Designing High-Performance Digital Circuits Using Wave Pipelining: Algorithms and Practical Experiences

Derek C. Wong, *Member, IEEE*, Giovanni De Micheli, *Senior Member, IEEE*, and Michael J. Flynn, *Fellow, IEEE*

***Abstract*—Wave pipelining is a technique for pipelining digital systems that can increase the clock frequency of practical circuits without increasing the number of storage elements. In wave pipelining, multiple coherent waves of data are sent through a block of combinational logic by applying new inputs faster than the delay through the logic. Ideally, if all paths from input to output have equal delay, then the circuit's clock frequency is limited by rise/fall times, clock skew, and setup and hold times of the storage elements. In practice, due to the above limits and variations in fabrication, clock frequency can be increased by a factor of 2 to 3 using the best available design methods.**

We present algorithms to automatically equalize delays in combinational logic circuits to achieve wave pipelining. The algorithms adjust gate speeds and insert a minimal number of active delay elements to balance input–output path lengths in a circuit. For both normal and wave-pipelined circuits, the algorithms also optimally minimize power under delay constraints. We present an analysis of the algorithms and comment on their implementation. Then we report experimental results, including the design and testing of a 63-bit population counter in CML bipolar technology.

A brief analysis of circuit technologies shows that CML and super-buffered ECL without stacked structures are well suited for wave pipelining because such technologies have uniform delay. Static CMOS and ordinary ECL including stacked structures and emitter-followers do have some delay variations, depending on the input patterns. A high degree of wave pipelining is still possible in those technologies if special design techniques are followed.

I. Introduction

WAVE PIPELINING is a design method that can boost the pipeline rate of a system without using additional registers. In ordinary pipelined systems, there is one "wave" of data between register stages. When a new set of values is clocked into one set of registers, the values are allowed to propagate to the next set of registers before the first set is clocked again.

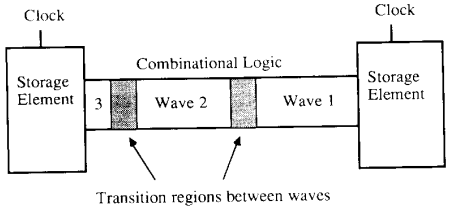


Fig. 1. In wave pipelining, multiple coherent waves of data are sent through combinational logic acting as a pipeline.

In contrast, wave pipelining is the use of multiple coherent "waves" of data between storage elements (see Fig. 1). This is achieved by clocking the system faster than the propagation delay between registers. In this method, the data values at the first set of registers are changed before the old data values have propagated to the next set of registers. The capacitance in the combinational logic circuit is being used to store values for pipelining.

For example, a fast 64-bit floating-point multiplier implemented in combinational logic might have a propagation time of 10 ns. If we used wave pipelining and neglected register setup–hold times and clock skews, the multiplier could operate using three pipeline waves to achieve a clock frequency of 300 instead of 100 MHz with the same 10-ns latency. At time 0 ns, the first wave of data would be clocked into the left register and begin propagating through the logic. This first wave would reach about $\frac{1}{3}$ of the way through the logic before the second wave starts at time 3.33 ns. The third wave would start at time 6.66 ns. At this point, the first wave is $\frac{2}{3}$ of the way through the logic and the second wave is $\frac{1}{3}$ of the way through. When time 10 ns is reached, the results of the first wave are stored in the right register while the fourth wave is started from the left register. Thus, three waves are simultaneously present in stages of the combinational logic and the clock frequency is tripled compared to the normal rate.

Manuscript received May 24, 1991; revised March 27, 1992. This work was supported by the National Science Foundation under an NSF Graduate Fellowship, by the Center for Integrated Systems, Stanford University, CA, and by the National Science Foundation under Contract MIP88-22961, using equipment provided by NASA under Contract NAGW 419. The demonstration chips were manufactured at Signetics, Inc. CAD tools were provided by Mentor Graphics, Inc. Chip testing was performed at Trillion, Inc. G. De Micheli was supported in part by the NSF, DEC, and AT&T under a PYI award. Use of equipment was provided by Philips/Signetics, Inc. This paper was recommended by Associate Editor R. K. Brayton.

The authors are with the Department of Electrical Engineering, Stanford University, Stanford, CA 94305.

IEEE Log Number 9202983.

Using two additional registers, the clock rate could also be nearly tripled using regular pipelining, but the registers would increase the input–output latency,[1] consume additional power and layout area, and increase the clock distribution requirements.

To achieve the highest possible wave-pipelining frequency, all path delays from every starting to every ending storage element must be the same. Clock skew, variations in path length, rise–fall times, and the setup time of the storage element limit the maximum pipeline rate.

The possible advantages of wave pipelining are the following:

- lower power, area, and delay by using fewer stages of storage elements;
- very high rates of pipelining without the added delay of storage elements dominating the pipeline latency;
- wide applicability to all pipelined digital systems.

Wave pipelining has disadvantages as well:

- Requirement for specialized design algorithms to equalize the length of all paths.
- More difficulty exists at the system level. A wave-pipelined chip must be run within a relatively narrow range of clock frequencies. If the chip were operated at a substantially different frequency, the number of waves within a logic section might be incorrect. When a system using multiple wave-pipelined chips is assembled, the chips must have matched speeds.
- The need to add delay buffers to lengthen some short paths. This increases the area of some circuits.

Previous and concurrent work in this area includes the following:

- Anderson and Cotten first described the concept as used in the floating point unit of the IBM 360/91 [1], [4]. The clock frequency using wave pipelining was twice the normal frequency.
- Lin and Xia also designed and implemented an experimental computer using wave pipelining in its arithmetic units [19].
- Fawcett described the theory of pipelined system clocking in detail [9]. He developed detailed equations for the maximum clock rate of Earle latch systems as a function of many parameters. His experimental work did not include wave pipelining, although his clocking theory applies.
- Ekroot developed a theory of wave pipelining [8]. Assuming gates and modules with fixed delays, he developed linear programs to determine where to insert delay elements to balance the circuit. Also, he compared the minimum clock period using wave pipelining and regular pipelining.
- Klass and Mulder have studied the use of wave pipelining in CMOS [13], [14]. A 4-bit adder has been simulated with effective results.
- Gray *et al.* have designed and built a wave-pipelined adder as well as wave-pipelined delay chains in CMOS [10], [11].
- Joy and Ciesielski have developed placement and routing algorithms for laying out wave-pipelined circuits [12].
- Lien and Burleson have applied wave pipelining to domino logic [18].
- Chappell *et al.* have described a high-performance SRAM which uses a concept similar to wave pipelining called bubble pipelining. The RAM has an access time of 3.8 ns but has a pipelined cycle time of 2 ns [3].

