asymmetric delays to one of the component formats (see next section). Without it, each target format must have its own coercion defined from an asymmetric delay; using merge-delay, a generic coercion can be defined from asymmetric-delay to any delay format that has merge-delay defined. The coercion is to coerce each of the rising and falling delay components to the target format separately, and then to merge-delay the two

of them together to yield the final result.

Yet another form of graph simplification was shown in the NELTAS system [NST82], where some internal paths between registers in a single module were eliminated entirely and reduced to constraints on the incoming clocks, which were then passed to higher hierarchical levels along with the simplified graph structure. This approach relied on the fact that they were only using edge-triggered registers, which could then be decoupled from the rest of the circuit. This strategy is more difficult in an abstract framework, but perhaps the constraint-satisfier operation could be used to generate such constraints.

Chapter 5 - Results

1. CRITICAL PATH ANALYSIS

ATV consists of approximately 4300 lines of Lisp code, including blanks and comments. The largest example it has been run on was a chip implementing the Data Encryption Standard, containing about 20,000 transistors. This chip was synthesized using standard cell components from the Mississippi State University CMOS double-layer metal library. The timing graph before simplification for this chip had 6982 nodes and 9533 arcs. After simplification, the graph had 3016 nodes and 5567 arcs.

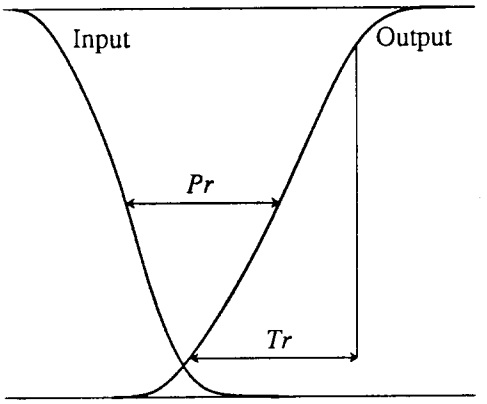


Figure 5-1: Propagation and Transmission Delays. Propagation delay is time from 50% mark of input to 50% mark of output response; transmission delay is time from 10% to 90% of output response.

Component delays were taken from the data sheets supplied with the library. These delays were specified as nominal and worst case transmission and propagation delays for both rising and falling outputs, specified to hundredths of nanoseconds. Transmission and propagation delays are shown in Figure 5-1. Ideally, both the transmission and propagation delays would be entered for each component, yielding one of the delay structures shown in Figure 5-2, which would then be coerced to the desired form. However, in order to simplify the problem, only the transmission delays were entered into the library, in the structure shown in Figure 5-3.

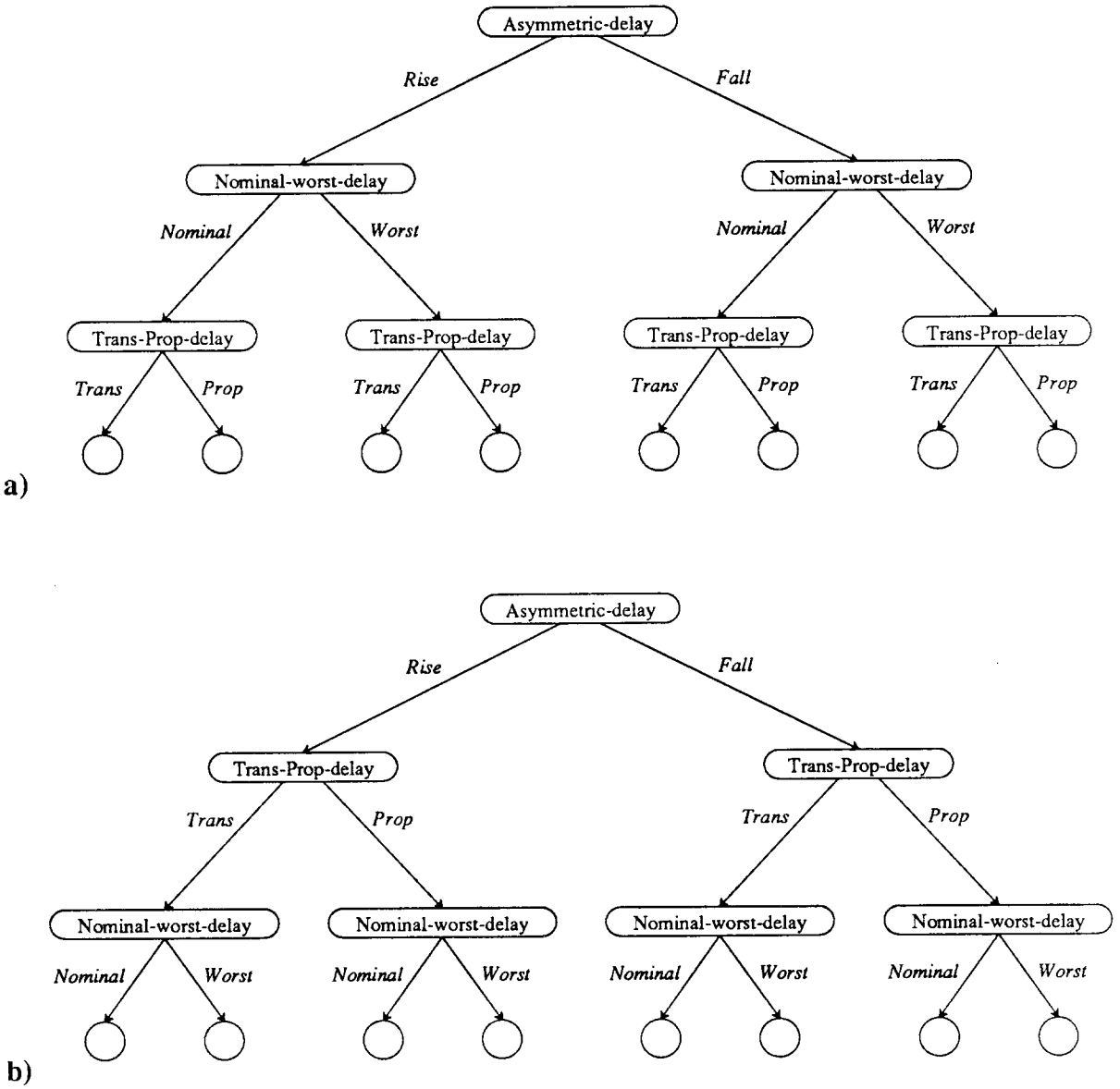


Figure 5-2: Ideal delay structure for representing delays of the MSU cells. Either (a) or (b) could be chosen. The two representations are conceptually identical, but present different grouping of delay information to coercion routines that proceed top-down.

For this example, interconnect delays were assumed to already be incorporated in the component delays, and thus arcs representing interconnect were generated with null delays. In a few cases where component data sheets were missing, delays from closely related parts were used instead; thus the 1860 cell, a four input or-and-invert: ((a+b)(c+d))' , was modeled using the data from the 1870 cell, a four

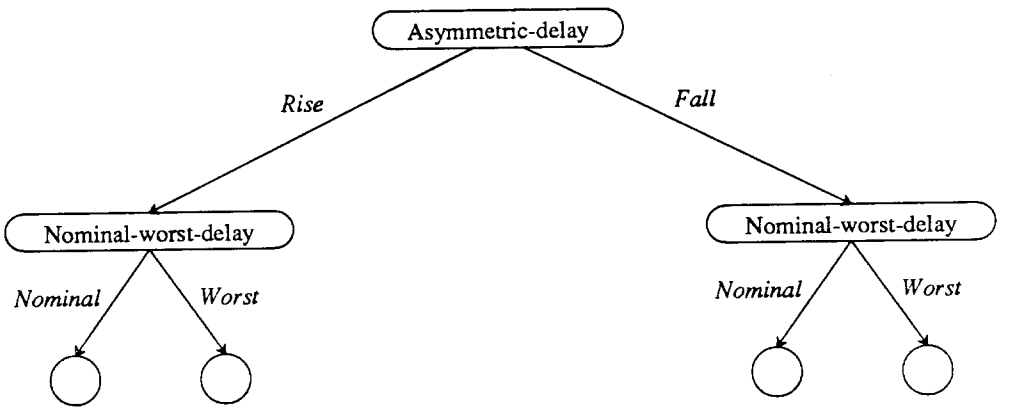


Figure 5-3: The actual delay structure used for representing delays of the MSU cells.

input and-or-invert: **(ab+cd)'** . This was consistent with the main purpose of this exercise, which was to identify potential critical paths from information available early in the design cycle, and observe how the choice of timing model affects the constraints and critical paths reported, starting from the same raw input data. One can only wonder how many industrial cell libraries may contain data of equally questionable ancestry.

The delays were entered into the Oct database in asymmetric nominal-worst form, and then coerced in ATV to get single_number, min_max, and mean_standard_deviation delays. Figure 5-4 shows the coercions used. For nominal-worst delay (n, w), the single_number model set delay$=n$, the min_max model set max$=w$, min$=n-(w-n)=2n-w$, subject to the restriction that min≥ 0, and the mean_standard_deviation model set $\mu=n$, $\sigma=(w-n)/3$. An alternate coercion to the min_max type would interpret n as the geometric mean rather than the arithmetic mean of min and max, setting min$=n^2/w$. This would keep many of the minimum delays from being set to zero when w is much larger than n, as was often the case here. This would not affect any of the long path calculations, which depend only on the max value, but would affect constraint calculations, reducing the required cycle length.

When coercing to a single delay (i.e., not asymmetric), the rising and falling delays were coerced to the target delay independently, and then the two delays were merged using the rules for long-path

event-time merging for each of the models. Because each of these models uses the same format for event times and delays, this merge was a natural choice; in general, an operation such as *merge-delays*, described in Chapter 3, should be defined to coerce from an asymmetric-delay to the types of its component elements.

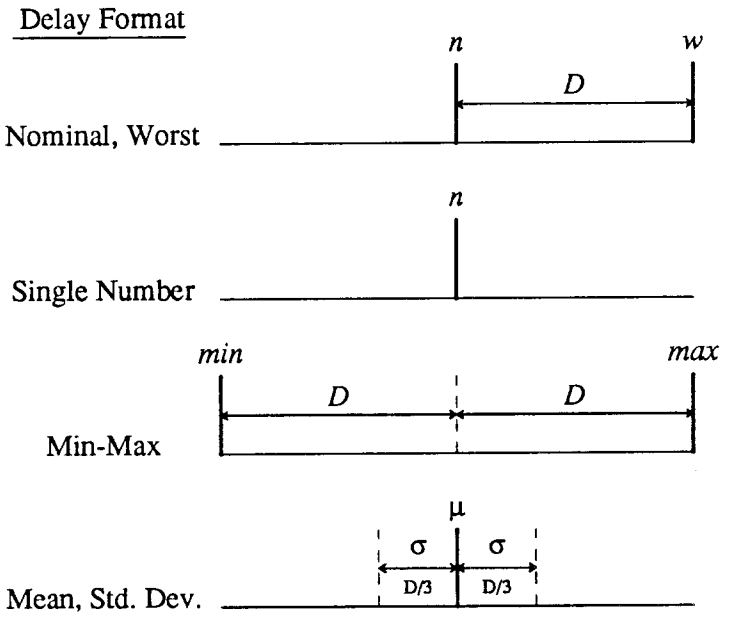


Figure 5-4: Coercions used from (nominal, worst) form to other delay formats.

1.1. Chip Clocking

The DES chip uses 3 off-chip clocks, shown in Figure 5-5. The edges of RDCLK and WRCLK are synchronized to the corresponding edges of SYSCLK, as shown. The basic machine cycle was defined as two cycles of SYSCLK. The analysis was conducted over two basic machine cycles, to account for all the possible interactions between the different clocks. The DES chip does not use any transparent latches: all the storage elements are master-slave latch combinations. Thus there were no paths extending over multiple cycles.

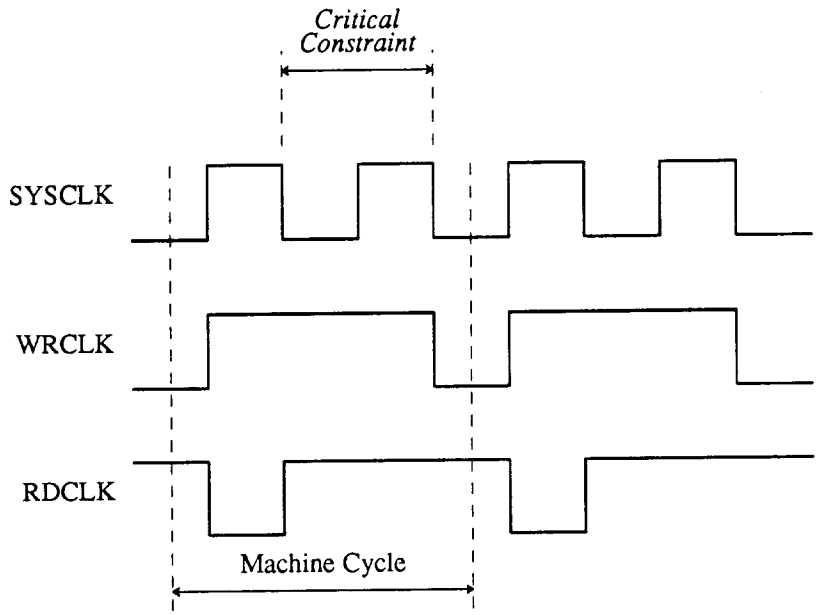


Figure 5-5: The three clocks defined for the DES chip.

1.2. Comparison of Results for Different Models.

The simplified DES graph was analyzed with three different models: the asymmetric versions of the single_number, min_max, and mean_standard_deviation models. The analysis for the asymmetric mean_standard_deviation model was also performed on the unsimplified graph to analyze the effects of the simplification for this model. Recall that the simplification operation should leave the results unchanged for the single_number and min_max models, but not the mean_standard_deviation model. All three models agreed that the critical constraint determining the length of a machine cycle was from one falling edge of SYSCLK to the next falling edge of SYSCLK (as shown in Figure 5-5). As expected, the single_number model was most optimistic (yielding a critical constraint of 236.6ns), the min_max model was most pessimistic (533.1ns), and the mean_standard_deviation model was in between (323.9ns). The mean_standard_deviation model on the unsimplified graph calculated this constraint as 283.1ns. The asymmetric mean_standard_deviation combination on the simplified graph identified the greatest number of critical paths, including all the paths that were identified as critical by any other model. These paths are shown in Figure 5-6. The path starts with a falling event at node SYSCLK, which propagates to the

controlling clock node for the top arc (from nodes 1830.I to [260]), becomes a CH-ST event initiated on that arc, and propagates down to the bottom of the graph, where it is compared against another falling event from SYSCLK.

