Pearl: A CMOS Timing Analyzer

James J. Cherry
Symbolics Cambridge Research Center
11 Cambridge Center
Cambridge, MA 02142

Abstract

Pearl is a timing analyzer that has been used to verify both full custom VLSI and gate array designs. Rather than verify that a design meets a given clock timing, Pearl automatically determines the minimum error free clock period and duty cycles. Delay equations compiled for the circuit facilitate fast incremental analysis when false paths are blocked or transistor sizes are changed while optimizing the circuit. Pearl is one component of a large, tightly integrated set of design tools. This allows it to report delay paths graphically in the schematic editor, greatly improving interpretation of results over simple textual reports. Because it can handle networks that mix transistors and functional models it is able to accurately analyze circuits in which switch level timing models fail. We also describe transistor signal flow direction rules for CMOS circuits used to eliminate false paths.

1 Introduction

Pearl is a timing analyzer designed for analyzing clocked synchronous digital circuits. The goal of analyzing this class of circuits is to determine the minimum clock period and duty cycle of the circuit's clocks. More specifically, given an ordering of the clock edges used in the circuit, we wish to determine the earliest time each clock edge may occur while maintaining correct behavior of the circuit. Previously reported timing analyzers [1, 2, 3, 4] take a different approach, where the time of each clock edge is specified and the timing analyzer looks for violations of the clock specification. The problem with this approach is that in many cases the goal is to design a circuit that is as fast as possible, rather than one that fits into a pre-defined clocking scheme. By starting with a slow clock specification and tightening it on successive iterations (relaxation) until clock violations occur the minimum clock period can be determined, albeit slowly. Pearl takes a more direct approach, solving for the clock period (edge times) directly by determining the timing constraints between clock edges necessary to ensure correct behavior of the circuit and solving them.

In this paper we first describe the mechanism used to determine the timing relationship each node in the circuit has with respect to the clock edges. Next, we show how these dependencies together with the setup and hold time requirements of latches and registers in the circuit are used to formulate timing constraints between the clock edges. These timing requirements are then solved using a linear programming algorithm to determine the minimum time of each clock edge. The algorithm is first described for the case of a circuit composed of functional models. Next, we show how the algorithm is applied to MOS switch circuits.

2 Building the Causality Graph

The first analysis step is to determine when each node in the circuit changes with respect to each of the clock edges. This is achieved by "compiling" node delay dependencies into a causality graph that represents the delay and state changes of a node in terms of nodes that cause it to change [4]. For example, consider the circuit shown in figure 1 with its associated causality graph. When clock PH1 rises, it causes the Q output of the register (node N1) to change. Since we do not know the register's contents, we assume that its output may either rise or fall when it is clocked. Each node in the causality graph is called a *delay event* and corresponds to a rising or falling node transition. Separate delay events are used for rising and falling transitions to accurately model asymmetric rise/fall times. Delay events are linked by arcs called *delay equation terms* that correspond to device pin to pin delays. For the causality graph in figure 1 we see that the delay events (N1↑) and (N1↓) have terms for the CLK to Q delay of REG. Together the delay equation terms at a delay event form an "equation" that is used to determine the time of a delay event by maximizing the arc delay plus the input event's delay.

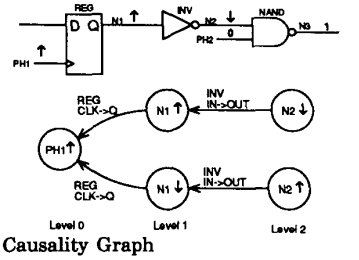

Figure 1. Causality Graph

The input edge at the root of the causality graph is termed the *primary input*. In this example we have chosen PH1↑ as the primary input. When analyzing sequential circuits the primary input is a clock edge. However, the analysis can be applied to any input of a circuit, which is useful for finding longest paths in combinational logic. Once the causality graph has been constructed, it remains in virtual memory to support fast incremental timing analysis.

The first step in determining the causality graph is to propagate constant node values with a simulator. Constant node values can be user supplied for case analysis, but the most important constants to propagate are the known values of clock nodes. For example, when analyzing the rising edge of the first clock phase in a two phase clock, the second clock phase is known to be low. This fact is important in blocking a large class of false paths. The simulator used is a version of RSIM [5] that has been generalized to handle functional models written in LISP as well as MOS switches. RSIM and Pearl use the same network data structures consisting of nodes and devices interconnected with terminals.