Our research goal has been to develop the necessary analytical tools and design techniques to build an actual wave pipelining chip in VLSI, identifying and solving the necessary practical problems enroute. We have focused on the following areas:

1) analyzing technologies and speed limits,
2) designing the algorithms and developing the necessary CAD tools to automatically balance the delays in combinational logic circuits,
3) designing, building, and testing a sample chip design.

The IBM 360/91 and the experimental computer by Lin and Xia were designed using manual design techniques to balance circuits of fixed-delay gates. In contrast, we are developing new algorithms to balance the delays using gates with adjustable delay. Unlike Ekroot's methods, our methods minimize power consumption and added circuitry. Also, we use gates with adjustable delay versus power, rather than assuming fixed delays.

This paper discusses some of the important issues in wave pipelining. First, we explain the frequency limits of wave pipelining. Next, we introduce new algorithms that balance delays to achieve wave pipelining. As a byproduct, these methods can also be used to optimize power versus delay for both normal and wave-pipelined circuits. We then describe the results of applying these algorithms to various example circuits. We present a wave pipelining chip designed using these methods that actually operates at 2.5 times the ordinary clock frequency. In Appendix II, various technologies are graded by their suitability for wave pipelining. We show why ECL and CML are well suited for this technique and why special design techniques are required by CMOS.

Additional descriptions of our work can be found in [24]–[27].

II. Wave Pipelining

A. The Minimal Clock Period Relation

The maximum pipeline rate is limited by technological parameters. Clocking the circuit at a frequency above the limit would mix the waves of data together.

[1]In this paper, the words *latency* and *delay* are used interchangeably. They both denote the time required for a data wave to propagate from the circuit inputs to outputs.

In Appendix I, we show that the minimum clock period at which a wave-pipelined circuit can be clocked is bounded by

$$t_{CP} > \Delta t_P + 2 * \Delta C + t_{SH} + t_{RF} \quad (1)$$

where

- t_{CP} clock period,
- t_P propagation time of the longest path in the combinational logic,
- Δt_P maximum difference between longest and shortest path lengths over worst-case design, process, and environment,
- ΔC worst case uncontrolled clock skew,
- t_{SH} setup plus hold time for edge-triggered registers (for latches, t_{SH} = length of transparent period plus hold time),
- t_{RF} worst-case rise or fall time (10%–90% voltage swing) at the last logic stage.

In contrast, the normal clock period t_{CPN} without using wave pipelining would be bounded by $t_{CPN} > t_P + t_S$ (+ possibly ΔC depending on the design technique) where t_S is the setup time of the storage element. In most cases, t_{CP} can be made much smaller than t_{CPN} by reducing the Δt_P to a small fraction of t_P.

B. Reducing the Minimal Clock Period

A designer would like to minimize t_{CP} by attacking each of its components. We assume here that t_{SH} and t_{RF} are parameters that depend on the technology. The clock skew ΔC can be minimized using standard design techniques.

Therefore, we consider the problem of minimizing t_{CP} by reducing Δt_P.

The path variation Δt_P arises from several sources:

- path differences due to design,
- process and temperature-induced variations within one chip,
- data-dependent delay variations.

Process and temperature-induced variations are unavoidable, but their effects are limited within one chip. Data-dependent delays can be limited by selecting the proper technology. Using our algorithms, the worst-case Δt_P can be reduced to 10%–30% of t_P in most cases.

For example, assume that in a hypothetical technology ΔC, t_{SH}, and t_{RF} are equal to 500 ps each (or one gate delay). Then the clock period is bounded by

$$t_{CP} > \Delta t_P + 2 \text{ ns}.$$

In this case, Δt_P can be reduced to 1 to 3 ns for a circuit that has t_P = 10 ns (20 gate delays), leading to a clock period t_{CP} of 3 to 5 ns. Thus, two to three waves of data can propagate simultaneously within a typical wave-pipelined circuit leading to a doubling or tripling of the normal clock rate.

C. Using Wave Pipelining in a System Design

A combinational circuit like a floating point adder is the ideal type of circuit for wave pipelining. Typically, a system designer may have one of two design goals when using wave pipelining:

1) Build the fastest possible circuit that can handle as many waves as possible. The designer will fit the rest of the system to the circuit after t_P and t_{CP} are determined.

   In this case, the desired delay t_P is equal to the critical path delay with all gate parameters set to achieve minimum delay. The design is balanced by a tuning algorithm that minimizes Δt_P while keeping the delay equal to t_P.

2) Build a circuit that has a delay t_P determined by the design of the rest of the system. For example, this might occur if the clock period is based on cache speed and the semantics of the system design require that the adder be three waves deep.

   In this case, the desired delay t_P is determined by external constraints. The objective is to minimize Δt_P at that delay.

Our algorithms are designed to balance the circuits to a user-specified nominal propagation delay called D_{MAX} while minimizing power and added area. If desired, the designer can iterate while varying D_{MAX} to construct a range of solutions with varying clock period t_{CP}, power, and area.

III. Algorithms for Designing Wave-Pipelined Circuits

If possible, a wave-pipelined circuit should be designed to have balanced paths under nominal fabrication and temperature conditions. When a technology satisfies some assumptions, this can be achieved by using our balancing algorithms. In Appendix II, we showed that technologies exist that satisfy these assumptions.

Next, we describe the CAD algorithms and tools for automatically taking a combinational logic circuit and balancing its delays. One of the tools can also be used to set the gate drives in both normal and wave-pipelined circuits to achieve the minimal possible power for a given maximum delay D_{MAX}.

Our methods are designed primarily to work with ECL/CML circuits.

A. Problem Formulation

This section describes the algorithms that can be used to balance the nominal I/O delays of a circuit to a given maximum delay D_{MAX}. If fabrication parameters such as doping levels and oxide thickness vary little within a single chip, the circuit will still be quite balanced under all possible fabrication conditions within the process envelope. Secondary goals in designing a circuit are to minimize area and power.

The following modeling assumptions are made.:

1) The circuit is combinational without feedback.
2) Each gate propagates signals one way from inputs to outputs.[2]

[2]Gates may have both an inverting and noninverting output.

3) Gate delay can be adjusted by a parameter, called *gate drive*.[3] Delay adjustment does not affect area.
4) Adjusting the delay of a gate does not affect the delays of gates connected to its inputs or outputs.
5) Path delays can be increased by inserting active delay elements. These elements are buffers with one input and one noninverting output.
6) When all gates are set to maximal drive, the longest path delay should be smaller than or equal to D_{MAX}. No technique can balance the circuit to D_{MAX} if this assumption is not met. It is straightforward to verify when circuits meet it.