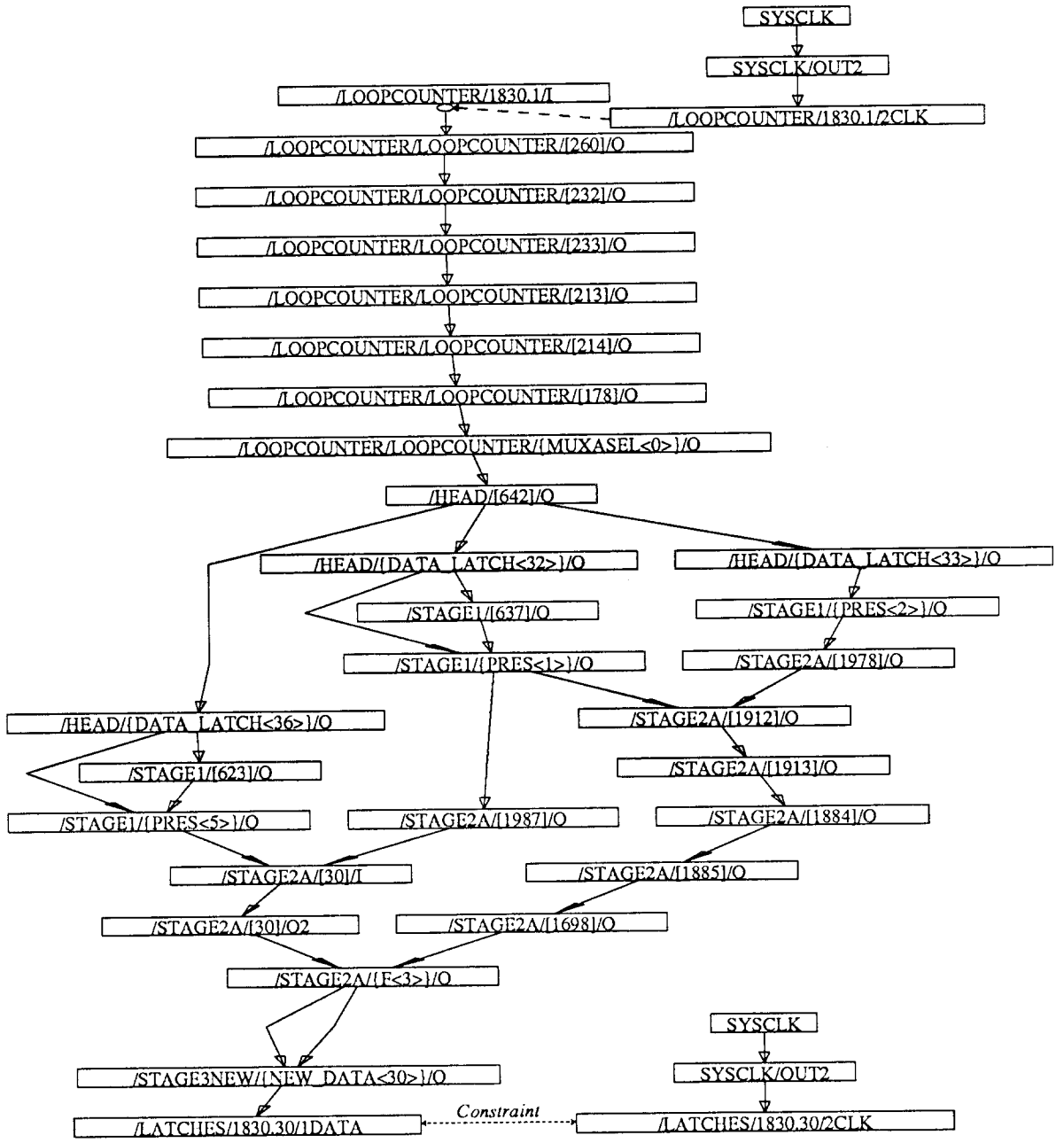


Figure 5-6: Critical nodes and arcs of the simplified DES chip, asymmetric mean_standard_deviation model.

This subgraph was extracted from the full chip, and three more analyses were run on these paths, yielding critical path information for the straight (non-asymmetric) single_number, min_max, and mean_standard_deviation models. The results of these runs are compared in the next set of figures. The differences in the critical paths observed by the different models are all located in the section of the graph of Figure 5-6 between nodes /HEAD/[642]/O and /STAGE2A/{F<3>}/O. This section has been extracted and displayed for each model. Figure 5-7 shows the general form that will be used for presenting the results. Nodes are labeled with a short identifying tag extracted from the name of the node. The delays for each arc are shown in italic type next to the arc. In this figure, symbolic names are used for the delays that suggest the elements that arc represents, such as *invl* for an inverter. Event times are shown beside each node in regular type. When two arcs meet at a node, each arc is labeled with the (delayed) event it contributes to the merge, along with the associated slack in bold type. The one exception is node [30]/I: since both incoming arcs have null delays, the events being merged are just the events from the preceding nodes. Inverting arcs are indicated with bubbles. Paths that are critical to the final event are shown with heavy arcs.

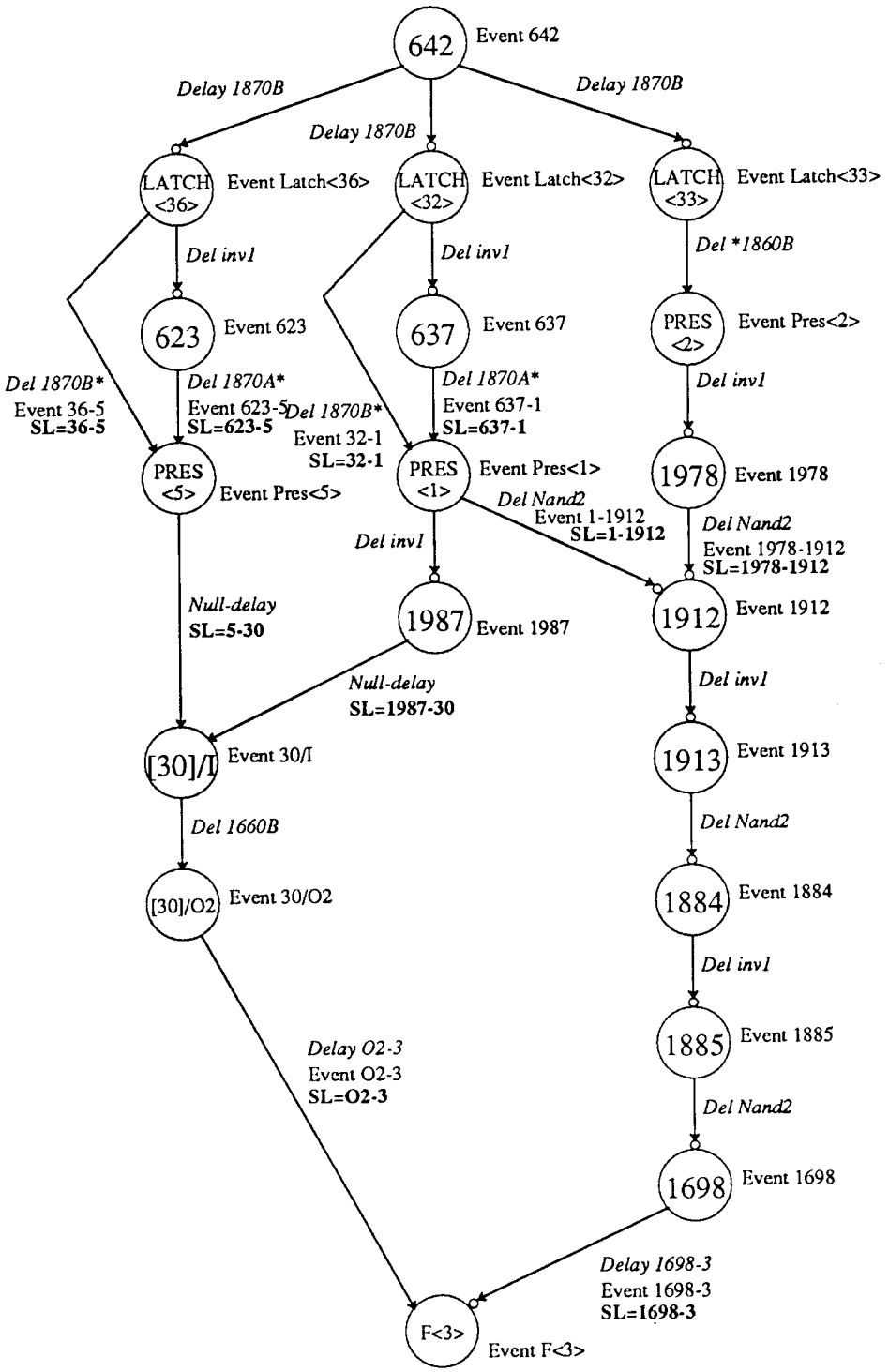


Figure 5-7: Format for presenting results.

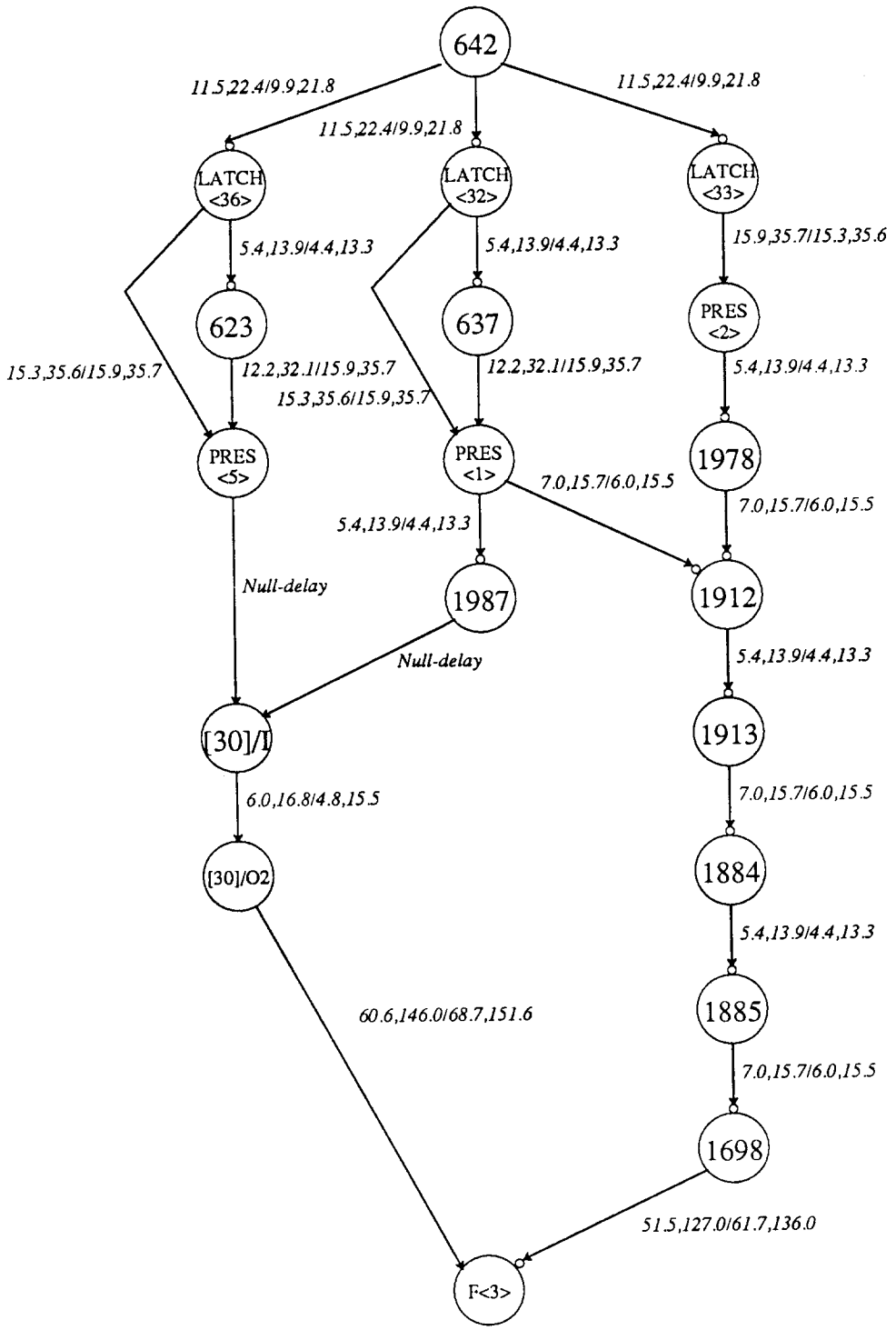


Figure 5-8: Input delay values (nominal and worst case): (RN, RW)/(FN, FW)

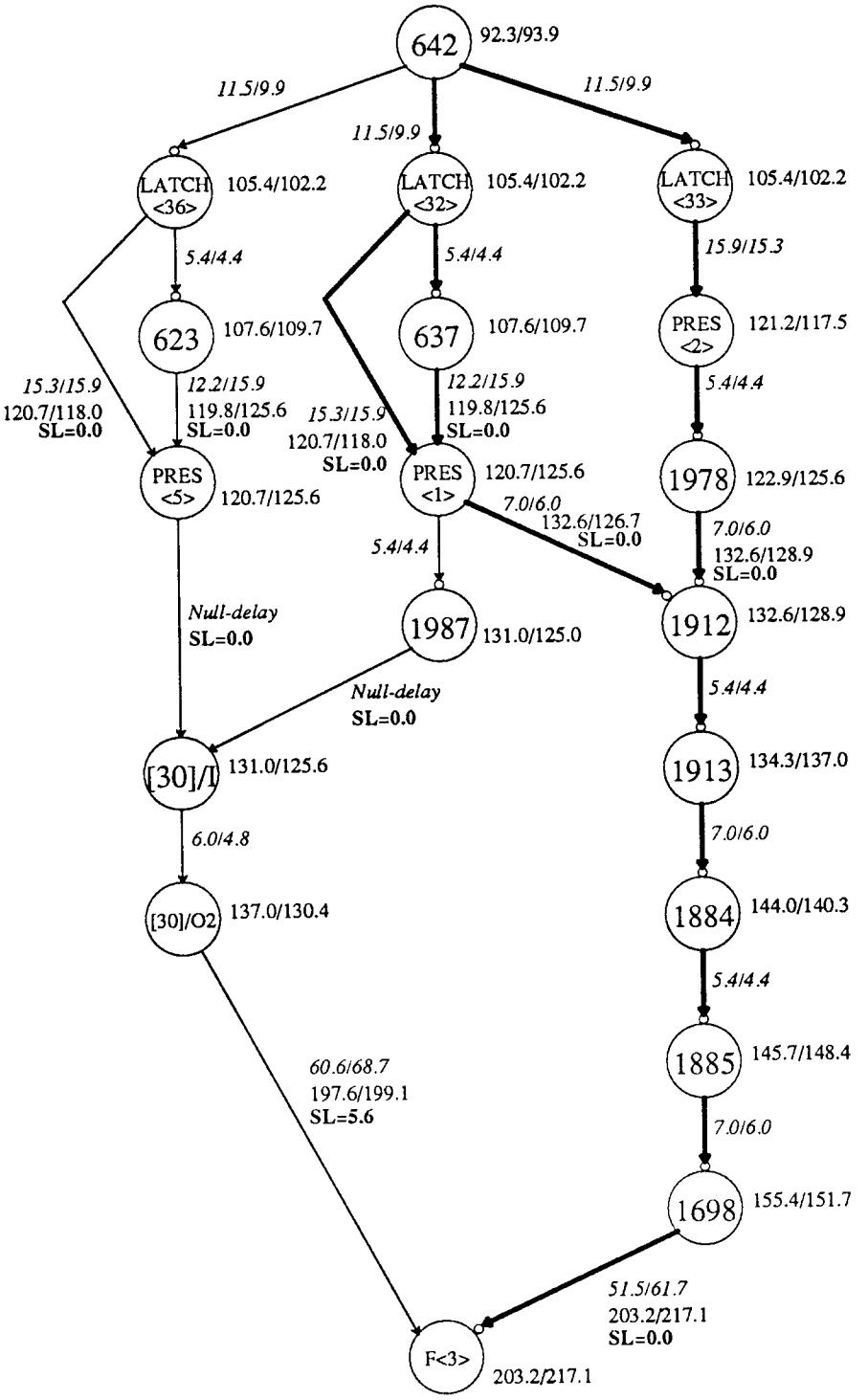


Figure 5-9: Asymmetric Single Number Results: Rise/Fall. Numbers may not add exactly due to rounding.