One aspect of the library database is a set of pin to pin delay relationships for each primitive device type in the network such as a gate or register. The delay relationships specify both delay and causality information. For example, a rising edge on the input of an inverter can cause the output to fall but not rise. The pin to pin delays for an inverter are expressed as follows:

```
(in↑ → out↓ :delay 1.0 :slope .1)
(in↓ → out↑ :delay 1.2 :slope .3)
```

The delay model has two components, a load independent intrinsic delay (:delay) and a load dependent delay (:slope) that is multiplied by the output node load capacitance. The slope (slew rate) can also be specified as a transistor size if the functional model is used to model an MOS switch circuit.

The causality graph is constructed by a depth first recursive walk of the network that visits each node at most once. Starting from the delay event corresponding to the primary input, all device terminals that the node is connected to are visited. For each delay relationship from the device input terminal to a device output terminal a delay equation term is added to the output node's delay event. If the output delay event does not exist it is created and the algorithm is recursively applied to the new delay event. If the output delay event already exists its consequences have already been determined and the recursion is terminated. Nodes on the current search path during the search are marked to detect feedback loops. No equation term is generated if the output delay event is on the current search path. If the output node has a constant value propagated by simulation, no delay events are constructed for it. For example, consider the transition on the output of the inverter (N2) in figure 1 while analyzing PH1 rising. No transition on the output of the NAND gate (N3) results because the output is known to be high from propagating the constant low value of PH2.

Once the causality graph has been constructed the delay events are ordered by the number of "logic levels" between them and the primary input. The level of a delay event is one greater than the maximum level of the delay events that can cause it to occur. When the level for a delay event is requested, it is recursively computed if it has not been determined. If it has already been determined it is simply returned. Thus each delay event is visited only once in the levelizing algorithm. Delay events at the same level are linked together and a table indexed by level to the head of each list is created.

At this point it becomes a trivial matter to determine the minimum and maximum delay between the primary input and any node in the circuit that it affects. Delay events are visited from the lowest level (closest to the primary input) to the highest level, computing the delay by maximizing the contributions from each equation term. Each equation term contribution is simply the input delay event's delay plus the pin to pin delay of the equation term. Since the "equations" are ordered, the delay of all input delay events have already been computed. Another way to view this is that the delay equations are evaluated in a breadth-first manner [3, 4]. The equation term that determines the maximum delay for a delay event is recorded as the *zero slack term*. Minimum as well as maximum delays are calculated and recorded to specify the range of time that the node is unstable. For example, if the input to output delay for the NAND gate in figure 2 is 2ns, the earliest the output can fall is 11ns, and the latest 12ns. Minimum and maximum delays can differ because of reconvergent delay paths as in this example. Pin to pin delay variations that can also lead to delay ranges are not modeled in the current implementation, although they could easily be incorporated.

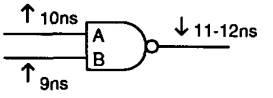

Figure 2. Minimum/Maximum Delay

3 Using the Causality Graph to Find Delay Paths

Although determining the delay dependencies is usually a component of determining clock edge requirements, it is also useful for analyzing combinatorial logic (no clocks) or one of the clock edges individually. For this reason, a command is provided to determine the longest delay paths from a rising or falling transition on any node in the circuit.

Delay paths terminate at delay events that do not trigger any other delay events, such as register inputs or output ports. When the delay equation for a delay event is evaluated, if it does not trigger any further events its delay is compared to the delay of previously encountered worst delay events. A user specified number of worst paths are recorded for reporting to the user. The longest delay path to an output is determined by tracing zero slack terms backwards until reaching the primary input. For this reason only the worst terminal delay events need to be recorded instead of explicit delay paths consisting of all the delay events on the path. To reduce the number of redundant paths reported to the user only the worst of the falling and rising delay to a node is recorded. Additionally, paths that differ by only the bit position in a bus anywhere along the path are collapsed into one path, the longest one being recorded.

Pearl is just one of many tightly integrated tools that are collectively known as the NS design system [6, 7]. All tools share the same user interface and address space. Timing analysis is invoked from the design editor in which the schematic and layout are captured. As the user steps through the longest delay paths they are printed with the largest pin to pin delays in the path highlighted in bold. The user can click the mouse on a device printed in the critical path to view the schematic it appears on. Additionally, delay paths are graphically displayed on the schematics by showing their delay next to icon terminals the path traverses (figure 3).