These assumptions are satisfied more easily in ECL/CML circuit families based on bipolar circuit technology. In ECL/CML, each gate's delay can be adjusted by controlling the gate's tail current without affecting the delays of other gates. We can also safely assume that gates can be designed in this technology such that the I/O connections and total area of each gate are independent of the gate's drive. In addition, the gate delay is a monotonically decreasing convex function of power. An example of a CML gate's power-delay characteristic is shown in Fig. 8.

Path delays are computed by adding the propagation delays of the gates. The CAD methods do not take advantage of possible false paths.[4] The methods are conservative in that all I/O delays are balanced to the delay of the longest path, which could possibly be false.

B. Rough and Fine Tuning

We propose two ways of attacking the balancing problem, inserting delay elements and adjusting gate drives. We combine these in a multistep process to balance a circuit while minimizing power and added area:

1) *Rough tuning* inserts a minimal number of delay elements so that it is feasible to balance the circuit by just adjusting gate drives. Minimizing the number of inserted elements minimizes the added area.
2) *Fine tuning* adjusts gate drives so that the circuit is nominally balanced to a delay D_{MAX} with minimal power consumption.

For some circuits and delay models, one tuning method suffices to balance the circuit. However, in the general case, we combine the two tuning methods in order to balance the circuit.

The primary goal for the two tuning techniques is to achieve exact delay balancing, i.e., guaranteeing that all I/O path delays are equal to D_{MAX}. The secondary goal is to minimize the added area and power. In this section, we show that under some additional assumptions the balancing problem, as well as the power and area minimization problems, have an exact solution. But we also show that exact balancing may be hard to achieve for practical circuit models and layout methods. However, the use of these techniques on actual designs shows that the methods do balance the path delays within a reasonable tolerance. Since the remaining imbalance is small compared to t_P, wave pipelining can provide an effective means of speeding up the circuit.

In practice, we propose the following procedure to balance circuit delays.

Definition Tuning Procedure:

1) First, an optional fine-tuning pass may be performed prior to placement and routing using a coarse estimate of capacitative loads.
2) Then rough tuning is performed to fix the remaining imbalances in delay.
3) A final fine-tuning pass is performed after placement and routing, when the value of each wire's capacitance can be extracted from the layout. Since the gates have the same area and I/O connections independent of the chosen gate drive, the gate drives can be fine tuned without changing any wire routing.
4) Following fine tuning, the maximum pipeline rate can be determined. □

The optional first fine-tuning pass reduces the number of imbalanced paths prior to rough tuning, thus reducing the number of delay elements inserted. Because the actual capacitances after layout can differ substantially from the initial estimates, a final fine-tuning pass is necessary.[5]

In the next two sections, we present both fine and rough tuning in detail. To make the theoretical description easier, each algorithm uses its own graph representation of circuits.

IV. Rough Tuning

We describe the rough-tuning problem first. As the name suggests, rough tuning attempts to balance a circuit by inserting buffers in the circuit to add discrete delays.

Ideally we would like to balance a circuit by fine tuning only, because no area penalty would be involved. However, a balanced solution might not exist unless rough tuning is used. Usually some sections of a circuit can be balanced using only fine tuning, but rough tuning is generally required for other parts of the circuit.

Because of the nature of the problem, we use a simplified delay model for each gate during rough tuning. In

[3]Another research group [12] in wave pipelining has recently been exploring the balancing of circuit delays by instead adjusting the *RC* delays of wires. This is done by altering the placement of cells and by partially routing wires in polysilicon. Our methods work in conjunction with standard commercial place-and-route tools; we have not investigated placement algorithms.

[4]A *false path* is a long path in the circuit that actually cannot be activated by any input patterns. Since the path is never activated, it can be ignored when computing the longest path delay.

[5]However, the use of an initial fine-tuning pass reduces the ability to subsequently adjust for the difference between actual and estimated capacitances. Thus, not using an initial fine-tuning pass can sometimes result in a solution that has more area but better balancing. This effect is covered in more detail later in Section VII-D. The designer can experiment to determine whether or not to use the initial fine-tuning pass in his/her situation.

particular, the propagation delay is assumed to be equal from all inputs of each gate. Furthermore, rising and falling delays need not be represented separately because this would have little effect on any decisions to insert buffers. One propagation delay is associated to each output, representing the delay from any of the inputs to that output. This delay model is sufficient to model delays for ECL/CML without stacked structures, which is a good target technology for wave pipelining as discussed in Appendix II. The rough-tuning techniques can be extended to more complex delay models, but the resulting delay graphs have less elegant theoretical properties.

A. Circuit Representation

We represent circuits to be optimized by rough tuning using a directed acyclic graph (DAG) representation.

Nodes represent the inputs and outputs of gates. Arcs are of two types and represent I/O dependency within a gate and among gates. The lengths of the two types of arcs represent propagation delays of gates and delays of inserted elements, called *padding elements*. The first set of lengths is known, while the second set represents the unknown of the problem. Our technique determines this second set of lengths.

A rough-tuning DAG (called an *RTDAG*) is constructed from a circuit using the following steps.

1) An *output node* is associated to each output (inverting and noninverting) of each gate. An additional *input node* is associated to each gate to represent all its inputs.
2) Directed arcs (i, j) are defined as follows:
   - Type I (internal)—i is an input node and j is an output node of the same gate. Type I arcs represent gate delays.
   - Type E (external)—j is an input node which is a direct fanout of output node I. Type E arcs represent delays of padding elements that are inserted.

   The arcs are numbered from 1 to NumArcs.
3) Weights on arc $n = (i, j)$, are defined as follows:
   - Type I (internal)—Weight $D[n] \geq 0$ indicating the nominal propagation delay from any input of the gate (represented by the node i) to an output j.
   - Type E (external)—Weight $W[n] \geq 0$ indicating the amount of nominal delay to insert between the output of a gate corresponding to i and the input of a gate corresponding to j. Initially, all the $W[n]$ are zero.
4) Source and sink nodes are added to the graph. Using type E arcs, the source node 0 is connected to all primary input nodes, and the sink node N is connected to all primary output nodes.

Fig. 2 shows an example conversion of a small circuit to our graph representation. Nodes 1 and 4 represent the inputs of the two gates. Nodes 2, 3, and 5 represent the outputs. Heavy arcs are internal to one gate; regular arcs are external.

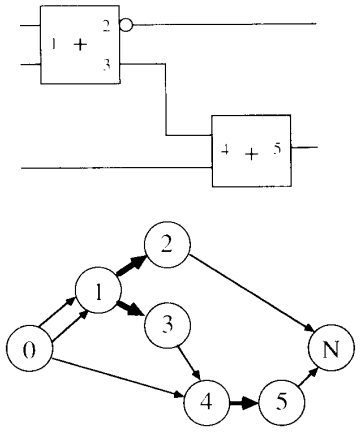


Fig. 2. An example conversion from a circuit to a DAG for rough tuning. Light type E arcs represent inserted delays. Bold type I arcs represent gate delays.