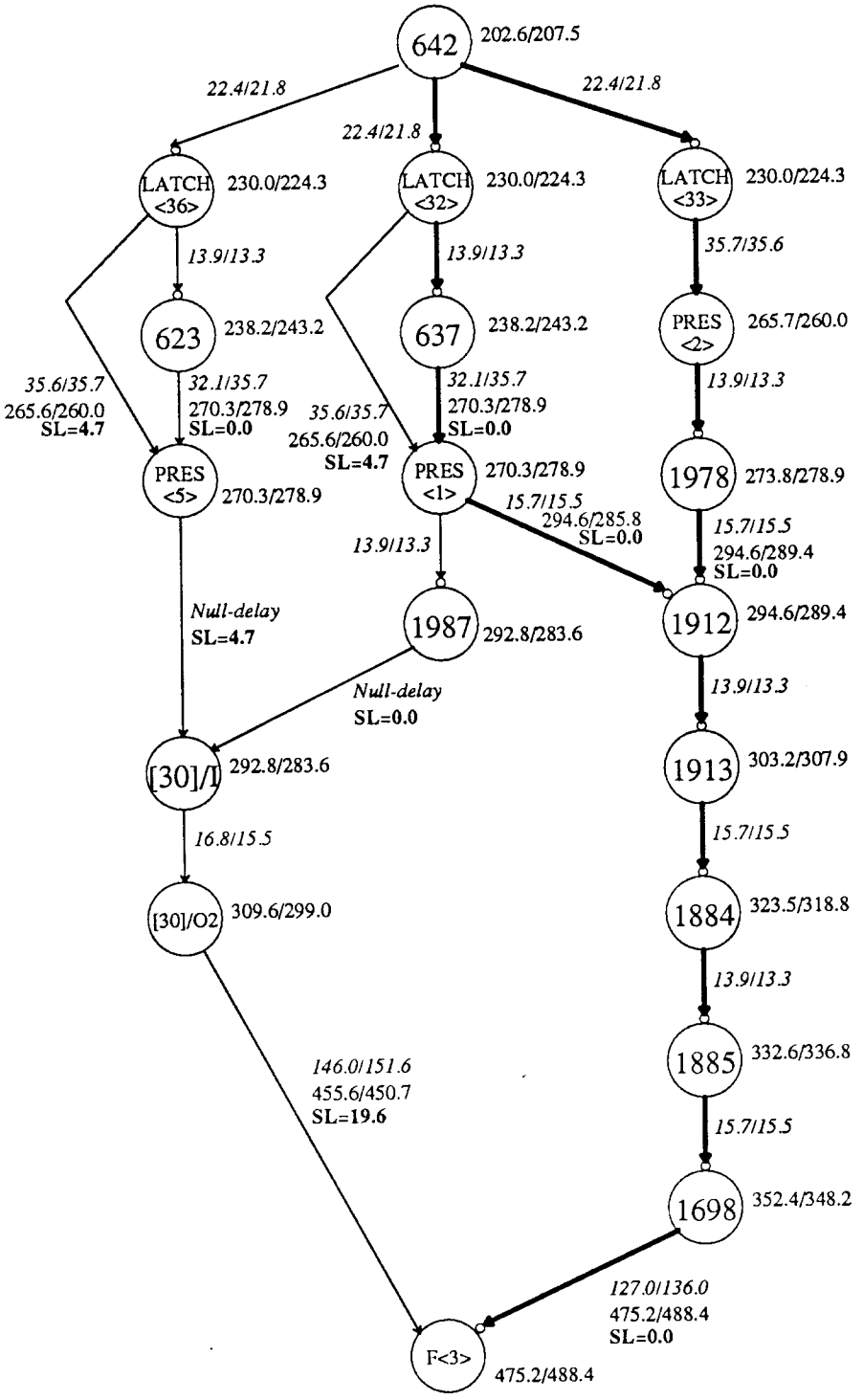


Figure 5-10: Asymmetric Min_Max Results: Rise/Fall (Max times only shown). Numbers may not add exactly due to rounding.

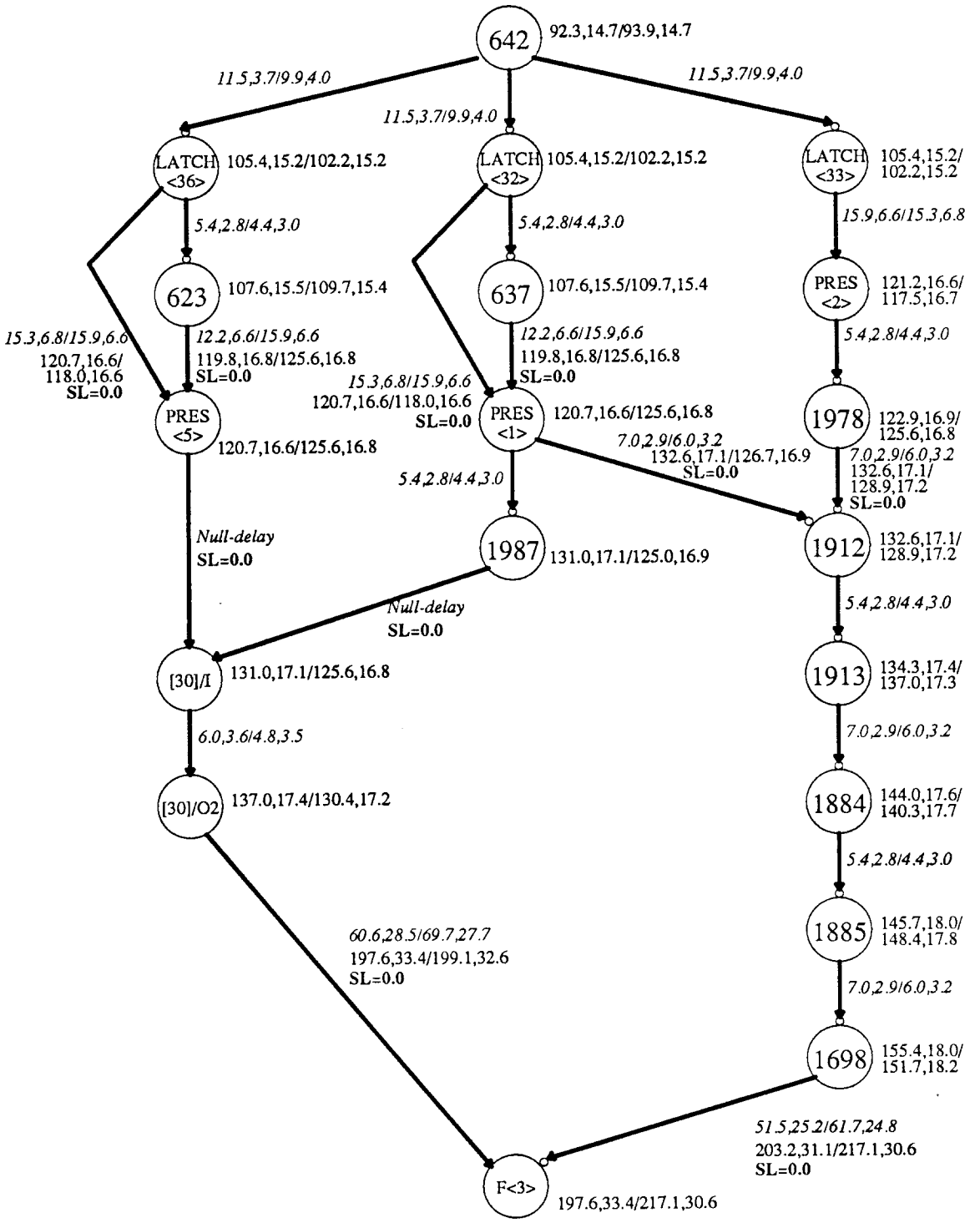


Figure 5-11: Asymmetric Mean & Standard Deviation Results: Rise/Fall. Numbers may not add exactly due to rounding.

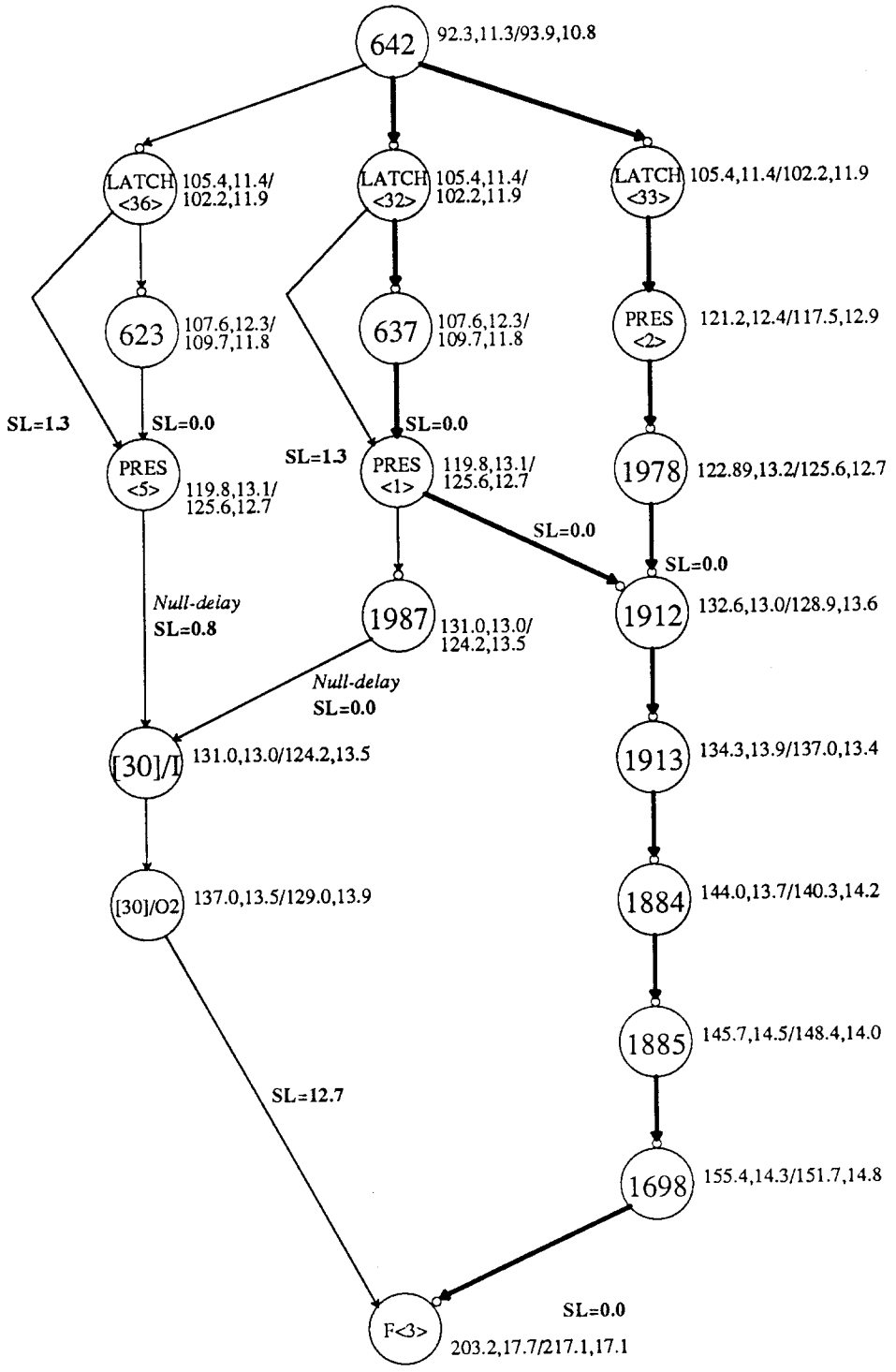


Figure 5-12: Asymmetric Mean & Standard Deviation Results for Unsimplified Graph. Delays have been omitted, as they do not always make sense in terms of the simplified graph.

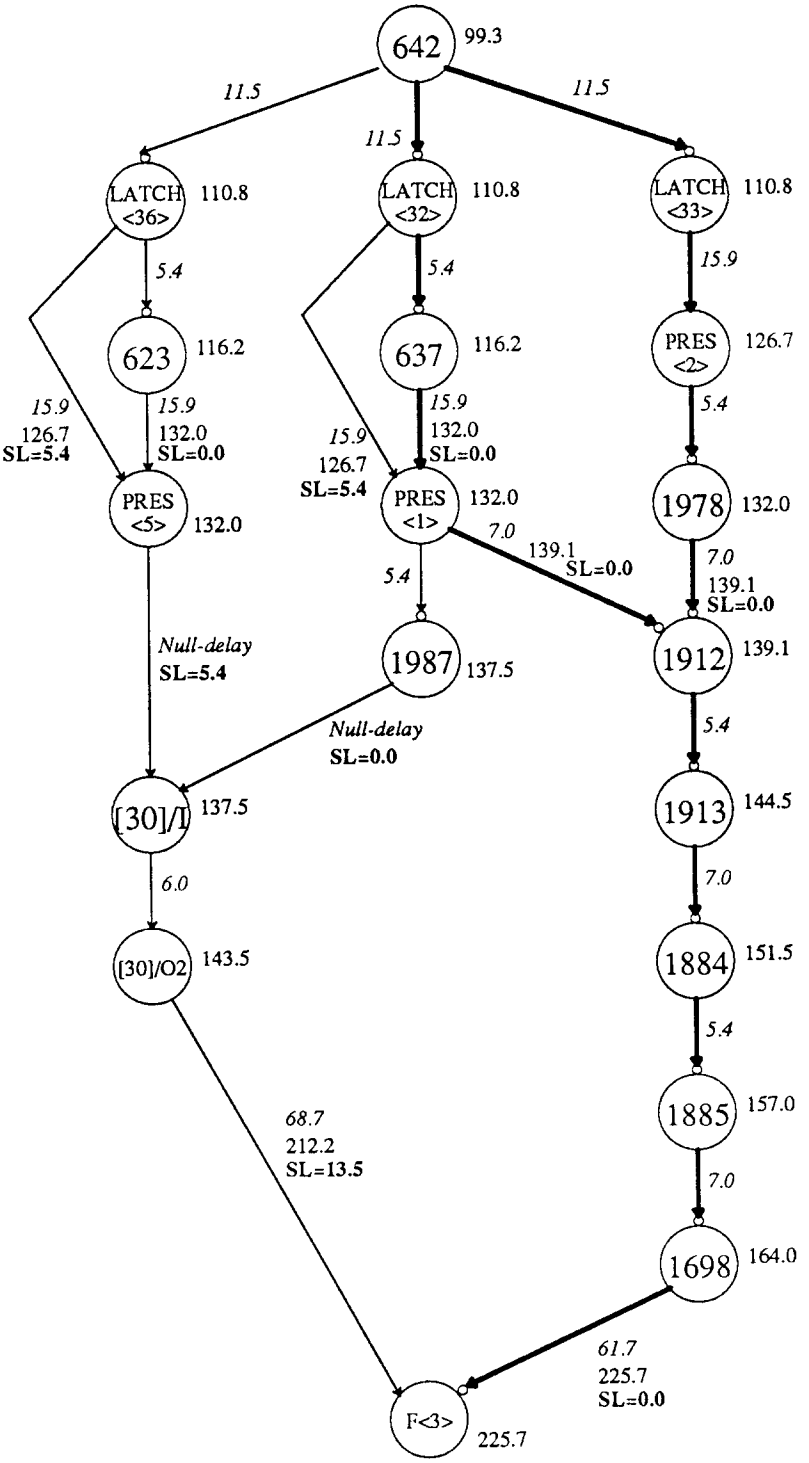


Figure 5-13: Single_Number Results. Numbers may not add exactly due to rounding.