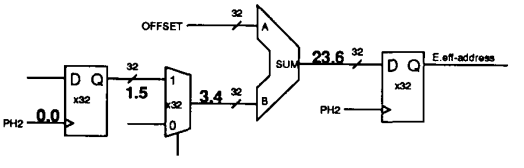

Figure 3. Graphical Delay Path

Since the delay information on a node is kept in its associated delay events, the user can interrogate any node on or off of the current delay path to determine the minimum and maximum time at which it changes with respect to the primary input. This is in contrast with timing analyzers that collect only delay paths [8]. The ability to find the delay to arbitrary nodes lets a user assess proposed changes to the circuit to improve its performance. For example, speeding up the A input to the NAND gate in figure 2 by 3ns will only improve performance by 1ns because the B input arrives 1ns later than the A input. If only the worst path is reported the designer (or an optimization program) is unable to determine how much slack there is on the other paths intersecting the worst path.

Because the timing analyzer is designed to find paths that are independent of circuit node values, inevitably some false paths are reported. For example, the microcode of a processor may never exercise a feature that the hardware supports. A false path can block interesting paths from being detected because the timing analyzer records the maximum delay at nodes. False paths can be blocked at any stage along a delay path by graphically specifying a delay equation term to ignore. The timing analysis can be incrementally re-run without rebuilding the causality graph by simply evaluating the delay equations of delay events that follow the blocked term. Path blocking can also be useful for performing experiments of the form:

> *Suppose this path (gate) was sped up, what are the next most constraining delay paths?*

False paths that are blocked may be saved in a file and restored for future use. Transistor sizes can also be incrementally changed to examine circuit performance impact.

4 Determining Clock Edge Requirements from the Causality Graph

Armed with the ability to determine when nodes change with respect to the clock edges we can return to the goal of solving for the clock period and edge times. To illustrate how this differs from simply finding the longest paths for each clock edge, consider the circuit shown in figure 4. In this example level sensitive latches similar to those commonly employed in MOS circuits are clocked with a two phase clock, with some arbitrary computation taking place between them. To simplify the discussion, assume that the CLK to Q and D to Q delay of the latches are both 1ns, independent of load and edge direction. Also assume that the setup and hold time of the latches are zero. When clock PH1 rises, nodes A, B, C and D change at times 1ns, 11ns, 12ns and 32ns respectively. For the correct data value to be latched by LATCH-3 the input must arrive before its clock input PH2 turns off. Thus, there must be at least 32ns between the time PH1 rises and PH2 falls. Similarly, there must be at least 16ns between PH2 rising and PH1 falling for LATCH-4 to latch the correct data. The setup time requirement of LATCH-2 forces PH1 to remain high for at least 11ns. Notice that in a timing analyzer only concerned with longest paths this last requirement could easily be overlooked, since the path to the D input of LATCH-2 is short compared with the longest path triggered by PH1 rising.

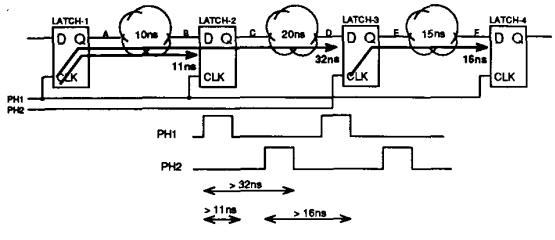

Figure 4. Clock Timing Requirements

This example illustrates the essence of how Pearl determines clock edge times. By examining setup and hold time requirements between inputs and outputs of devices in the circuit and relating them to the clock edges that trigger them using the causality graph, constraints between the clock edges can be derived. These constraints take the form of linear inequalities that are solved with a linear programming algorithm.

Pearl is not restricted to built in clocking schemes. Instead, the clocking scheme is described as the set of the clock node names and the order in which they occur. For example, the following form is used to define two phase non-overlapping clocks named PH1 and PH2:

```
(define-clocks two-phase "PH1↑" "PH1↓" "PH2↑" "PH2↓")
```

This defines the logical sequence that the clock edges must occur, but not the exact timing information about when each edge occurs. The goal of timing analysis is to determine the time that each edge occurs such that correct behavior of the circuit it guaranteed.