The *path length* between two nodes along a series of arcs is defined as the sum of the arc lengths, including both I- and E-type arcs.

B. Problem Formulation for Rough Tuning

The selection of a gate delay model affects the capability of a rough-tuning technique to solve the balancing problem.

Assume first that all gate delays are equal to or are integer multiples of a fixed propagation delay through a padding element. In this case, the circuit can be exactly balanced by an appropriate insertion of delay elements. Similarly, any circuit can be balanced if the propagation delay of a padding element can take any value greater than zero.

In practice, gate delays vary, and the delay of a padding element must be set within the range B_{MIN} to B_{MAX}. The finite range of delay of the padding element has two consequences. First, the minimal number of delay elements to be inserted along arc k is $\lceil (W[k]/B_{MAX}) \rceil$. Second, there is no physical implementation of a delay if $W[k] < B_{MIN}$; such delays are said to be *not implementable*. We assume that $B_{MAX}/B_{MIN} \geq 2$ (easily satisfied by gates in ECL/CML technology).

Since some arc lengths are not implementable, rough tuning might not guarantee the exact balancing of a circuit. For this reason, we state the rough-tuning problem as follows:

Definition Rough-Tuning Problem: Given a combinational logic circuit without feedback, find a set of implementable arc lengths W such that

1) all input-output paths have length $\leq D_{MAX}$,
2) the length of the shortest input-output path is maximal. □

Let ΔD_R be the length of the longest path minus the shortest path. The parameter ΔD_R measures the path length difference due to design using the simplified delay model. It does not account for differences between rise and fall delays or for gate delays that are different for each input.

An *optimal* solution is one that maximizes the length of the shortest path, thus minimizing ΔD_R. A secondary goal is to minimize added area, i.e., minimize the number of added delay elements $\Sigma \lceil (W[i]/B_{\mathrm{MAX}}) \rceil$.

C. Using Loops to Balance the Circuit

The number of paths in a polar graph may be exponential in the problem size. We therefore transform the rough-tuning problem into an equivalent one where we balance the loop lengths. This is the basis for the rough-tuning algorithm presented next.

A *loop* is defined to be a set of arcs that form a cycle in the underlying, undirected graph. Each loop in this graph has a source and a sink, which are defined to be the nodes with zero incoming and zero outgoing arcs, respectively, when considering only the arcs in the loop. The two directed paths from the loop's source to sink are called the *sides* of the loop. The length of each side is the sum of the arc lengths, including both *I*- and *E*-type arcs. If the lengths of the two sides are equal, then the loop is balanced.

Suppose we augment the directed acyclic graph by adding one arc of type *I* from source to sink with length D_{MAX}.

Theorem 4.1: The circuit is balanced with delay D_{MAX} if and only if all the loops in the graph are balanced.

Proof: If there were an input–output path with delay not equal to D_{MAX}, then an unbalanced loop could be formed using that path plus the arc with weight D_{MAX} from the graph's source to sink.

If there were an unbalanced loop, then two unequal input–output paths could always be constructed containing the two sides of the loop. □

We can define a signed addition of all the arc weights in each loop. If a walk is taken around a loop, the signed addition is the sum of the weights of all arcs, taking into account each arc's direction. Arcs pointing in the direction of the walk are added; arcs pointing in the opposing direction are subtracted. This sum is zero if the loop is balanced. If each arc's weight is a uniquely labeled variable, then each loop defines a *loop equation* stating that a signed sum of variables must be equal to zero in order to balance the loop.

Then the circuit is also balanced if and only if a spanning set of linearly independent loops is balanced [7]. A spanning set of linearly independent loops is simply a set of loops corresponding to a maximal set of loop equations that are algebraically linearly independent. The loop equation of any other loop can be generated by linearly combining equations of the loops in a spanning set.

Since the number of linearly independent loops is linear in the size of the problem, this result enables us to verify efficiently whether a circuit is balanced or not.

One type of spanning set is a set of *fundamental loops*. As stated in [7], a spanning set of fundamental loops can be constructed as follows.

1) Construct a spanning tree in the DAG beginning from the source.
2) Let A represent the set of *links*, i.e., arcs that are not in the tree.
3) Let L be the (initially empty) set of loops.
4) Add one link from A to the tree. This defines exactly one loop called a *fundamental loop* that is added to L. We say that the link *closes* the loop.
5) Repeat for all the arcs in A.

The loops in L are linearly independent, since each link is in exactly one loop.

The choice of a spanning tree is important because it affects some properties of the resulting fundamental loops. A *longest-path* spanning tree is a spanning tree rooted at the source such that the tree contains a longest path from the source to each node.

Proposition 4.1: In any fundamental loop constructed from a longest-path spanning tree, the link is always on the side of smaller or equal length.

Fig. 3 is an example of constructing loops based on a spanning tree. The DAG (copied from Fig. 2) has been assigned arc lengths indicating delay. A longest path spanning tree has been built from the source and is shown in heavy arcs. Each one of the links forms a loop when added to the tree. For instance adding the arc (0, 4) forms the loop with (0, 4) on one side and [(0, 1), (1, 3), (3, 4)] on the other side. If each such loop has the same length on both sides, then the circuit is balanced.

Theorem 4.2: Given a DAG representing a circuit and a longest path spanning tree built from the source, every link is a type *E* (external) arc.

Proof: The head of each arc of type *I* is at an output node. The head of each arc of type *E* is at an input node. (The source is considered an "output" node, and the sink is an "input" node.) Since each output node has only one incoming arc, a loop formed using a spanning tree from the source cannot be closed at an output node. Therefore, all loops must be closed at input nodes, and only type *E* arcs have input nodes as heads. Since each link closes a loop, each link must be type *E*. □

From Proposition 4.1 and Theorem 4.2, it follows that every fundamental loop has an adjustable (type *E*) arc on the side with smaller or equal total length.

D. An Algorithm for Rough Tuning

Based on the above idea of balancing a spanning set of loops, the rough-tuning procedure balances a circuit by inserting delay along type *E* arcs as necessary. The rough-tuning algorithm has three major steps: constructing a balanced solution; changing the arc lengths to minimize the added padding elements; and implementing the arc lengths using padding elements. The second step may be skipped if non area-optimal, but balanced, circuits are sought.

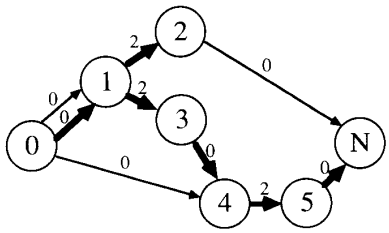


Fig. 3. An example of constructing loops based on a spanning tree. The longest path tree is shown using bold arcs.