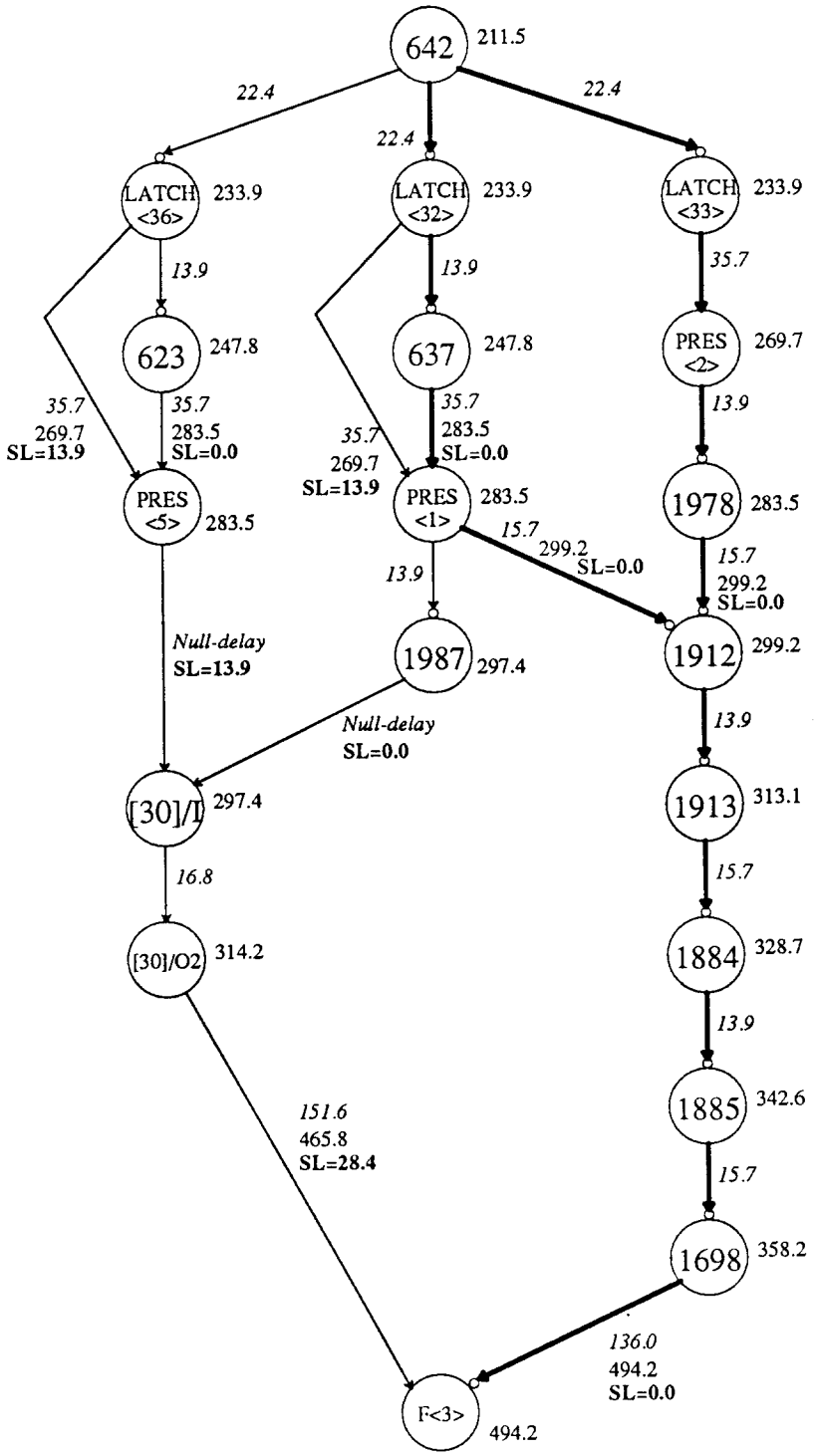


Figure 5-14: Min_Max Results: (Max times only shown). Numbers may not add exactly due to rounding.

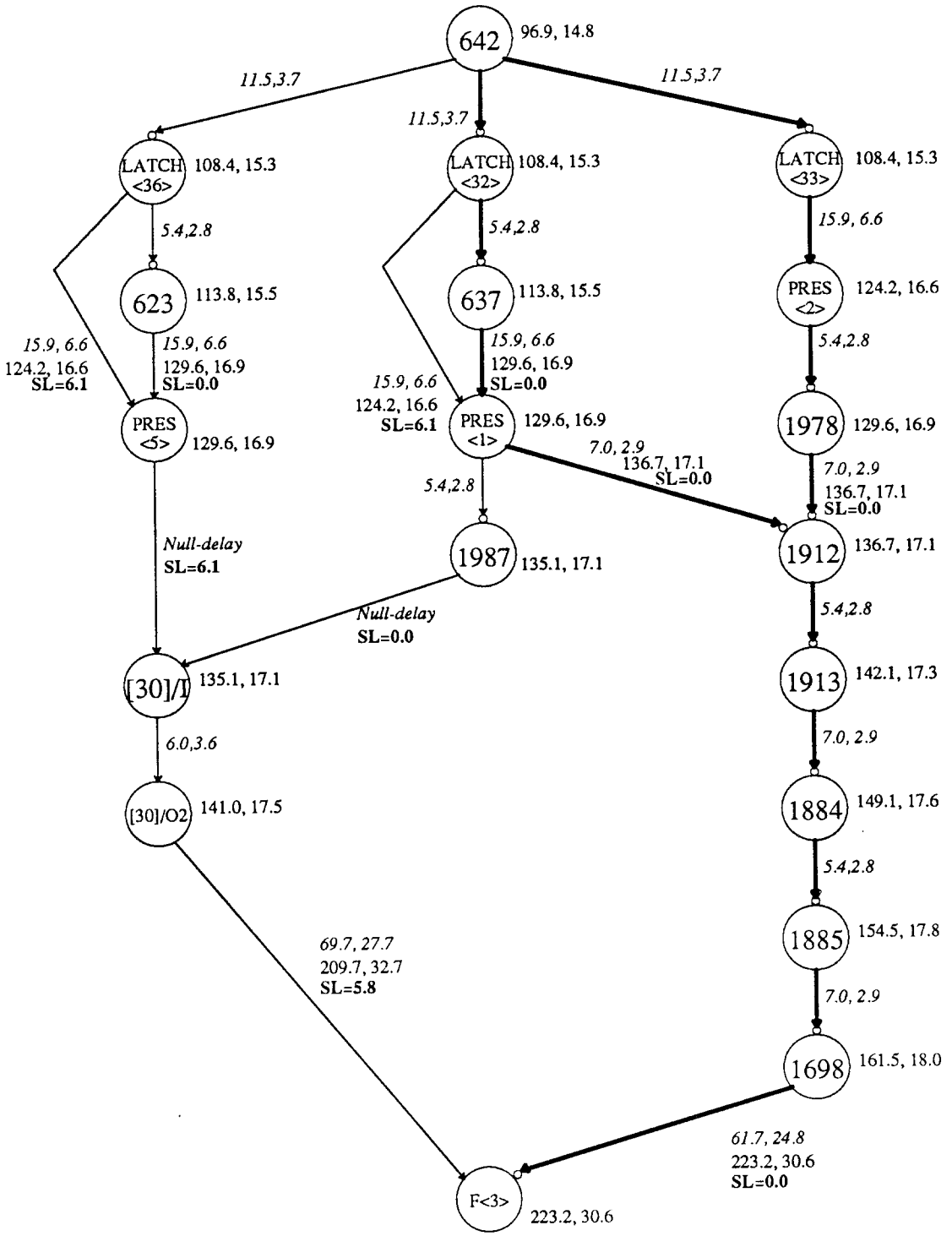


Figure 5-15: Mean & Standard Deviation Results. Numbers may not add exactly due to rounding.

The input delay values for each arc are shown in Figure 5-8, rounded for display purposes. Figure 5-9 shows the results of an asymmetric single_number analysis, with a total of 3 critical paths through this part of the graph. Figure 5-10 shows the results for the asymmetric min_max analysis. Only the max times are shown, because only they are relevant to a long-path search (as the event type is CH-ST). For purposes of determining this critical path, this model acts like a single_number model using worst case rather than nominal values; the difference between min_max and such a model comes when the actual constraint spacing is calculated, when the max value from these paths is compared to the min value along the path from the clock. Here only two paths are critical: the direct path from LATCH<32> to PRES<1> has additional slack and is not part of the critical path for these data values. Figure 5-11 shows the critical paths for the asymmetric mean_standard_deviation model, a total of seven critical paths. Figure 5-12 contrasts this with the results for the asymmetric mean_standard_deviation model on the unsimplified graph. This figure shows event times at those nodes that also appear in the simplified graph. Most arcs in the simplified graph shown here correspond to multiple arcs in the original graph, so it is difficult to express the unsimplified delays in terms of the simplified graph, and most of the paths merging at individual nodes in the simplified graph actually merge some distance back from these nodes and share one or more final segments in the unsimplified graph, so the input events to a merge do not bear an obvious relation to the final event at the end of that path. Figures 5-13, 5-14, and 5-15 show the results of the single component versions of these three models. All three agree on the same two critical paths.

1.2.1. Comparison of Critical Paths

All the models agree that the two paths from node 642 to 1912 (passing through 637 and 1978, respectively) are equally critical, even though the logic functions are not quite identical. The reason for this may be seen by examining the logic diagram for this section of the chip (Figure 5-16). Because the or-and-invert cell is being modeled as an and-or-invert (as noted above), both paths go through an inverter, the and-or-invert delay, and another inverter. The only difference is that the path through PRES<2> goes through the lower group of inputs to the gate, which have the same rising delay but not

falling delay as the upper group of inputs. Because this rising delay is the dominant delay of the two, both paths appear equally critical whether or not an asymmetric delay model is being used. The two paths are grouped differently by graph simplification, but in each case, the program folds one of the inverters in with the larger gate. Thus the grouping did not affect the distinction between these critical paths for any of the models studied.

The asymmetric single_number and asymmetric min_max models differ in their assessment of the direct path from LATCH<32> to PRES<1>. Both these models agree that the delays along this direct path are greater than the total delay through node 637; whether or not it is critical depends on the differences between the delays vs. the skew between the rising and falling edges at the origin node. This is explained in Figure 5-17, below. The non-asymmetric models are not concerned with this difference between the phases, so all of them disregard this shorter path.

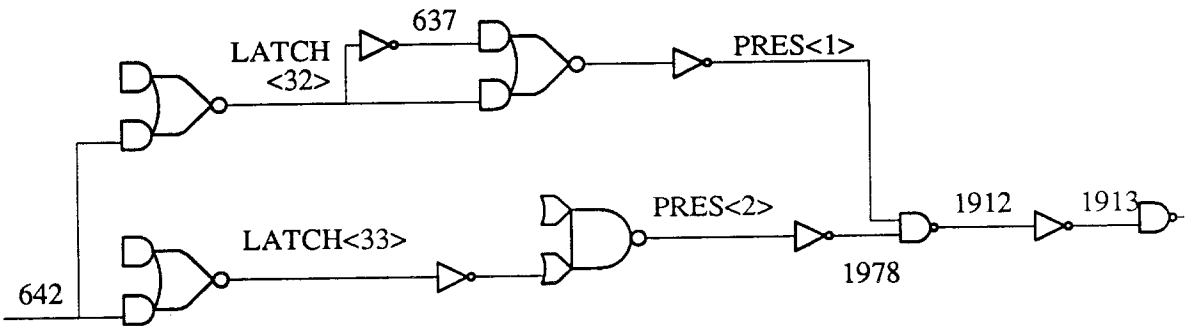


Figure 5-16: Logic for part of two critical paths of the DES chip. The nodes labeled LATCH<32> and LATCH<33> are part of the control logic for those cells (which are latches), not themselves latches.

The asymmetric mean_standard_deviation model has several critical paths that are not critical in any other model. This is principally due to the rising event of the left hand input to node F<3> becoming critical despite a lesser mean than the right hand input because of the larger standard deviation along the left hand paths. Hence it has a larger 3σ point. This in turn is due to a combination of two factors: the phase difference noted above (between the two paths from PRES<1> to F<3>), and the different degrees of simplification performed on the two paths. The left-hand path was able to coalesce more elements into a single arc (from [30]/O2 to F<3>) than the right. Because the nominal and worst case delays were

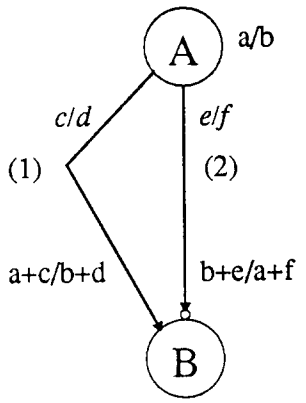


Figure 5-17: Out-of-phase paths in the asymmetric rise-fall model. Even if the ordering of all delays and events is known (a>b, c<e, d<f), both paths will be critical if a-b > e-c (i.e., a+c > b+e). The right hand path is always critical because a+f > b+d.

combined by simple addition, the effect was to increase the correlation between the delays along the left hand path vs. the right (delays that are combined together before coercion to mean_standard_deviation have an effective correlation of 1.0, where all other delays have a correlation of 0.0). This increases the standard deviation of the results. Both factors are necessary in this case: the other asymmetric models lack this different correlation (thus lengthening the right hand path for these models), while the non-asymmetric mean_standard_deviation model lacks the effect of the two phases. Although the difference in the correlations is an artifact of the non-transparent nature of graph simplification when coercing to the mean_standard_deviation model, this example demonstrates how the left hand path could become critical for this model where it would not be for a single_number model looking only at the mean values.

The two inputs to node [30]/I are another example of two out-of-phase paths both being critical: in this case, nodes PRES<5> and PRES<1> have the same event times and therefore act as if they were the same node. The path through node 1987 is thus the longer path that is out of phase with the shorter path directly from PRES<5> to [30]/I.