A causality graph is built for each clock edge in sequence by making it the primary input and applying the algorithm described in section 2. The primary input is set to the CHANGING state, and the other clocks to their respective values when the clock edge occurs. For example, PH2 is low during both PH1 edges for two phase clocks. Each node records a set of delay events indexed by the clock edge that triggers the event and the rising or falling direction of the transition. Thus, a node can have up to 2N delay events, where N is the number of clock edges.

Timing restrictions on inputs of devices that are necessary to ensure error free operation of the circuit are termed *local timing requirements*. Examples are setup time, hold time, and minimum clock pulse width for latches, registers or memories. Functional models in the design explicitly specify local timing constraints in their definition. In MOS switch circuits local timing constraints are determined by identifying implicit latches and registers by finding dynamic storage nodes. This is described in more detail in section 5.

Local timing requirements between inputs and outputs of devices are related back to the primary clock input edges using the delay events recorded on nodes. For example, a setup time requirement on an edge triggered register states that the D input must be stable T_{setup} before the clock rises.

$$T_{clk\uparrow} \geq T_{D} + T_{setup}$$

Suppose the clock input of the register rises 5ns after clock PH2 rises, and the D input changes 23ns after PH1 rises. Substituting this into the equation above, the inequality becomes:

$$T_{PH2\uparrow} + 5ns \geq T_{PH1\uparrow} + 23ns + T_{setup}$$

By rearranging this inequality we obtain a relationship that specifies a minimum time between PH2 rising and PH1 rising.

$$T_{PH2\uparrow} - T_{PH1\uparrow} \geq 23ns - 5ns + T_{setup}$$

If other clock edges cause the D input to change or the clock input to rise, they too will generate spacing requirements of this form. Notice that the setup time requirement is more likely to be violated for an early arriving clock than a late arriving clock. For this reason both the minimum and maximum time that a delay event can occur are calculated when the delay equations of the causality graph are evaluated. The minimum delay to the register's clock input and the maximum delay to the data input are used to formulate a setup timing constraint.

A table of the worst spacing requirements between each pair of clock edges is built by visiting each device in the network and recasting its local timing requirements as spacing requirements between pairs of clock edges. For example, a setup time requirement on a register turns into a spacing requirement between the clock edge that causes the register's data input to change and the clock edge that causes the active edge of the register's clock input. Each spacing requirements is of the form:

$$T_i - T_j \geq \text{spacing} \geq 0$$

where the clock edges are numbered from 1 to N, and T_i denotes the time of clock edge i. The first clock edge (T_1) is defined to occur at time 0. The equation above shows the case when clock edge i chronologically follows clock edge j (i>j). If instead clock edge j chronologically follows clock edge i (i<j), the inequality is interpreted as a requirement extending to the edge in the next clock phase and the inequality becomes:

$$T_i + T_{period} - T_j \geq \text{spacing} \geq 0$$

Where T_{period} is the clock period. If clock edge i and clock edge j are the same edge (i=j) then the inequality is is a requirement on the clock period and the equation above simplifies to:

$$T_{period} \geq \text{spacing} \geq 0$$

If there are N clock edges, there are as many as N^2 spacing requirements. Additionally, one inequality is added for each clock edge that ensures the clocks edges have the correct chronological ordering.

$$T_{i+1} - T_i \geq 0 \text{ for i from 1 to N-1} \quad (4)$$

The inequalities are then solved using the simplex method [9], with the clock period as the minimization criteria. The solution is the minimum clock period and edge times that guarantee error-free operation.

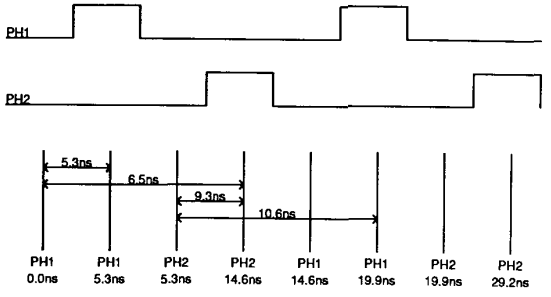

Figure 5. Automatically Generated Timing Diagram

A timing diagram showing the earliest time each clock edge can occur and the longest constraints found between clocks is automatically produced (figure 5). Two periods of the clocks are drawn so that constraints between edges from one cycle to the next are visible. Constraints between clock edges are graphically depicted as arrows between edges, labeled with the minimum time between the edges that satisfies all local timing constraints in the design. The user can examine the edge spacing requirements between each pair of clock edges, with the worst delay path graphically displayed on the schematics.