Rough-Tuning Algorithm

1) Construct an RTDAG to represent the circuit. Add one type *I* arc of length D_{MAX} from the source to the sink.
2) Build a longest path spanning tree T from the source.
3) For each link i, insert the proper arc length $W[i]$ to balance the corresponding fundamental loop.
4) Apply the repadding algorithm (described in detail later) to minimize the number of delay elements.
5) Implement the delay lengths by inserting the delay elements.

Since the fundamental loops are linearly independent (each link is in exactly one loop), all fundamental loops can be balanced independently from each other.

At step 5, the delay lengths are implemented as follows. For a length $W[i] \geq B_{MIN}$, insert $\lceil W[i]/B_{MAX} \rceil$ delay elements on arc i. (Recall that B_{MIN} and B_{MAX} are the minimum and the maximum delay of a padding element, respectively).

Any weight $W[i] \geq B_{MIN}$ can be implemented exactly by adjusting the gate drives for the inserted buffers appropriately. For a link having $n \geq 2$ buffers, the first $n - 2$ are set to delay of B_{MAX} each, and the last two are set appropriately to remove the remaining imbalance in the loop.

For any lengths that are smaller than B_{MIN}, a heuristic method must be used to minimize the imbalance. One method is to simply ignore any lengths smaller than B_{MIN}. In this case, ΔD_R is bounded from above by $\Delta D_R < nB_{MIN}$ where every input–output path has at most n type E arcs that have length $< B_{MIN}$.

If the constraint that the longest path must be $\leq D_{MAX}$ can be replaced by $\leq D_{MAX} + n * B_{MIN}/2$, a second heuristic can be used. In this method, lengths greater than $B_{MIN}/2$ are implemented with a single delay element. This might reduce the imbalance better than the first heuristic since ΔD_R is often less than $nB_{MIN}/2$, even though the bound is still $\Delta D_R < nB_{MIN}$. It is necessary to relax the longest path constraint because if D_{MAX} is as small as possible (i.e., equal to the critical path with all gates at maximum power in the original circuit), then the new critical path t_P could exceed D_{MAX} by up to $nB_{MIN}/2$.

Note that ΔD_R represents an upper bound on the mismatch of the source–sink path delays. Fine tuning can further reduce this mismatch, and in some cases, it may perfectly balance a circuit. However, the possible presence of non-implementable delay lengths prevents a guarantee that perfect balance $\Delta D_R = 0$ can be achieved.

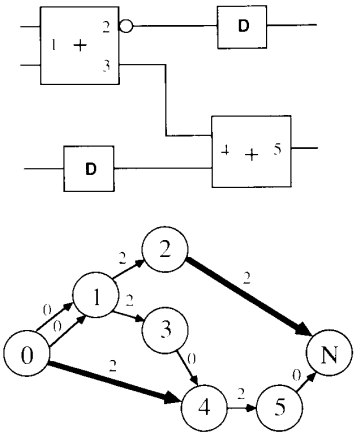


Fig. 4. An example of an RTDAG that has been balanced and the corresponding circuit with delay elements D inserted. Padded arcs are shown in bold.

Fig. 4 shows a simple example of how the rough-tuning algorithm would insert delay elements. The logic gates in the circuit have a delay of 2. By inserting a delay of 2 on each of the two bold external arcs, the graph can be balanced. Supposing that $B_{MIN} = 1$ and $B_{MAX} = 3$, the corresponding circuit is then padded with a single delay element for each of the bold arcs as shown in the diagram.

1) Achieving a minimal-area solution:

Delay lengths can sometimes be shifted from one portion of a circuit to another. For instance, if delay elements are to be placed on all the wires on one side of a gate (either at the inputs or outputs), then some delay can be shifted to the other side. This can reduce the total number of delay elements when the number of wires is fewer on the other side.

This section presents a method for systematically shifting delay lengths to minimize the number of delay elements. We call this method *repadding*. It is derived from a method described in [17] for minimizing the number of register bits in sequential circuits by moving register boundaries.

The problem formulation for repadding can use a simplified graph model because the information about the delays of the original gates is no longer necessary. The required information is the circuit topology and the position and number of the padding elements. Therefore, a simplified graph is obtained by contracting all arcs of type *I* (Internal). An example of this is shown in Fig. 5. Nodes 1 and 2 correspond to the two gates. An external arc exists for each connection in the circuit.

The delay lengths are given to the repadding algorithm as a vector of non-negative real numbers. However, a shifting should be performed only if the result would save delay buffers. Shifting small amounts of delay might not be useful and could even divide padding delay between

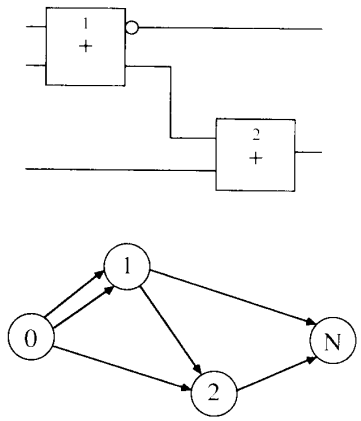


Fig. 5. An example conversion from a circuit to a simplified DAG for repadding. A simplified graph uses one node per gate and only type E arcs.

gate inputs and outputs in a way that increases the number of buffers.

To avoid this problem, we conservatively allow the repadding algorithm only to shift delays in units of B_{MAX}. If the repadding algorithm decides to shift a delay of B_{MAX}, the number of buffers will definitely be decreased.

Therefore, new arc lengths are defined as the pair $(W_X[(i,j)], W_Y[(i,j)])$ for each arc (i,j):

$$W_X[(i,j)] = \lfloor W[(i,j)]/B_{\mathrm{MAX}} \rfloor$$

$$W_Y[(i,j)] = W[(i,j)]/B_{\mathrm{MAX}} - W_X[(i,j)].$$

Shifting padding elements from the outputs to the inputs of gate i is represented by a signed integer variable $R[i]$ associated to each node i. Given a vector R representing repadding, the corresponding lengths on the arcs are

$$W'_X[(i,j)] = W_X[(i,j)] + R[j] - R[i].$$

The formula $R[i]$*(*indegree*(i) − *outdegree*(i)) is the change in the total number of padding elements due to a repadding by $R[i]$ at node i. (*Indegree*(i) and *outdegree*(i) are the number of incoming and outgoing arcs, respectively, at node i.)