1.3. Evaluation of the Different Models

For identifying critical paths, the sharpest distinction between the different models appears between the asymmetric rise-fall models and the base versions of the same models. The critical paths found with the asymmetric version of min_max are more like those found with the asymmetric version of either of the other two models than the paths found with min_max alone. Asymmetric models find more critical paths because they have two independent components that can each be critical, while each of the three base models studied here in effect is only critical in a single component for long path searches. This is a mixed blessing. Because only one slack is kept for the whole event time, having separate critical components can lead to non-critical paths being falsely identified as critical, as shown in Chapter 4. But it can also catch true critical paths that would be missed by the simpler models. For reconvergent out-of-phase paths such as shown in Figure 5-17, either path really could be part of a true critical path; it depends on whether the rising or falling time at node B proves to be the more critical with respect to the ultimate constraints. All the simple models would only identify path (2) as critical, however. Therefore if the rising time at B ultimately proves the more crucial of the two components, the true critical path from A to B could be (1), not (2). As discussed in Chapter 4, having separate critical paths for each separately translatable component would preserve this advantage for the asymmetric models, while eliminating most of the falsely identified critical paths. In their present form, the asymmetric models offer a tradeoff: a greater chance of identifying the true critical path, at the risk of swamping the designer with too much extraneous information. For this example, this advantage of the asymmetric models was minimal because most of the reconvergent out-of-phase paths only differed slightly from each other, (most were alternate inputs to and-or-invert gates) but such paths could be much longer and more important in another design.

For predicting cycle times, however, the differences between the base models become much more pronounced, with each base model and its asymmetric version yielding similar predictions. The specific estimates produced in this exercise are highly dependent on the quality of the input data, which is suspect. Unless the input data can be validated, comparing these numbers with actual performance data

on the chip will say more about which model was the luckiest rather than which model was the most accurate. The results do show that the choice of model makes a big difference in the projected cycle time, at least when the individual block delays are as loosely characterized as they are here. Performance estimates varied by more than 2:1 between the single_number model and the min_max model. This is mostly due to the difference between using nominal delays and using worst-case delays; a difference between the coercion strategies rather than between the models themselves. If the single_number model used worst-case times, the only difference between the models would be that the min_max model would compare the worst-case data time to the best-case clock arrival time, where the single_number model would use the worst-case times for both. Thus the projected cycle time with min_max model would be longer by the difference between the best and worst case clock arrival times at the final latch. The single_number model with worst-case times gives the cycle time if all delays are perfectly correlated and at their maximum values; the min_max model gives the worst-case possible time regardless of the correlation between delays. Thus the min_max model estimate is more robust, and clearly safer - but it gives no indication of how likely this worst-case time really is.

The difference between the min_max and mean_standard_deviation results, however, does reflect the differences between the models. The coercions chosen ensure that the positive 3σ point for each delay is equal to the max time for that delay. Thus the difference between the max time and the 3σ point for event times is strictly due to the difference between the models. The difference is significant: compare the arrival time at node F<3> in Figure 5-14 vs. that in Figure 5-15. The 3σ point for the mean_standard_deviation model is 315ns, the max time is 494.2ns: 56% greater. The mean_standard_deviation number is the statistical worst-case time provided all delays are independent; the min_max number is the worst-case time regardless of the correlation between delays. In effect, the min_max number assumes that all delays are perfectly correlated, as this is the worst-case assumption.

Which of these two assumptions better reflects the underlying reality? If the design were composed of discrete components chosen at random from large batches, the assumption of independence would be likely to hold, so the mean_standard_deviation model would be best. In this case, all the

components are located on a single chip. The correlation between delays on the same chip depends on both global factors common to the whole chip, such as how long the chip spent in a particular process step, and local fluctuations in parameters within the chip. If the global factors dominate, the delays will tend to be correlated; if local factors dominate, delays will tend to be uncorrelated over the whole chip, with components placed close together more highly correlated than those far apart. For most manufacturing processes today, the global factors will tend to dominate, producing highly correlated delays. Thus for a single chip, the min_max model is probably the best of the three basic models presented here.

Clearly a statistical model with adjustable correlations between delays, such as the correlation_classes model described in Chapter 3, could be more accurate than either the min_max or the simple mean_standard_deviation models, provided the correlations were properly chosen. Such a model would allow both global and local variations to be accounted for, as well as the distinction between components on a single chip and components split between different chips.

Finally, just as the consensus of a panel of experts is often more accurate than the opinion of any one expert, there may be a real value to a designer in being able to run the same problem using several different models to get a consensus opinion on critical paths and performance bounds. By using models that reflect different assumptions about the sources of delay, a designer can see how sensitive the reported results are to these different assumptions. No timing verifier limited to a single model, no matter how accurate, can provide the kind of perspective that an abstract timing verifier like ATV can by changing only the timing model and leaving everything else fixed.

In summary, the single_number model is mostly useful when simplicity is at a premium, such as estimating block delays early in the design cycle. The mean_standard_deviation model is best when variations in individual delays are mostly uncorrelated, such as designs consisting of discrete components, and when the primary concern is how likely a particular system is to fail. It provides tighter bounds on path lengths than the min_max model. The min_max model is best when variations in individual delays are highly correlated, such as single-chip designs, and when absolutely safe bounds on design performance are required. A model with adjustable correlations, such as the correlation class model from

Chapter 3, is likely to be more accurate than any of the models studied. Using an asymmetric delay version of any of the above base models has a relatively small effect on system performance estimates in most cases, but increases the number of critical paths reported. An asymmetric delay model may catch a true critical path that the base model alone would have missed. By using ATV to rerun an analysis with different timing models, a user can see how sensitive the reported results are to the assumptions made by the different models.

2. PERFORMANCE OF ATV

Run-time results for the analysis of this chip are shown in Table 5-1. The chip was analyzed over two full machine cycles, corresponding to four separate high phases of the main system clock. The four runs described are:

1) An analysis of the unsimplified graph, using the asymmetric single_number model (AS-SN),

2) an analysis of the simplified graph, also using the asymmetric single_number model (AS-SN),

3) an analysis of the simplified graph, using the asymmetric min_max model (AS-MM),

4) an analysis of the simplified graph, using the asymmetric simple mean_standard_deviation model (AS-MS).

All times are in CPU seconds on a VAX 8800. Time spent in garbage collection is not counted, because this varies with available memory, and varied sufficiently from run to run to significantly reduce the repeatability of the results. All VAX times had to be measured while the machine was in multi-user mode; although efforts were made to take the measurements when the machine was not performing other work, this was not always possible (especially for the 8/27/87 PCL measurements reported below). The effect of additional activity on the measured CPU times appeared to increase the measured times somewhat. The Common Lisp version used was VAXLISP version V2.0; the PCL version was the version of March 3, 1987. Graph read-in time and graph simplification time are shown separately, since these only need to be done once, no matter how many analyses are being run.

Table 5-2 shows the performance improvement from going to the 8/27/87 version of PCL. These results also used the VAXLISP version U2.2. The same four runs were made as in the previous case and, in addition, an analysis of the unsimplified graph using the asymmetric mean & standard deviation model (AS-MS) was performed.

Table 5-1: DES Chip Analysis Run Times - PCL 3/21/87 (CPU secs.)				
Phase	Unsimplified	Simplified		
	AS-SN	AS-SN	AS-MM	AS-MS
Graph Read-In	450	450	450	450
Graph Simplification	-	370	370	370
Event Computation	1650	1070	1200	1200
Constraint Analysis	150	160	150	140
Total Time	2250	2050	2170	2160
Analysis Only	1800	1230	1350	1340

Table 5-2: DES Chip Analysis Run Times - PCL 8/27/87 (CPU secs.)					
Phase	Unsimplified		Simplified		
	AS-SN	AS-MS	AS-SN	AS-MM	AS-MS
Graph Read-In	380	380	380	380	380
Graph Simplification	-	-	200	200	200
Event Computation	1050	1250	720	880	750
Constraint Analysis	70	90	80	80	70
Total Time	1500	1720	1380	1540	1400
Analysis Only	1120	1340	800	960	820

Static storage requirements for the data structures, in Megabytes, were:

2.1 (ATV) + 2.9 (Simplified Graph) + 6.5 (Events) = 11.5 (total).

To reduce garbage collection down to a reasonable level, 36 Megabytes of dynamic virtual storage were used. The Lisp system divides this into two 18 Megabyte dynamic spaces, only one of which is used at a time (when one space fills up, garbage collection copies all the live objects from that space to the other).

2.1. Analysis of Delay Times

About 40% of the time labeled Event Computation is used to perform delay operations. In analyzing the simplified DES graph, 25505 top-level delay operations were performed (delaying an asymmetric event-time by an asymmetric delay). Table 5-3 shows the amount of time taken by 2000 delay operations, in CPU seconds, for the 3/21/87 version of PCL.

Table 5-3: Components of Delay Operation - Delaying an Asymmetric (Min, Max) Event-Time by an Asymmetric (Nominal, Worst) Delay (3/21/87 PCL)	Time (sec)	%
Delay Computation	6	16
Coercions	11	30
Flexibility	7	19
Allocating Results	13	35
Total	37	100

Coercions represents the time to perform the delay coercion to the appropriate types. Flexibility indicates how much faster the delay operation could run if the type of the timing model was hardcoded and all compiler optimizations were turned on. Allocating results is the difference between results being allocated dynamically (as they are now), and being put into a pre-allocated structure. Delay computation covers everything else: it is the run time when the delays already come in the appropriate form, the delay model is hardcoded, and the result is preallocated.

Possible speedups to the program come in two forms: speedups to the underlying system (a faster version of Common Lisp; a faster implementation of the Common Lisp Object System standard than PCL can provide) and speedups to ATV itself. For the delay operation, the latter must focus almost entirely on the coercion and result allocation times above, and on ways of reducing the total number of delay operations done by the program. Little can be done about the delay computation or flexibility times (except for a few minor compiler optimizations turned on for the flexibility measurements), except by speeding up the underlying system. Both the coercion and result allocation times can be reduced, at the expense of additional storage space and additional complexity in the program. Pre-allocating space for results could save significant time if this space can be effectively managed by the program. Coercions offer an additional speedup: at present, each delay is coerced to the appropriate form every time it is needed. The average delay in one analysis of the DES chip was coerced 4.6 times, producing the same result each time. A significant speedup would be possible by caching the result of previous coercions on each delay, so that a given coercion would only be done once per delay per analysis.

The most significant potential speedups to ATV would require reducing the number of delay operations done. The user can reduce the time required by reducing the extent of the analysis: on the DES chip

it would have been possible to identify the actual critical path by looking only at one machine cycle or less, a time savings of around 50-75%. Shortening the analysis does decrease the potential speedup from caching coercions, however. The basic clock-phase length analysis algorithm generates many intermediate results just in case they should be required. Many of these results are not used in the end. Switching to some form of lazy evaluation might reduce this overhead considerably.

ATV was primarily designed for flexibility rather than performance, and the above numbers reflect this. Based on the numbers in Table 5-3, the two improvements proposed (saving the results of delay coercions, special storage management for event times) might reduce the time per delay by up to 50%, yielding an overall improvement of up to 20% of the event computation time. If other operations exhibit similar speedups, the speed of the program could perhaps be doubled. Most of the overhead associated with ATV, however, comes directly from the underlying PCL system.

This overhead was studied by extending the delay benchmark reported in Table 5-3. A simplified version of the benchmark, without the delay coercions or flexibility offered by the full benchmark, was implemented in PCL, in Common Lisp without PCL, and in C (for static results only). The results are reported below, for both the VAX 8800 running VAXLISP and a Sun 3/160 running ExCL, running compiled versions of the benchmark. The Sun measurements were taken in single-user mode. Times are reported in both absolute (CPU times) and relative terms (relative to the C version with static results), for 2000 delay operations (average of 5 runs each). The C results were measured by measuring the time for 2,000,000 delay operations and scaling down by 1000.

Dynamic results represents the time when a new result is allocated from the memory system every time through the loop (i.e., the combination of Delay Computation and Allocating Results from Table 5-3). Static results represents the time when the results are put into the same statically allocated storage every time through the loop (just the Delay Computation from Table 5-3). The only *methods* called in this simplified version of the benchmark are instance accessors (functions to access members of the data structures). The measurements labeled PCL w/o accessors show the results when these access methods are replaced with direct function calls (using the appropriate option in the PCL *with-slots* macro). The

overhead associated with accessor methods is much greater for the 3/21/87 version of PCL than for the 8/27/87 version (it almost disappears entirely for the Sun measurements in Table 5-5). The versions of VAXLISP used for the VAX measurements of PCL differ because the 8/27 version of PCL only runs in the new version of VAXLISP. The times reported for the Common Lisp Defstructs version of the benchmark in tables VI and VII should provide a sense of the speedup due to the new version of VAXLISP.

The benchmark was rewritten from the ATV code used in Table 5-3, to make the code for the different Lisp versions of the benchmark as identical as possible. This may account for the difference between the dynamic results for PCL classes in Table 5-6 (28.6 seconds) and the sum of Delay Computation and Allocating Results from Table 5-3 (19 seconds).

Table 5-4: Sun Measurements for simplified delays benchmark
Sun 3/160, PCL 3/21/87, ExCL 1.5.10 4/9/87 Single-user:

Absolute Times (CPU secs):	Static Results	Dynamic Results
PCL Classes:	64.7	199
PCL w/o Accessors:	24.4	169
Common Lisp Defstructs:	1.01	12.0
C	0.594	-
Relative Times (Static C = 1.00)	**Static Results**	**Dynamic Results**
PCL Classes:	109	335
PCL w/o Accessors:	41.1	285
Common Lisp Defstructs:	1.70	20.2
C	1.00	-

Table 5-5: Sun Measurements for simplified delays benchmark
Sun 3/160, PCL 8/27/87, ExCL 1.5.10 4/9/87 Single-user:

Absolute Times (CPU secs):	Static Results	Dynamic Results
PCL Classes:	10.4	77.8
PCL w/o Accessors:	9.82	76.3
Common Lisp Defstructs:	1.01	11.9
C	0.594	-
Relative Times (Static C = 1.00)	**Static Results**	**Dynamic Results**
PCL Classes:	17.5	131
PCL w/o Accessors:	16.5	128
Common Lisp Defstructs:	1.70	20.0
C	1.00	-

Table 5-6: Vax Measurements for simplified delays benchmark Vax 8800, PCL 3/21/87, VAXLISP 2.0 Multi-user:		
Absolute Times (CPU secs):		
	Static Results	Dynamic Results
PCL Classes:	6.35	28.6
PCL w/o Accessors:	0.81	24.0
Common Lisp Defstructs:	0.28	0.94
C	0.034	-
Relative Times (Static C = 1.00)		
	Static Results	Dynamic Results
PCL Classes:	187	841
PCL w/o Accessors:	24	706
Common Lisp Defstructs:	8.2	27.6
C	1.00	-

Table 5-7: Vax Measurements for simplified delays benchmark Vax 8800, PCL 8/27/87, VAXLISP 2.2 Multi-user:		
Absolute Times (CPU secs):		
	Static Results	Dynamic Results
PCL Classes:	3.52	21.5
PCL w/o Accessors:	0.91	19.4
Common Lisp Defstructs:	0.19	0.70
C	0.034	-
Relative Times (Static C = 1.00)		
	Static Results	Dynamic Results
PCL Classes:	104	632
PCL w/o Accessors:	27	571
Common Lisp Defstructs:	5.6	20.6
C	1.00	-

PCL is a rapidly changing system, and these performance numbers have been improving dramatically in new versions: the 8/27/87 version of PCL shows significant improvement over the 3/21/87 version in the above tables, though the performance improved more on the Sun version than on the VAX. As PCL continues to evolve, ATV should benefit from these performance enhancements.