When the same clock edge triggers both the data and clock input events involved in a timing requirement, there is no degree of freedom to move clock edges to satisfy the requirement. Timing requirements that cannot be satisfied for this reason are reported as errors.

5 Application to CMOS Circuits

Unlike a gate array, a full custom VLSI circuit is not built out of a fixed set of primitive gates and registers. Although the number of circuit topologies may be relatively small, the transistor sizes used in different instances of the topology are typically tuned to optimize performance. Furthermore, the delay of circuits that employ pass gates at their inputs or outputs are difficult to abstract without knowledge of the context they are used in. For these reasons, constructing a library of well characterized functional models to model the circuits used in a full custom circuit is difficult. To accommodate full custom designs Pearl has been structured to allow networks composed of MOS transistors modeled as resistive switches. This section explains how the causality graph and setup/hold time requirements of switch networks are determined.

When analyzing circuits composed of MOS switches we must infer the causality relationships between delay events. Rather than recognizing groups of switches into higher level primitives such as gates and latches, switches are grouped into *stages* as in Crystal [8]. A stage is a series of source/drain (channel) connected switches between a power or ground node and a node connected to one or more transistor gates or functional model inputs. A stage is *triggered* by one of the series transistors turning on, or when a transistor turns off and enables a passive pullup (an NMOS depletion load, or a grounded gate P channel device in a ratioed CMOS circuit). When a stage is triggered the output node of the stage is charged to the power or ground rail at the source of the stage. Figure 6 shows a simple circuit and its corresponding stages. Each stage corresponds to an input/output delay relationship as explicitly specified in the case of functional models. One advantage of using stages to determine event causality is that they elegantly handle networks containing pass transistors.

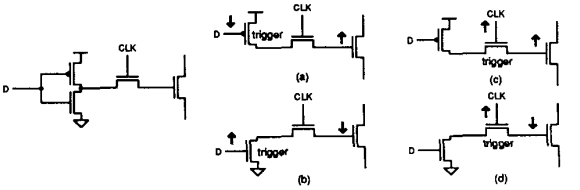

Figure 6. Sample Circuit and Corresponding Stages

Rather than enumerating all possible stages in the network, stages are determined as required while building the causality network. Each transistor that can be turned on by a delay event is considered as a trigger for possible stages. Stages are found by searching from the source (drain) node of the transistor until power or ground is encountered, and from the drain (source) node until a node connected to transistor gates or a functional model input is encountered. A stage never passes through a transistor that is off by virtue of a constant value at its gate node. The parallel N and P transistors of a CMOS pass gate are recognized and treated as one switch in the stage, since the delay of either switch alone would be an overly pessimistic estimate. A delay equation term from the triggering delay event is added to the output node delay event for each stage that is found.

Figure 7 shows a pseudo-NMOS latch that uses a refresh clock (RFSH) to dynamically refresh the storage node (D-LATCHED). Consider the causality graph that will be constructed when the refresh clock rises. When the refresh clock rises, the storage node is pulled low which in turn causes the latch output to rise. The rising delay event on the latch output may in turn trigger many paths. In reality, the refresh clock cannot cause the latch output to rise because the refresh pulldown path depends on the output being high in the first place. To suppress this type of false path, no delay equation term is added when the input delay event depends on the output node's final value.

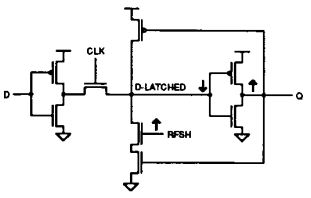

Figure 7. Latch with Dynamic Refresh

If a transistor is known to be unidirectional then we need only search either the source or drain of the trigger transistor for a power source, cutting the stage search space in half. Another benefit of identifying unidirectional transistors is that many false paths that otherwise satisfy the criteria for a stage can be eliminated. A pre-processing pass over the network is used to identify transistors as unidirectional. Previously reported rules for determining transistor signal flow [10] for NMOS circuits depend on transistor ratios to find pass transistors, rendering them virtually unusable for CMOS circuits.