The repadding problem can be formulated as a linear program corresponding to a minimum-cost network flow problem:

Find a vector R to minimize $\Sigma R[i]$(*indegree*$[i]$ − *outdegree*$[i]$) subject to:

$$R[i] - R[j] \leq W_X[(i,j)] \text{ for all arcs } (i,j).$$

$$R[0] = R[N] = 0 \text{ for source 0 and sink } N.$$

The constraints guarantee that the solution does not have negative arc lengths [17].

The quantity *indegree*(v) − *outdegree*(v) measures the reduction in $\Sigma W_X[i]$ if one unit of delay is shifted from the input arcs of node v to the outputs. Therefore, the linear program also minimizes the desired function $\Sigma W_X[i]$.

The minimum-cost flow problem can be solved by either the matching algorithm of Edmonds and Karp [16] or by the Simplex algorithm. We have chosen the second route to leverage the use of powerful linear problem solvers [21]. Therefore, the steps for achieving an optimal solution are as follows:

Repadding Algorithm

1) Construct the simplified graph model and the corresponding linear program.
2) Compute R by solving the linear program.
3) Compute the new arc lengths using
$$W'_X[(i,j)] = W_X[(i,j)] + R[j] - R[i].$$
4) Recompute the full arc lengths using
$$W'[i] = W'_X[i] * B_{\mathrm{MAX}} + W_Y[i] * B_{\mathrm{MAX}}.$$

Fig. 6 shows a simple example of repadding. The circuit and simplified RTDAG on the top are prior to repadding. Suppose that $B_{\mathrm{MAX}} = 4$, so that the delays of 4 on the two arcs from node 2 to the sink N correspond to [1, 0] when split into the vector $[W_X, W_Y]$. The solution of the repadding linear program shifts the delay to the arc from node 1 to 2. After converting back to ordinary delay units, the arc from node 1 to 2 now has a delay of 4. The revised circuit is then as shown on the bottom.

2) Remarks on Repadding:

- It can be shown that every basic optimal solution is an integral vector [17], [22]. This ensures that only entire units of delay can be shifted.
- When an output has multiple fanouts and more than one fanout arc has a positive delay length, the physical implementation can share delay elements, rather than implementing multiple delay chains. It is possible to use a linear program to minimize the total number of padding elements including the effects of sharing padding elements. The details are not reported here, but a similar formulation has been reported [17].

E. Properties of Rough Tuning

The rough-tuning algorithm has the following properties:

1) For a circuit whose gate delays are integer multiples of the padding element delay, the rough-tuning algorithm is guaranteed to balance the circuit with a minimum number of padding elements.

2) For circuits using delay elements that have an adjustable delay from B_{MIN} to B_{MAX}, the rough-tuning algorithm is guaranteed to balance a circuit to within $\Delta D_R < nB_{\mathrm{MIN}}$, where every input–output path has at most n type E arcs with length less than B_{MIN}. (The inequality is sharp since nB_{MIN} would result only if the n arcs had delay equal to B_{MIN}.)

3) For circuits where all the padding delays are implementable (i.e., either 0 or $\geq B_{\mathrm{MIN}}$), rough tuning balances the circuit exactly.

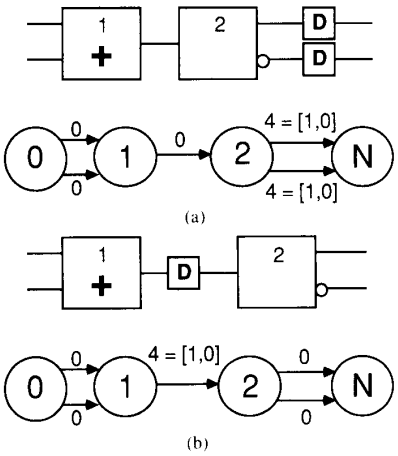


Fig. 6. An example of repadding. (a) The circuit and DAG prior to repadding. (b) The revised circuit and DAG after repadding.

4) For circuits using adjustable delay elements, the method locally minimizes the number of added delay elements. The repadding algorithm is a very good heuristic, but may not guarantee a globally optimum solution because it finds a global minimum for the function $\Sigma W_X[i]$ rather than the true cost function $\Sigma \lceil W[i]/B_{\mathrm{MAX}} \rceil$. The division of the arc lengths by B_{MAX}, while ensuring that the repadding can only improve the circuit, restricts the search space, thus preventing a guarantee of global optimality.

5) The theoretical computational complexity of the rough-tuning algorithm is dominated by the solution method for the minimum-cost flow problem. In practice, the Simplex algorithm for solving the linear program is superlinear with respect to the problem size.

V. Fine Tuning

The fine-tuning algorithm is presented as follows. First, we describe a model of the circuit as a directed acyclic graph. Then we formulate two related problems: fine tuning and power minimization. We show that a solution to the fine-tuning problem can be derived from a solution to power minimization. Then we show that the power-minimization problem can be solved by a linear program. Finally, the effects of uncertain delays and other practical considerations are presented.

A. Circuit Modeling Using a Fine-Tuning Directed Acyclic Graph

We model the circuit using a polar weighted directed acyclic graph. This graph uses a more complex delay model than that used by rough tuning. In particular, rising and falling delays are distinct, and delays from each gate input are independently specifiable. A fine-tuning DAG (FTDAG) is constructed from a circuit using the following steps:

1) Each node i is in one-to-one correspondence with a gate output i. (Note that in ECL/CML, there are two outputs per gate.)
2) A directed arc (i, j) exists if the gate corresponding to j takes i as an input. We number all the arcs from 1 to *NumArcs*.
3) Each arc (i, j) indicates gate delay by a tuple length (rising delay R, falling delay F, unate flag U) where
   - R is the delay of j's gate from the tail i to cause a rising transition at j.
   - F is the delay of j's gate from the tail i to cause a falling transition at j.
   - U indicates whether j's gate output is positive unate, negative unate, or not unate, with respect to the tail input. The *unateness* property is defined as follows for each input/output combination of a logic gate:

     In a *positive unate* input/output combination, rising transitions at the input can only cause rising transitions at the output. Falling transitions at the input only cause falling transitions at the output. All input/output combinations in AND and OR gates are positive unate.

     In a *negative unate* input/output combination, rising transitions at the input can only cause falling transitions at the output. Falling transitions at the input only cause rising transitions at the output. All combinations in NAND and NOR gates are negative unate.

     In a *nonunate* input/output combination, the transition direction at the input does not determine the transition direction at the output. All combinations in XOR gates are nonunate.

     Each input/output combination should be treated individually because gates can have some combinations that are positive unate and some that are negative unate. For instance, ECL gates typically have both true and complementary outputs.
4) A source node 0 is connected to all primary input nodes, and a sink node N is connected to all primary output nodes.