Chapter 6 - Future Work

1. ENHANCEMENTS TO ATV

Now that ATV has been developed, one area of future work involves enhancements to the basic program: either further refining the ATV code itself, or developing interface programs and preprocessors that can be used to make the program more usable for different applications, and at different stages of the system design process.

1.1. Integration with Linear Programming

At present, ATV merely provides the spacing constraints between different reference frames, leaving the interpretation to the user. As noted in Chapter 4, the logical next step would be to solve the constraints together with any external constraints as a linear program to produce the actual clock schedule. This would allow the program to identify those constraints that are truly critical to the cycle time of the design rather than leaving this task for the user. Given this information, the program could compute global slacks for events with respect to the design cycle time, rather than just with respect to a particular successor event. This would make the program easier to use: the ATV analysis algorithm computes many constraints that are either implied by or dominated by other constraints, and it would help to filter these out. In designing the algorithm, I was not concerned with eliminating redundant constraints, anticipating some ultimate filter such as this that would remove those constraints that were unimportant. For example, when there are two consecutive transparent latch arcs in the timing graph that are controlled by different clocks, ATV passes events through both of them even though it is likely that any such events will arrive at the second latch before its clock and hence be dominated by the events originating on the opening edge of that latch. If the ultimate clock schedule ensures that this condition holds, the events coming through the latches will generate constraints that are less restrictive than those generated by the opening of the second latch, and hence unimportant. On the other hand, if the delays and clock distribution skews are such that the second latch can open before its data stabilizes (as in the graph for Jouppi's loop

example), it is essential to pass the events through the second latch. I felt it was more important to make sure that no constraint that possibly could matter was omitted, in order to assure that no true constraint was overlooked.

1.2. Coercion Management

Another enhancement that would make the program easier to use is more automatic handling of coercions between delay representations (and ultimately, between event times). At present, the user can control which coercions are allowed (by explicitly loading the allowable coercion methods), but has no real control over how the allowable coercions are applied. The choice of what coercion method is invoked is determined by the position of the source delay class in the class hierarchy (see Figure 6-1).

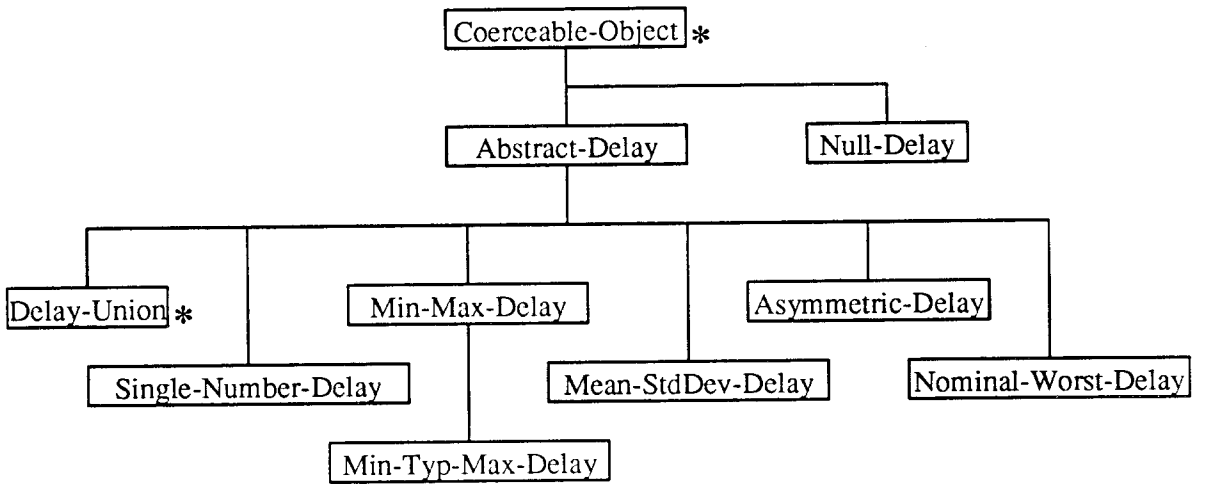


Figure 6-1: Class Hierarchy for Delay Classes. Coercion methods are initially specified only for the coerceable-object and delay-union classes; other coercions must be explicitly loaded by the user. When a coercion is attempted on a delay, PCL looks up the class hierarchy from the class of the delay, looking for the first class that has a coercion method defined. A method can either succeed (by returning a delay of the desired class), fail (by returning a failure code), or invoke the next higher-level method. The default coercion method supplied by coerceable-object fails unless the source delay is already a member of the desired class. Thus when a min_typ_max delay is coerced, PCL looks first for a coercion method defined for min-typ-max-delay, then for min-max-delay, then for abstract-delay, and finally will invoke the coerceable-object method (which fails unless the target is one of the above classes) if no more specific method was found.

The greatest potential for coercion strategies occur when coercing from a *delay-union*,* where several delay formats may be present, offering a potential choice of coercions. The default coercion method for delay-unions offers no real scope for exploring alternative coercion paths: the method first checks to see if the target delay format is a member of the delay-union, and if not, tries to coerce each delay format in order, returning the result of the first coercion to succeed. For example, suppose that we are attempting to coerce to a mean_standard_deviation delay from a delay-union containing (in order) a single_number delay, a min_typ_max delay, and a nominal_worst delay (Figure 6-2). Suppose further that coercion methods have been defined from the min_max delay format and the nominal_worst delay format to the mean_standard_deviation delay format. The coercion method for delay-unions will first check to see if there is a mean_standard_deviation delay in the union (there isn't). It next attempts to coerce the single_number delay to a mean_standard_deviation delay, which fails (the only applicable method is the coerceable-object method). It then attempts to coerce the min_typ_max delay to a mean_standard_deviation delay. The coercion method for min_max delays is invoked on the min_typ_max delay and succeeds, returning a mean_standard_deviation delay that is then returned as the result of coercing the delay-union. The alternate coercion from the nominal_worst delay format to the mean_standard_deviation delay format is not considered, because the earlier coercion has already succeeded.

This rule for coercing delay-unions makes the order in which a delay-union is created significant, which is undesirable. Only one level of coercion is performed: if there is no direct coercion from the source format (or one of its superclasses) to the target, the coercion fails. Thus in the above example, if there was a coercion method defined from single_number delays to min_max delays, the single_number delay would *not* be coerced to a mean_standard_deviation delay via an intermediate min_max delay.

It would be useful to give the user more control over these coercions, by keeping a graph of the available delay classes and the permissible coercions between them, and allowing the coercions to be

*As described in chapter 4, a delay-union is a special class of delay that consists of a list of one or more other delays. This allows multiple delay formats to be specified for the same arc.

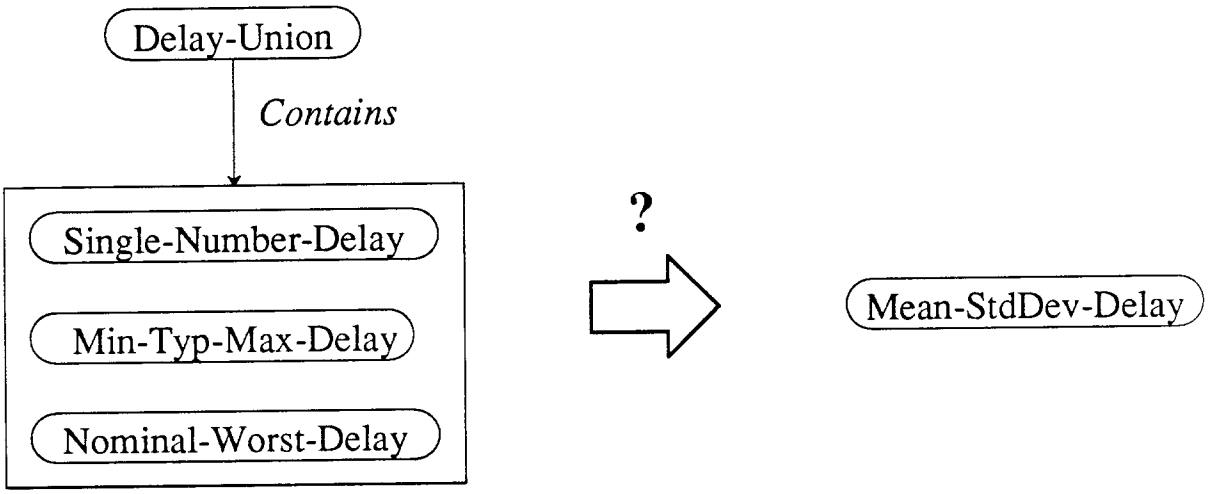


Figure 6-2: Coercion Example. The program attempts to coerce a delay-union into a mean_standard_deviation delay.

graded on how safe or desirable they are. Then the coercion methods could develop coercion strategies based on this graph, the formats available, and user input about allowable coercions. For example, if we are trying to coerce from a min_max format to a mean_standard_deviation format, it is more desirable to go directly from one to the other rather than to first coerce from min_max to a single_number (such as by taking the average of min and max) and then from the single_number to mean & standard deviation (such as by taking some percent of the number as the standard deviation), even if both the latter coercions are supplied. These two coercions are less safe than a direct coercion, as information is destroyed in coercing to the single_number, and hence the coercion is not invertible. If a the direct coercion was not supplied in this case, the user might want to have control over whether or not an indirect path such as this double coercion was allowed. If each coercion was graded on a scale from 0 to 5, with 0 representing a completely safe coercion (min_typ_max to min_max delay union to any of its components) and 5 representing a totally unsafe coercion (no information preserved; the program makes up data based on some default - used only for testing the program or for pushing an example through in the presence of missing delay data), the user could specify the maximum grade of coercions permitted, or require the program to ask before attempting a potentially unsafe coercion. Rules could be generated that would rate the

safety of a series of coercions based on the safety of the component coercions.

1.3. Depth-first Loop Analysis

I originally intended to include an optional depth-first algorithm in ATV to aid in detailed path analysis for any users that wanted such a capability and were willing to pay the price in execution time. Although most applications are handled by the existing ATV search algorithm, full depth-first search is uniquely capable of analyzing complex loop structures, as shown in Chapter 2. Unclocked loops generally represent storage elements and should be encapsulated by a preprocessor into gated arcs before entering ATV. Loops that are purely clocked by transparent latches can be better analyzed by a depth-first search. Consider the example of Figure 6-3, where the delay around the loop is 18ns and the desired cycle length is 16ns. The ATV search algorithm will detect the timing error provided that the analysis is carried forward for a sufficient number of machine cycles.

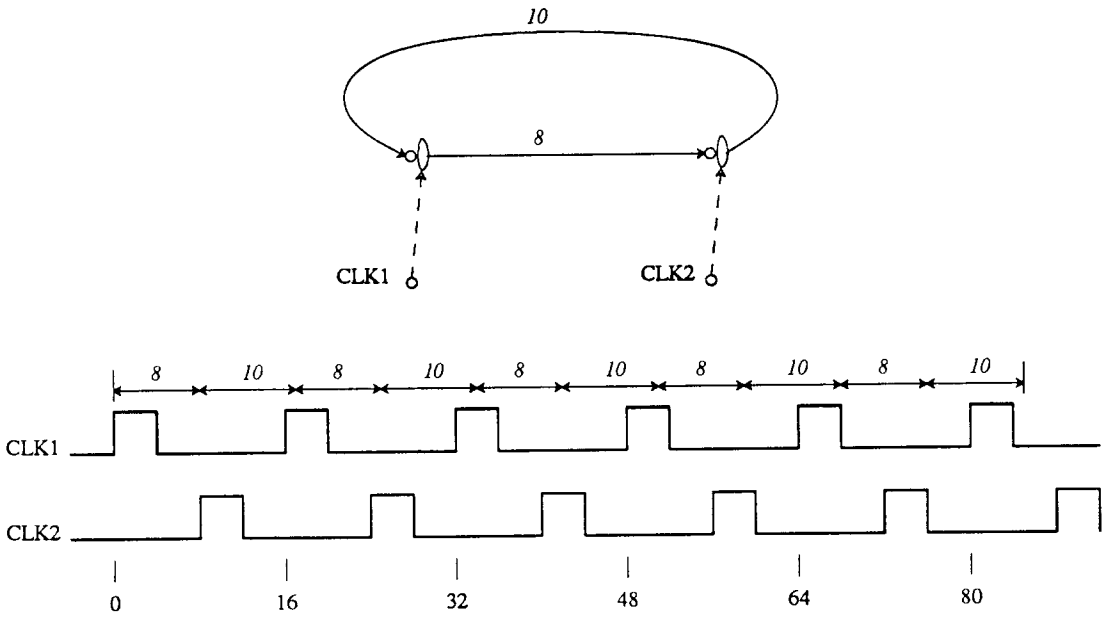


Figure 6-3: Transparent latch loop structure. It requires analysis for multiple cycles for the ATV algorithm to detect the excessive loop length, as shown here. Depth-first search or a special loop handling algorithm could address this special case.

As shown in the figure, six machine cycles are necessary to detect this error, as it is not until the fifth cycle that the arrival time misses the closing edge of the clock. The number of cycles required is inversely proportional to the size of the error. It is unreasonable to expect the user to always anticipate this problem and increase the length of the analysis appropriately. The problem is that although in general, inputs to a transparent latch must stabilize before the closing edge of the latch, in the case of a loop they must stabilize before the opening edge of the latch that initiates the loop. Depth-first search could recognize this special case, and handle it appropriately. Alternately, a special loop analysis algorithm could be added to do special checks for this case.

1.4. New Model Development

ATV provides a framework in which new models can be developed and experimented with. In addition to the models mentioned in Chapter 3, here are some ideas for other unique models to meet special needs.

1.4.1. Confidence Levels

As timing verification is used in a developing design, the delay estimates within that design will have varying degrees of quality. Some blocks may have been exhaustively analyzed by circuit simulation of the actual layout, while other delays may simply represent designer estimates for blocks yet to be designed. In reporting results, it would be helpful to have some idea of how reliable the underlying numbers are. The confidence levels model would be a higher-level model, much like the asymmetric rise-fall model, that could use any other model as its components and would attach *confidence levels* to the values in a delay format. It would then have rules for combining the confidence levels from different delays to provide an overall index of confidence in the reported results.

1.4.2. Complex Transistor Structures

Nothing in ATV requires delays to be limited to a few numeric parameters: they can in principle be arbitrary data structures, or even executable code.* In particular, a delay could be a network of transistors, such as a stage in Crystal, and the delay operation could invoke an analysis program, such as SPICE, to compute a delay through the structure. This could be used to model low-level analog effects in individual blocks when this is important, while retaining simpler models for those blocks where such effects are unimportant.

1.5. Application Areas

The following four areas all represent potential application areas for ATV that would require the writing of some new interface program to generate the ATV dependency graph. These are system integration tasks that would make ATV a more useful tool by integrating it more closely with existing tools and design methodologies.