All switches are initially assumed to be bidirectional. The first rule attempts to prove that flow is impossible from either the source to drain or drain to source. A search is made from the source (drain) of the switch through switch source/drain nodes looking for transistor gate inputs to some maximum search depth. If a transistor gate is found, information can flow towards gate so the marker on the switch that indicates flow is possible is left on. If only feedback paths or power sources are found, then it is safe to mark flow as impossible. Note that this rule is guaranteed to be safe. Some unidirectional devices may be left bidirectional, but no bidirectional devices can be marked unidirectional. The maximum depth of the search determines how hard we are willing to try and prove that flow is impossible in some direction. By limiting the search depth to reasonable level we can avoid getting lost in circuits that require inordinate amounts of computation.

The other rules that are used to mark transistor directions are not strictly safe, but cover the majority of switches that remain bidirectional after the first rule for the circuit methodologies we employ. If a switch is driven by a logic gate (has a pullup and pulldown) on the source but not on the drain, then it is marked as unidirectional allowing flow from the source to the drain. If the drain of the switch is connected to gates and the source is not, it is marked as unidirectional allowing flow from the source to the drain.

One circuit that these rules are unable to cope with is a barrel shifter constructed with pass transistors. In general the only way to determine signal flow in this type of circuit is to know what select nodes connected to the pass transistor gates are mutually exclusive. For circuits of this complexity flow tags in the schematic are used to mark these transistors as unidirectional [8].

Pearl currently uses an Elmore delay model [11] to calculate stage delays. Capacitance contributions of nodes between the source and trigger transistor are ignored because these nodes are assumed to be charged before the transition that triggers the stage arrives. To avoid overly pessimistic delays in CMOS pass gates, the resistance used assumes that both the N and P channel device are on. Because the delay model is only applied once per stage, a more accurate delay model could be efficiently applied. One model that we have experimented with uses a simple circuit simulator [12] that is optimized for solving the circuits found in stages.

Local timing requirements for switch circuits are inferred by identifying dynamic storage nodes and the switches that isolate the dynamic node. A pre-processing pass over the network marks nodes that have no path to power or ground when all of the clocks are disabled as dynamic storage nodes. For each dynamic node, those stages that are formed by the switch that isolate the dynamic node are identified. For example, stages (c) and (d) in figure 6. To meet the setup time requirement of the latch, each input event that enables the stage must arrive before the dynamic node is isolated. The setup time is simply the time it takes for the stage to charge the dynamic node. As with timing requirements in functional models, this local timing requirement is recorded in terms of the clock edges that cause the stage inputs to change and the pass gate to isolate the dynamic node. Additionally, the clock must be on long enough for the slowest stage to charge the dynamic node. Thus, a timing requirement is generated between the clock edge that causes the pass gate to turn off (isolating the dynamic node) and the chronologically preceding clock edge that turns on the pass gate.

Some circuits are particularly difficult to deal with at the switch level. One example is an adder that uses carry by-pass logic. The timing analyzer will ignore the by-pass path and report the longer delay path through the carry chain, since it has no idea that the two paths are logically equivalent. Another troublesome circuit is a six transistor RAM with shared bit lines. The access transistors of the RAM cell are bidirectional, since data is both read and written through them. The timing analyzer will find a multitude of paths that snake in and out of the individual RAM cells. If the access transistors are assumed to be used for only for writing, a setup time requirement is correctly identified but delay paths that originate from the read port of the RAM will be ignored, and visa versa. To handle these difficult cases the user can specify a functional model that is substituted for the switch level circuit. Substitution is done as the schematics are flattened into a network data structure by identifying them as primitives to the schematic extractor. SPICE simulations are used to characterize the pin to pin delays and timing requirements used in the functional model.

6 Relationship to Previous Work

Many of the ideas that Pearl implements are closely related to previous results, most notably those of LEADOUT [4]. The idea of a compiling delay expressions into causality graph comes directly from LEADOUT. The fundamental differences are how the causality graph is related to the clocks and how it is solved. Pearl builds separate causality graphs for each clock *edge* rather than a single causality graph that includes delay events from each clock *phase* as LEADOUT does. Given a specification of each clock phase duration, LEADOUT solves for the clock period and delay event times using simple relaxation, evaluating all delay equations until stability results. Pearl on the other hand does not require the duration of clock phases to be specified, and solves directly for the clock edge times without relaxation.