An example conversion from a circuit to a corresponding DAG is shown in Fig. 7. In the example, three global inputs go to outputs 1 and 2 through an OR/NOR gate. Three arcs are drawn from the source node to each node 1 and 2 representing the delays through that OR/NOR gate. Similarly, output 2 goes through an OR gate to output 3. The corresponding arc goes from 2 to 3 and represents the OR gate's delay.

Path Delays: A source–sink path is a sequence of nodes and directed arcs leading from the source to the sink. *Path delay* denotes a pair (X, Y) where

X delay to rising transition at sink
Y delay to falling transition at sink.

The path delay is the sum of the arc lengths, taking into account whether the corresponding gates are positive unate, negative unate, or non-unate. The corresponding arcs for each gate inherit the same unateness property as the gate. For a path consisting of positive-unate arcs

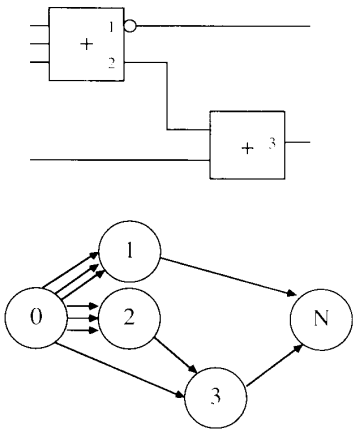


Fig. 7. DAG for fine tuning. Each global input or gate output is connected to all of the nodes in its fan-out.

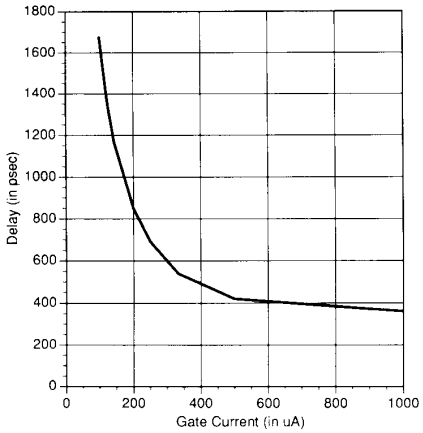


Fig. 8. Monotonically decreasing, convex delay versus power curve plotted for a typical CML gate.

$1 \cdots n$, the path delay is

$$\left(\sum_{i=1}^{n} R[i], \sum_{i=1}^{n} F[i]\right).$$

Using a series of arcs for negative unate gates, the path delay equals

$$(\cdots + R[n-2] + F[n-1] + R[n],$$
$$\cdots + F[n-2] + R[n-1] + F[n]).$$

For paths with some non-unate arcs, the idea of a unique path delay cannot be applied. However, maximum and minimum delays for a path can be defined. For instance, suppose that the head of arc i represents the output of a nonunate gate. The maximum rising path delay up to arc i is defined recursively as

$$d_{\text{MAX-RISE}}[i] = max(d_{\text{MAX-RISE}}[i-1] + R[i], d_{\text{MAX-FALL}}[i-1] + R[i]).$$

This $d_{\text{MAX-RISE}}[i]$ may be used for any maximum delay computations with arcs beyond arc i.

B. Problem Formulation for Fine Tuning and Power Minimization

Let us now define the fine-tuning problem and the necessary information to formulate it.

1) Definition: Combinational Logic Circuit with Fine-Tuning Information: A *combinational logic circuit with fine-tuning information* is a circuit represented by a DAG with the following delay model:

1) Drive (power or current) $P[i]$ for each gate i.
2) Load capacitance $L[i]$ for each gate output i.
3) Power limits $P_{\text{MIN}}[i]$ and $P_{\text{MAX}}[i]$ for each gate i.
4) Two delay functions for each arc (i, j) where node j is an output of gate k:
   $f[i, j, 0](power\ P[k], load\ L[j])$ = maximum time from a transition on tail i to a falling transition at head j.
   $f[i, j, 1](power\ P[k], load\ L[j])$ = maximum time from a transition on tail i to a rising transition at head j.
   We assume that these functions are convex and monotonically decreasing with respect to power (Fig. 8). Technologies nearly always satisfy these assumptions over reasonable power bounds P_{MIN} and P_{MAX}.
5) A user-specified delay D_{MAX}.
6) $D[i, 0]$ is the maximum delay from source to a falling transition at node i.
7) $D[i, 1]$ is the maximum delay from source to a rising transition at node i.

2) Definition: Fine-Tuning Problem: The following is the *fine-tuning problem*.

Given a combinational logic circuit with fine-tuning information, find a vector of gate drives P such that

- all input–output paths have delay $\leq D_{\text{MAX}}$
- the length of the shortest path is maximal. □

A secondary goal is to minimize total power consumption $\Sigma_{i=1}^{n} P[i]$ where n is the number of gates.

Let ΔD be the difference between the lengths of the longest and shortest I/O paths. Ideally, the difference ΔD should be zero. The finite range of tunability of the individual gates and the asymmetry in rise and fall delays may prevent the existence of an exactly balanced solution. We, therefore, consider first fine tuning under some simplifying assumptions and show exact balancing techniques. Then we show how the complete fine-tuning problem can be solved with a guaranteed bound on ΔD.

When gates have different rising and falling delays, a solution with $\Delta D = 0$ might not even exist. For instance, consider a small circuit of one gate with different delays for rising and falling output transitions. By definition, ΔD is not zero but instead equal to the absolute difference between rising and falling delays. We therefore postulate a restriction of the fine-tuning problem, called a symmetric fine-tuning problem.

We develop a method for finding a solution to the symmetric fine-tuning problem. This method also solves the full fine-tuning problem with a guaranteed bound on ΔD. If ΔD is small with respect to the other components of t_{CP}, then an efficient wave-pipelined implementation can be constructed.

3) Definition: Symmetric Fine-Tuning Model: If the gates in a fine-tuning problem all have rising delays equal to falling delays, i.e.,

$$f[i, j, 0](P, L) = f[i, j, 1](P, L) \ \forall i, j, P, \text{and } L$$

then it is a *symmetric fine-tuning problem.* □

The fine-tuning problem is closely related to the *power minimization problem* which is formulated as follows:

4) Definition: Power Minimization Problem: Given a combinational logic circuit with fine tuning information, find a vector of gate drives P such that 1) all input–output paths have delay $\leq D_{MAX}$ and 2) the total power consumption $\Sigma_{i=1}^{n} P[i]$ is minimal.

C. Solution Method

We now explain how to solve the power minimization problem. Later, we show when and why the solution to power minimization is also the solution to the fine-tuning problem.