1.5.1. Microarchitecture Analysis

ATV was designed to be useable early in the design cycle. In principle, verification with ATV could proceed in parallel with the development of the microarchitecture description. Annotating an ISP description with delay estimates and writing an interface program to generate a dependency graph from this description could allow the timing analysis to begin at a very early stage, and provide quick feedback to the designer about the timing consequences of possible changes to the microarchitecture. Microarchitecture descriptions written in BDSYN/BDNET (the input language to the Berkeley Synthesis System) already have a path to ATV via the interface program from Oct: timing information can be added to the BDNET description of the blocks and hence to the Oct database.† However more designers use ISP, so a

*From a practical standpoint, arbitrary data structures may be more difficult to store in an external database or to represent cleanly in an input format to ATV. These are interfacing issues that can be addressed outside of ATV.

†Or the synthesis procedure can be run on the description, automatically bringing in any timing information attached to the library cells that the design compiles into.

separate path from an ISP description would be useful.

1.5.2. Floorplanning

Another enhancement to make ATV more useful early in the design cycle would be a program to take chip floorplanning information into account when estimating early delay information. This would be integrated with the Berkeley Synthesis System, and would provide timing estimates for blocks based on their size, logic depth,* and number of fan-outs, and estimates for interconnect based on estimated vertical and horizontal length. To accommodate a wide variety of delay formats, the program would fill in these parameters into user-specified templates that would specify the expressions for the delay formats chosen by the user.

For example, one user might define a template for single_number delay such as:

```
(sn-del (+ 3.0 (* 4.0 %fan-out) (* 2.6 %wire-length)))
```

(i.e., $d = 3.0 + 4.0f + 2.6w$), where the % sign identifies a parameter to be supplied by the program. Another user might specify a template for min_max delay:

```
(mm-del (+ 2.0 (* 2.0 %fan-out) (* 1.0 %wire-length))
        (+ 4.0 (* 7.0 %fan-out) (* 3.5 %wire-length)))
```

(i.e., $\min = 2.0 + 2.0f + 1.0w$, $\max = 4.0 + 7.0f + 3.5w$).

1.5.3. MOS Preprocessor

Although the current version of ATV is well-suited to analyzing designs consisting of components chosen from well-characterized libraries, components produced by module generators or custom hand-design are not so well characterized. Designs in Oct with a well-defined block structure can have timing descriptions added to the generated blocks. Pranav Ashar is working on a program to characterize

*The number of logic levels would be specified by the user.

complex combinational logic gates by analyzing their transistor structure, simulating them for various input waveforms, analyzing the data, and then annotating the information back as delay information in the format required by ATV. Many existing designs are only available as a flat transistor netlist. A preprocessor to identify gate structures within such a netlist and characterize them would be useful. MOS gates are sensitive to the effects of drive strength; it is unclear whether the time-slope model from Chapter 3 is sufficient to model all MOS gates or if a new model needs to be developed.

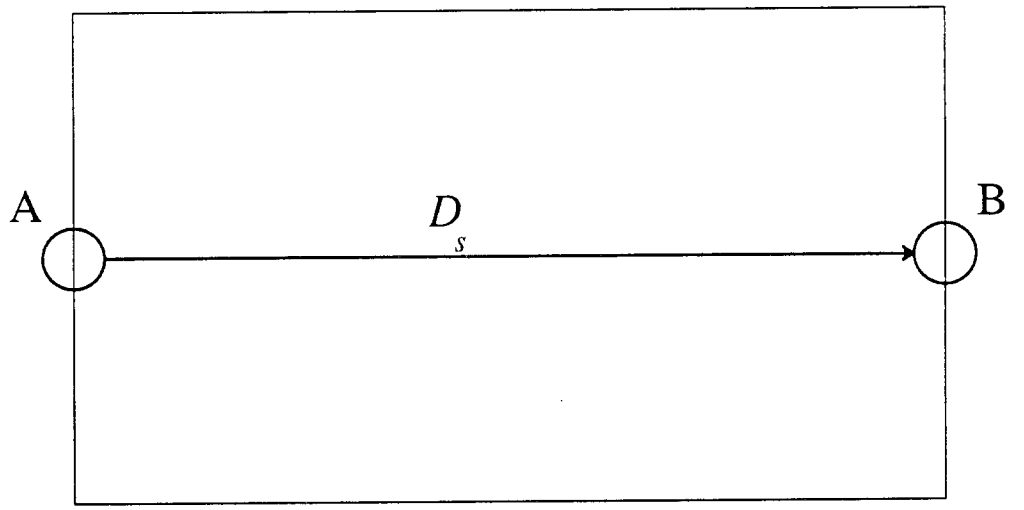
1.5.4. Integrating Analysis of Chips and Boards

An integrated environment for design capture and simulation is required to analyze the interactions of chip circuitry with the surrounding circuitry on a board. I expect that Oct can be used to represent designs that include off-chip circuitry, and hence that the existing interface from Oct to ATV can be used for any such designs that are entered in Oct. Such an extension to the board level would probably require new coercions between timing models at chip boundaries.

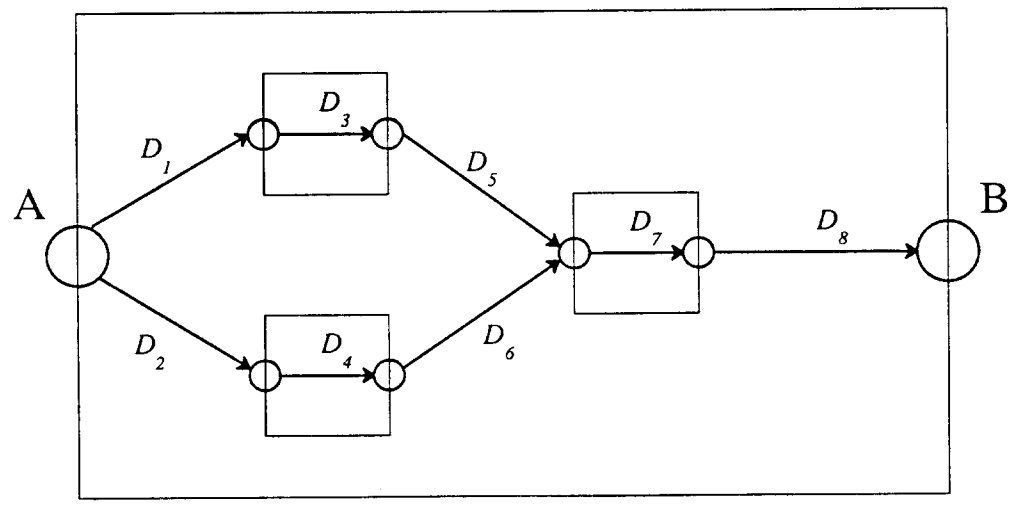
2. HIERARCHICAL TIMING VERIFICATION

As a design evolves, the most appropriate level of detail to describe it may change. Working top-down, the designers may start with very rough descriptions of blocks and successively refine them into more detailed descriptions. Working bottom-up, once a block has been designed out of lower-level components, the designers may wish to use a simpler description of the block when working at the next higher level, creating an abstraction that retains only those features that are important to describe how the block interacts with its peers. The Oct database supports these different levels of description by defining two *facets* (complementary descriptions) that may exist for any block in a design: the *contents facet*, which describes the block's implementation, and the *interface facet*, which is the description the block presents to the surrounding design. Figure 6-4 shows how the timing description of a block might appear at these two levels: Figure 6-4a shows a simple timing description consisting of a single delay arc between the ports of the block (the interface); Figure 6-4b shows the more detailed description involving

several lower-level blocks (the contents). Either level could come first: working top-down, designers would work from the interface to design the contents; working bottom-up, they would try to derive the interface from the contents. The interface program from Oct to ATV accepts timing descriptions on either the interface facet or contents facet of a block (or even on an individual instance of the block) in composing the timing graph for ATV.



a)



b)

Figure 6-4: Hierarchical timing verification. (a): High-level description (b): More detailed description

Previous work on hierarchical timing verification has generally been limited to the automatic derivation of simpler higher-level descriptions in the bottom-up style [NST82, ReC86, TON83]. ATV could use the graph simplification algorithm described in Chapter 4 to derive simpler descriptions of combinational logic modules that could then be written back to the database as an interface description for the block. As noted in the discussion of simplification, graph simplification uses an implied timing model for each delay type, and results may change if the simplified graph is used with a timing model other than the implied one. This is clearly more of an issue for an abstract timing verifier than for verifiers that use a single built-in timing model.

The previous systems mentioned above have depended on eliminating paths between edge-triggered registers to achieve significant simplification. With transparent latches, less simplification is available. This is why Jouppi [Jou84] chose not to include hierarchical modeling as part of TV: he speculated that the cost to do so would be greater than that of verifying a flat design, and that the available simplification would be minimal. ATV could simplify out paths between edge-triggered arcs when such arcs exist. More work is needed to determine bounds on the errors introduced for simplification with different timing models, and on discovering appropriate approximations.

New features would be needed to support top-down stepwise refinement of designs. It would be useful to be able to automatically check whether a timing description entered at a high level matched the detailed implementation. This is difficult for general delay models and may require the development of new operations in the abstract model. It is possible to pass constraints entered at a higher level down so that they can be checked in a flat verification of the design at a lower level, but this requires flattening and analyzing the whole design at each level of interest. Another issue is automatically distributing a higher-level delay among its components at a lower level. Again, new features are likely to be needed to support such automatic operations for general timing models.

3. BEYOND TIMING VERIFICATION: ASYNCHRONOUS AND SELF-TIMED SYSTEMS

Asynchronous and self-timed systems pose new challenges for programs that are to check their timing behavior. Although portions of these systems may be analyzed by traditional timing verification algorithms, these algorithms do not really address the true timing needs of these designs. One of the major timing issues with asynchronous systems is verifying that asynchronous signals are properly synchronized before interacting with each other. The synchronizers themselves have a non-zero probability of entering a metastable state when asynchronous inputs arrive too close to each other; the design of synchronizers to minimize this probability is a delicate issue of circuit design [KlC87] that has little to do with the value-independent analysis characteristic of timing verifiers. The problems of verifying the correctness of an asynchronous design have more to do with verifying that the synchronizers are properly designed and are inserted between any two asynchronous components. However, if bounds can be calculated on the response of an asynchronous component, such bounds can be translated into a delay for a timing verifier estimating the performance of the surrounding system.

Likewise for the special case of asynchronous systems known as self-timed systems, traditional timing verification methods have limited utility. The problem may be seen by considering the typical self-timed functional unit seen in Figure 6-5. This unit reads data from its predecessor when the Input Ready signal is raised, computes for some indefinite time, and then raises the Output Ready signal when it is done, so that the next self-timed unit may process its output. Acknowledgement signals are raised when the data is accepted by a successor unit. The duration of the computation may be data-dependent. For example, this unit might be a multiplier that performs bit-serial multiplication taking 1 to 32 internal cycles to complete its work.

We could analyze this unit with a traditional timing verifier, such as ATV, to see how long the path delays are inside this block. The results of the analysis, though, will only tend to indicate how long the single-cycle path is through the unit, and that only if the unit does not contain any self-timed subblocks. Value-independent timing verification will not tell us much except that the unit could theoretically produce an output result every internal cycle, though this is in fact rare. The problem is that when analyzing

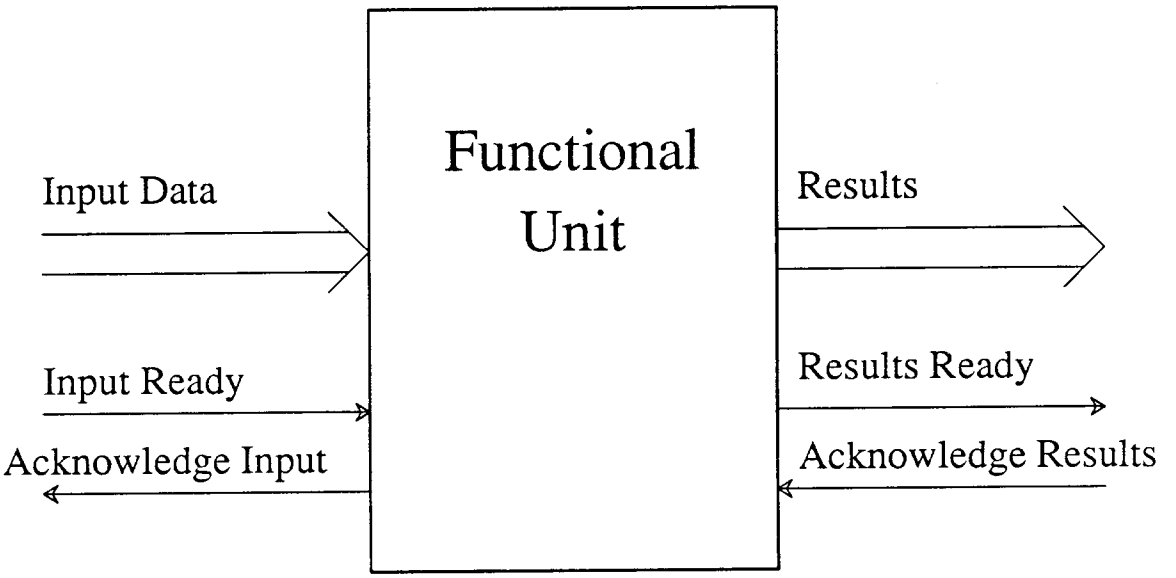


Figure 6-5: Self-timed functional block.

the next-level design up from this unit, what is important is how long the unit may take to completely process one set of data. What we want to know is *how many* iterations could take place; what timing verification tells us is *how long* the path for each iteration might be. There does not appear to be any way for a timing verifier to deduce that the multiplication will take at most 32 cycles.

Timing simulation would appear to be a more promising approach to the analysis of self-timed systems, perhaps coupled with the abstract timing model of ATV to provide flexibility in representing event times. But simulation results are highly data-dependent. There does not appear to be any good way to assure that the data being simulated not only take the maximum number of cycles, but exercise the worst-case paths through the logic for each cycle. To get worst-case bounds on paths through self-timed systems seems to require an imaginative combination of theorem-proving, simulation, and timing verification. I consider this one of the most important unsolved problems in timing verification today.

Chapter 7 - Conclusions

Different design environments and different phases of the design process demand different models for time and delay. The best model for circuitry on a MOS chip is not necessarily the best model for the surrounding TTL circuitry on a board; the best model for crude delay estimates early in the design process is not necessarily the best model when more detailed information is available. Nor are the priorities of all design teams the same: one team may want an absolute guarantee that a design that passes timing verification will not have any timing errors, a second team may be willing to accept a statistical statement that errors are unlikely in order to get a more aggressive design, and a third team may put their faith in the measurement of real parts and just want a rough statement of what cycle times they can expect under a variety of assumed conditions. ATV, the Abstract Timing Verifier, is a general and uniform framework for timing verification that can accept a wide variety of different models for time and delay. As long as a model supplies the operations defined in the Abstract Timing Model, it can be plugged in to ATV and used for timing verification. The four fundamental operations, *translating, delaying, merging* and *comparing* event times, together with the secondary operations *merge_with_slacks, satisfy_constraint* and *sum_delays* are readily defined for most models used by other timing verifiers today, and may be defined for many new models as well. Many different models were proposed and studied in Chapter 3.