7 Results

We currently run the timing verifier on an entire chip. This avoids having a database with timing information for all global signals that connect sub-modules together. For example, delay paths can originate in one module and propagate into another before being latched. Running the timing verifier on only one of the modules does not provide any information about these inter-module delay paths. Although the timing analyzer works on symbolic layouts and masks as well as schematics, for large circuits we typically run it on schematics because the results are so much easier to interpret. Stray capacitances from the layout are used to

annotate the schematic network by running a network comparison [13] between the layout and schematic to identify node correspondence. Pearl was used to significantly improve the performance of a custom single chip CMOS LISP processor [14] that consists of 133K transistors in addition to functional models for all RAM, CAM and ROM modules. Run time to determine the clock times for the four clock phases used in the chip is approximately four hours on a Symbolics 3640. Incremental delay equation evaluation takes three minutes per affected clock edge, and 10 minutes to identify and solve the clock timing requirements.

Pearl consists of 3,000 lines of Common LISP. The Common LISP version of RSIM used by Pearl is 2,500 lines of code.

8 Conclusion

We have described a timing analysis program applicable to both gate arrays and custom CMOS circuits that is able to automatically determine the clocking requirements that result in error free operation of the circuit. By compiling the delay equations into a persistent data structure fast incremental analysis is supported. Furthermore, the algorithm is linear in the number of nodes in the circuit.

In the future we intend to characterize the setup and hold time requirements of non-clock inputs of the circuit. By doing so we can characterize each block in the hierarchy as a functional model and perform a hierarchical analysis that does not require flattening [15].

The author wishes to acknowledge many helpful discussions with Bruce Edwards and Chris Terman.

References

1. T. M. McWilliams. "Verification of Timing Constraints on Large Digital Systems,". *Journal of Digital Systems V(4)* (1981), 401-427.

2. R. B. Hitchcock. Timing Verification and the Timing Analysis Program. Proc. of the 19th Design Automation Conference, 1982, pp. 594-604.

3. Norman P. Jouppi. "Timing Analysis and Performance Improvement of MOS VLSI Designs". *IEEE Trans. Computer-Aided Design CAD-6*, 4 (July 1987), 650-665.

4. Thomas G. Szymanski. LEADOUT: A Static Timing Analyzer for MOS Circuits. IEEE International Conference on Computer-Aided Design, November, 1986, pp. 130-133.

5. Christopher Terman. Simulation Tools for Digital LSI Design. Tech. Rept. TR-304, MIT/LCS, MIT Laboratory for Computer Science, Cambridge Mass, September, 1983.

6. J. Cherry, H. Shrobe, N. Mayle, C. Baker, H. Minsky, K Reti, N Weste. NS: An Integrated design system. In E. Horbst, Ed., *VLSI '85*, Elsevier Science Publishers B.V. (North Holland), 1986, pp. 325-334.

7. J. Cherry. CAD Programming in an Object Oriented Programming Environment. In W. Fichtner, M. Morf, Ed., *VLSI CAD Tools and Applications*, Kluwer Publications, 1987, pp. 265-294.

8. John K. Ousterhout. "A Switch-Level Timing Verifier for Digital MOS VLSI". *IEEE Trans. Computer-Aided Design CAD-4*, 3 (July 1985), 336-348.

9. W. Press et al. *Numerical Recipes.* Cambridge University Press, 1986.

10. Norman P. Jouppi. "Derivation of Signal Flow Direction in MOS VLSI". *IEEE Trans. Computer-Aided Design CAD-6*, 3 (May 1987), 480-490.

11. P. Penfield Jr., J. Rubenstein. Signal Delay in RC Tree Networks, Proc. of the 18th Design Automation Conference, 1981, pp. 613-617.

12. B. Ackland, N. Weste. Functional Verification in an Interactive Symbolic IC Design Environment. Proc. 2nd Caltech Conf. on VLSI, 1981, pp. 285-298.

13. C. Ebling and O. Zajicek. Validating VLSI Circuit Layout by Wirelist Comparison. Proc. IEEE Interational Conference on CAD, sept, 1983, pp. 172-173.

14. C. Baker et al. The Symbolics Ivory Processor: A 40 Bit Tagged Architecture Lisp Microprocessor. Proc. IEEE International Conf. on Computer Design, October, 1987, pp. 512-515.

15. M. Nomura et al. Timing Verification System Based on Delay Time Hierarchical Nature. Proc. of the 19th Design Automation Conference, 1982, pp. 622-628.