Because the other operations are invariant with respect to translation, we can group events into *reference frames*, each of which may be rigidly translated with respect to the others. This observation is the key to the development of a new analysis algorithm that can either verify a specified clock schedule or derive spacing constraints between clock edges in a flexible clock schedule. This algorithm operates on critical paths that may extend through transparent latches over multiple machine cycles. It does not require either clock phases or the spacings between clock phases to have uniform lengths within a machine cycle. It is a breadth-first algorithm whose running time is linear in the size of the design. In subsequent analysis, the program can enumerate all critical paths generating a given event in order of increasing slack.

A coercion mechanism for delay formats allows delay information to be entered in its natural form and coerced into whatever form is required for a particular analysis. By making coercions explicit rather than hidden in the data entry process, they become accessible to the user of the program, who may change coercions that do not agree with his or her assumptions. The same basic mechanism may be extended to create delays with special properties, including the potential of coercing between timing models at specified module boundaries.

The Abstract Timing Model, by abstracting out the common properties of many different timing models, provides a new conceptual framework for thinking about timing issues that distinguishes between those concepts that generalize to many different timing models and those that are specific to a single model. It underlines the distinction between *event times* and *delays*, a distinction that has not always been observed by those working with a model where the two have the same format. The challenge of defining critical inputs in a general way shows how many intuitive definitions are based on assumptions that do not generalize to all models.

ATV was designed to be a flexible and wide-ranging program that could accept input from a variety of sources, ranging from microarchitectural descriptions to VLSI chips to discrete logic components, and provide a common framework for verifying designs spanning all these sources. It provides a powerful and general framework for timing verification at all stages of the design cycle that makes it easy to develop new models as needed to address special needs.

Bibliography

[Agr82] V. D. Agrawal, "Synchronous Path Analysis in MOS Circuit Simulator," *Proceedings of the 19th IEEE/ACM Design Automation Conference*, Las Vegas, NV, 1982, 629-635.

[BLA82] L. C. Bening, T. A. Lane, C. R. Alexander and J. E. Smith, "Development in Logic Network Path Delay Analysis," *Proceedings of the 19th IEEE/ACM Design Automation Conference*, Las Vegas, 1982, 605-615.

[BMC87] J. Benkoski, E. V. Meersch, L. Claesen and H. D. Man, "Efficient Algorithms for Solving the False Path Problem in Timing Verification," *Proceedings of the IEEE International Conference on Computer-Aided Design (ICCAD-87)*, Santa Clara, CA, November, 1987, 44-47.

[BKK86] D. G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik and F. Zdybel, "CommonLoops: Merging Lisp and Object-Oriented Programming," *OOPSLA '86*, Portland, Oregon, Sept. 29 - Oct. 2, 1986, 17-29. Special Issue of SIGPLAN Notices Notices, Vol. 21, No. 11, November, 1986.

[BoS77] A. K. Bose and S. A. Szygenda, "Detection of Static and Dynamic Hazards in Logic Nets," *Proceedings of the 14th IEEE/ACM Design Automation Conference*, New Orleans, 1977, 220-224.

[BCS84] D. J. Boyce, E. R. Cohn, J. H. Shelly and D. R. Tryon, "Statistical Technique for Array Delay Calculation," *IBM Technical Disclosure Bulletin 26* (8):4168-4171, Jan 1984.

[CaL82] D. Caldwell and E. D. Law, "Verifier Homes in on Timing Errors in Digital Designs," *Electronics 55* (19):163-6, Sept. 22, 1982.

[DaR86] M. R. Dagenais and N. C. Rumin, "Circuit-Level Timing Analysis and Design Verification of High-Performance MOS Computer Circuits," *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, Port Chester, New York, October, 1986, 356-359.

[EiW77] E. B. Eichelberger and T. W. Williams, "A Logic Design Structure for LSI Testability," *Proceedings of the 14th Design Automation Conference*, New Orleans, 1977, 462-468.

[FVP82] J. K. Foderaro, K. S. Van Dyke and D. A. Patterson, "Running RISCs," *VLSI Design III* (5):27-32, September/October 1982.

[FrR80] A. L. Frisiani and J. P. Roth, "System for Timing Verification," RC 8373 (#36454), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, July 23, 1980. A copy is available at the Stanford University Mathematical Sciences Library, cataloged as Technical Report #014556.

[FrR81] A. L. Frisiani and J. P. Roth, "Computer Timing Verifier," *IBM Technical Disclosure Bulletin 23* (11):5235-6, April 1981.

[FrR83] A. L. Frisiani and J. P. Roth, "Computer Timing Verification," *IBM Tech. Disclosure Bulletin 26* (3A):969-970, Aug. 1983.

[GaJ79] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, San Francisco, 1979.

[Gho87] S. Ghosh, "A Distributed Approach to Timing Verification of Synchronous and Asynchronous Digital Designs," *IEEE Transactions on CAD CAD-6* (4):666-677, July, 1987.

[GSS86] M. Glesner, J. Schuck and R. B. Steck, "SCAT - A New Statistical Timing Verifier in a Silicon Compiler System," *Proceedings of the 23rd ACM/IEEE Design Automation Conference*, Las Vegas, NV, June, 1986, 220-226.

[HaO71] R. A. Harrison and D. J. Olson, "Race Analysis of Digital Systems Without Logic Simulation," *Proceedings of the 1971 IEEE/ACM Design Automation Workshop*, Atlantic City, NJ, 1971, 82-94.

[HMS86] D. S. Harrison, P. Moore, R. L. Spickelmier and A. R. Newton, "Data Management and Graphics Editing in the Berkeley Design Environment," *ICCAD-86*, Santa Clara, CA,

November 11-13, 1986, 24-27.

[HSC82] R. B. Hitchcock, G. L. Smith and D. D. Cheng, "Timing Analysis of Computer Hardware," *IBM Journal of Research and Development 26* (1):100-105, 1982.

[Hit82] R. B. Hitchcock, "Timing Verification and the Timing Analysis Program," *Proceedings of the 19th Design Automation Conference*, Las Vegas, 1982, 594-603.

[HoS78] E. Horowitz and S. Sahni, *Fundamentals of Computer Algorithms*, Computer Science Press, Inc., University of Southern California, 1978.

[HKN86] S. H. Hwang, Y. H. Kim and A. R. Newton, "An Accurate Delay Modeling Technique for Switch-Level Timing Verification," *Proceedings of the 23rd IEEE/ACM Design Automation Conference*, Las Vegas, NV, June, 1986, 227-233.

[Jou83a] N. P. Jouppi, "Timing Analysis for nMOS VLSI," *Proceedings of the 20th IEEE/ACM Design Automation Conference*, Miami Beach, 1983, 411-418.

[Jou83b] N. P. Jouppi, "TV: An nMOS timing analyzer," *Proceedings of the 3rd Caltech Confence on VLSI*, March 1983, 71-86.

[Jou84] N. P. Jouppi, *Timing Verification and Performance Improvement of MOS VLSI Designs*, PhD. Thesis, Department of Electrical Engineering, Stanford University, Palo Alto, CA, June 1984.

[Jou87] N. P. Jouppi, "Timing Analysis and Performance Improvement of MOS VLSI Designs," *IEEE Transactions on CAD CAD-6* (4):650-665, July, 1987.

[KYC81] R. Kamikawai, M. Yamada, T. Chiba, K. Furumaya and Y. Tsuchiya, "A Critical Path Delay Check System," *Proceedings of the 18th Design Automation Conference*, Nashville, 1981, 118-123.

[KKS84] Y. H. Kim, J. E. Kleckner, R. A. Saleh and A. R. Newton, "Electrical-Logic Simulation," *ICCAD-84*, Santa Clara, CA, November 12-15, 1984, 7-9.

[KiC66] T. I. Kirkpatrick and N. R. Clark, "PERT as an AID to Logic Design," *IBM Journal of Research and Development 10* (2):135-141, March 1966.

[KlC87] L. Kleeman and A. Cantoni, "Metastable Behavior in Digital Systems," *IEEE Design & Test of Computers 4* (6), December, 1987.

[Law76] E. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holdt, Rinehart and Winston, New York, 1976.

[LiM84] T. Lin and C. A. Mead, "Signal Delay in General RC Networks," *IEEE Transactions on Computer-Aided Design CAD-3* (4):331-349, October, 1984.

[MAC77] *MACSYMA Reference Manual*, The Mathlab Group, Laboratory for Computer Science, MIT, Cambridge, MA, December 1977. Version Nine, Second Printing.

[Mag77] B. Magnhagen, "Practical Experiences from Signal Probability Simulation of Digital Designs," *Proceedings of the 14th IEEE/ACM Design Automation Conference*, New Orleans, 1977, 216-219.

[McW78a] T. M. McWilliams and L. C. Widdoes, Jr., "The SCALD Physical Design Subsystem," *Proceedings of the 15th IEEE/ACM Design Automation Conference*, Las Vegas, 1978, 278-284.

[McW78b] T. M. McWilliams and L. C. Widdoes, Jr., "SCALD: Structured Computer-Aided Logic Design," *Proceedings of the 15th IEEE/ACM Design Automation Conference*, Las Vegas, 1978, 271-277.

[McW80a] T. M. McWilliams, "Verification of Timing Constraints on Large Digital Systems," *Proceedings of the 17th Design Automation Conference*, Minneapolis, 1980, 139-147.

[McW80b] T. M. McWilliams, *Verification of Timing Constraints on Large Digital Systems*, PhD. Thesis, Computer Science Department, Stanford University., Palo Alto, CA, May 1980.

[MPV81] M. Mezzalama, P. Prinetto and I. Visintin, "Timing Analysis and Logic Verification via User Selectable Multiple Value Logics," *Circuit Theory and Design. Proceedings of the*

1981 European Conference on Circuit Theory and Design, The Hague, Netherlands, Aug. 1981, 346-51.

[MPD83] J. J. Moder, C. R. Phillips and E. W. Davis, *Project Management with CPM, PERT and Precedence Diagramming (Third Edition)*, Van Nostrand Reinhold Company, New York, 1983.

[MIK85] M. Muraoka, H. Iida, H. Kikuchihara, M. Murakami and K. Hirakawa, "ACTAS: An Accurate Timing Analysis System for VLSI," *Proceedings of the 22nd IEEE/ACM Design Automation Conference*, Las Vegas, NV, June 23-26, 1985, 152-158.

[Nad79] A. Nádas, "Probabilistic PERT," *IBM Journal of Research and Development 23* (3):339-47, May 1979.

[Nad80] A. Nádas, "Random Critical Paths," *Proceedings of the 1980 IEEE International Symposium on Circuits and Systems, Part I*, Houston, TX, April 1980, 32-5.

[NaP73] L. W. Nagel and D. O. Pederson, "Simulation Program with Integrated Circuit Emphasis," *Proc. 16th Midwest Symp. Circ. Theory*, Waterloo, Canada, April 1973.

[NeV86] H. C. Neto and L. M. Vidigal, "A Novel Timing Verification Technique," *Proceedings of the Integrated Circuit Technology Conference*, Limerick, Ireland, 1986, 246-251.

[NG81] P. Ng, W. Glauert and R. Kirk, "A Timing Verification System Based on Extracted MOS/VLSI Circuit Parameters," *Proceedings of the 18th IEEE/ACM Design Automation Conference*, Nashville, 1981, 288-292.

[NST82] M. Nomura, S. Sato, N. Takano, T. Aoyama and A. Yamada, "Timing Verification System Based on Delay Time Hierarchical Nature," *Proceedings of the 19th IEEE/ACM Design Automation Conference*, Las Vegas, NV, 1982, 622-628.

[Ous83] J. K. Ousterhout, "Crystal: A Timing Analyzer for nMOS VLSI Circuits," *3rd CalTech Conference on VLSI*, Pasadena, CA, 1983, 57-69.

[Ous84] J. K. Ousterhout, "Switch-Level Delay Models for Digital MOS VLSI," *Proceedings of the 21st IEEE/ACM Design Automation Conference*, Albuquerque, NM, 1984, 542-548.

[Ous85] J. K. Ousterhout, "A Switch-Level Timing Verifier for Digital MOS VLSI," *IEEE Transactions on Computer-Aided Design CAD-4* (3):336-349, July 1985.

[Pen85] J. M. Pendleton, *A Design Methodology for VLSI Processors*, PhD. Thesis, University of California, Berkeley, Berkeley, CA, December, 1985.

[PeR81] P. Penfield, Jr. and J. Rubinstein, "Signal Delay in RC Tree Networks," *Proceedings of the 18th IEEE/ACM Design Automation Conference*, Nashville, 1981, 613-617.

[ReC86] R. Reddi and C. Chen, "Hierarchical timing verification system," *Computer-Aided Design 18* (9):467-471, November 1986.

[RoF80] J. P. Roth and A. L. Frisiani, "Computer Timing Verification," RC 8335 (#36281), IBM Thomas J. Watson Research Center, Yorktown Heights, NY, July 1, 1980. A copy is available at the Stanford University Mathematical Sciences Library, cataloged as Technical Report #014593.

[RPH83] J. Rubinstein, P. Penfield, Jr. and M. A. Horowitz, "Signal Delay in RC Tree Networks," *ToCAD CAD-2* (3):202-211, July, 1983.

[SYA81] T. Sasaki, A. Yamada, T. Aoyama, K. Hasegawa, S. Kato and S. Sato, "Hierarchical Design Verification for Large Digital Systems," *Proceedings of the 18th IEEE/ACM Design Automation Conference*, Nashville, 1981, 105-112.

[ShT83] J. H. Shelly and D. R. Tryon, "Statistical Techniques of Timing Verification," *20th Design Automation Conference*, Miami Beach, 1983, 396-402.

[Ste85] R. B. Steck, *Entwicklung und Implementierung von Verfahren zur Verifikation des Zeitverhaltens hochintegrierter digitaler Systeme [Development and Implementation of Algorithms for the Verification of the Timing Behavior of Large Scale Integrated Digital Systems]*, Technische Hochschule Darmstadt, Institut für Halbleitertechnik, FG

Halbleitershaltungstechnik, 1985. Studienarbeit (in German). English-language appendix is an early draft of the 1986 D.A. Conference paper that includes equations left out of the final paper.

[Ste84] G. L. Steele, Jr., *Common LISP: the Language*, Digital Press, 1984.

[Szy86] T. G. Szymanski, "LEADOUT: A Static Timing Analyzer for MOS Circuits," *ICCAD-86*, Santa Clara, California, November, 1986, 130-133.

[TON83] E. Tamura, K. Ogawa and T. Nakano, "Path delay analysis for hierarchical building block layout system," *Proceedings of the 20th IEEE/ACM Design Automation Conference*, Miami Beach, 1983, 403-410.

[TAR84] D. R. Tryon, F. M. Armstrong and M. R. Reiter, "Statistical failure analysis of system timing," *IBM Journal of Research and Development* 28 (4):340-355, July 1984.

[Ung87] D. M. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, The MIT Press, Cambridge, Massachusetts, 1987. (PhD. Thesis from the University of California, Berkeley; republished as an ACM Distinguished Dissertation).

[Wol78] M. A. Wold, "Design Verification and Performance Analysis," *Proceedings of the 15th IEEE/ACM Design Automation Conference*, Las Vegas, 1978, 264-270.