The power minimization problem can be efficiently solved as a linear program by approximating the nonlinear delay functions $f(P, L)$ by piecewise-linear functions. This is possible because the functions are assumed convex. In practice, the delay functions are roughly hyperbolic and can be well approximated with a few line segments. Without this simplification, we would have a nonlinear program whose solution would be costly for large circuits. The following formulation can also be applied to the full fine-tuning problem, since it is formulated in terms of separate rising and falling delays.

This linear programming formulation is related to a nonlinear program used by Marple [20] for optimizing MOS circuit power under timing constraints.

1) Linear Program: A weighted power minimization problem can be formulated as the following linear program:

$$\text{Minimize } P = \sum_{i=1}^{n-b} P[i] + \beta * \sum_{i=n-b+1}^{n} P[i]$$

where gates $1 \cdots (n-b)$ are the original gates, $(n-b+1) \cdots (n)$ are the delay buffers, and $0 \leq \beta \leq 1$ is a weighting factor discussed later. The total number of gates is n, and the number of buffers is b. The following constraints apply.

1) *Power constraints*

$$P_{MIN}[i] \leq P[i] \leq P_{MAX}[i] \text{ for each gate } i.$$

2) *Source boundary constraints*

$$D[0, 0] = D[0, 1] = 0.$$

3) *Sink delay constraints*

$$D[N, 0] \leq D_{MAX}$$

$$D[N, 1] \leq D_{MAX}.$$

4) *Graph delay constraints*

We build delay constraints from each arc (i, j) depending on the arc's unate flag U. Let x be the gate corresponding to output j.

If U is positive unate:

$$D[i, 1] + f[i, j, 1](P[x], L[j]) \leq D[j, 1]$$

$$D[i, 0] + f[i, j, 0](P[x], L[j]) \leq D[j, 0].$$

If U is negative unate:

$$D[i, 1] + f[i, j, 0](P[x], L[j]) \leq D[j, 0]$$

$$D[i, 0] + f[i, j, 1](P[x], L[j]) \leq D[j, 1].$$

If U is not unate:

$$D[i, 1] + f[i, j, 1](P[x], L[j]) \leq D[j, 1]$$

$$D[i, 1] + f[i, j, 0](P[x], L[j]) \leq D[j, 0]$$

$$D[i, 0] + f[i, j, 0](P[x], L[j]) \leq D[j, 0]$$

$$D[i, 0] + f[i, j, 1](P[x], L[j]) \leq D[j, 1].$$

By replacing each nonlinear function $f(P, L)$ (where L is a constant) by a piecewise-linear function, these constraints can be made linear. One linear constraint is superimposed on each line segment of the piecewise-linear function. Since the function is convex, each such constraint is binding only along its line segment. This turns each constraint above into a set of linear constraints, each using P only as a linear variable.

D. Equivalence of Fine-Tuning and Power-Minimization Problems

At this point, we first explain how and when power minimization solves the symmetric fine-tuning problem. We address the full fine-tuning problem later.

Let $\hat{P}$ be the optimal solution to the power minimization problem. We say that an inequality constraint is *active* at the optimal solution $\hat{P}$ if the two sides of the constraint are actually equal, i.e., the constraint is at its limit. We define *delay constraints* to be those stated in items 3 and 4 of the weighted power minimization linear program. Since all delays are equal for rising versus falling, the delay variables for rising and falling transitions are also equal, i.e, $D[i, 1] = D[i, 0] \ \forall i$. Therefore, each arc either has all or none of its corresponding delay constraints active.

Theorem 5.1: Any source–sink path where all arcs have active delay constraints has length D_{MAX}.

Proof: Let $D[i] = D[i, 0] = D[i, 1]$ denote the delay variable at each node i. Consider the head h and tail t of each arc along the path. Since each arc has active constraints, the delay variable $D[h]$ exactly equals the delay variable $D[t]$ plus the delay of the arc. The delay variables at the sink node, therefore, equal the sum of the arc

lengths. Since the final arc leading into the sink has active constraints, the final delay variable $D[N]$ equals D_{MAX}. Therefore, the sum of the arc lengths also equals D_{MAX}.

This leads to the following theorem:

Theorem 5.2: Given a combinational logic circuit with a symmetric fine-tuning model, if $\hat{P}$ is a solution to the power minimization linear program such that all delay constraints are active, then $\hat{P}$ is also a solution to the symmetric fine-tuning problem for the same circuit and D_{MAX} with $\Delta D = 0$.

Proof: Every arc on each input–output (I/O) path has an active delay constraint. Since Theorem 5.1 applies to each I/O path, all such paths have length D_{MAX}. Therefore, the difference ΔD between the longest and shortest I/O paths is zero.

Unfortunately, a solution to power minimization might not satisfy the assumptions of Theorem 5.2, because some power constraints might become active or because there might not be enough independent "tuning points" in the circuit.

It is interesting to note that both maximum and minimum power bounds on buffers can prevent the guaranteed existence of a fully balanced solution. Indeed, the lower bound on power P_{MIN} (corresponding to a finite maximum delay B_{MAX}) might be active for some gate drives, possibly preventing the corresponding delay constraints from becoming active. The presence of a finite upper bound P_{MAX} on power (corresponding to a minimum delay $B_{MIN} > 0$) might prevent a circuit from being fully balanced by rough tuning. If any delay greater than 0 could be inserted by rough tuning to balance each independent loop, then the circuit would have enough tuning points to be subsequently balanced by fine tuning.

Removing the upper bound on power is theoretically sufficient to allow rough tuning to balance the circuit at the expense of some added area. However, for practical circuits bounds on power are mandatory.

In typical technologies, *active* delay elements have a certain minimum delay $B_{MIN} > 0$, so the rough-tuning algorithm balances the circuit to a certain ΔD_R which might be greater than zero. Therefore, in the following discussions, we assume that ΔD_R might be a positive number.

In Section VI, we return to discussing the imbalance ΔD which may remain after the entire tuning process. Next, we derive straightforward bounds on ΔD for the cases when delays have an uncertainty range or when rising and falling delays are different.

E. Properties of the Algorithm When Delays are Uncertain

Due to process variations and temperature-induced effects within one chip and data-dependent delays, a gate's speed is not perfectly controllable relative to other gates within a chip.

Theorem 5.3: Balancing using Uncertain Delays: Suppose each arc (i, j) has a delay $d[i, j]$ that has maximum value $d_{MAX}[i, j]$. Then a parameter α is used to measure speed variations that do not track within a chip. Suppose every $d[i, j]$ can be bounded using

$$(1 - \alpha)d_{MAX}[i, j] \le d[i, j] \le d_{MAX}[i, j],$$

$$\text{for some } \alpha, \quad 0 \le \alpha \le 1.$$

If the tuning procedure is performed using the $d_{MAX}[i, j]$'s as the delays and the solution has a bound ΔD, then a similar bound ΔD_2 can be defined to include the effects of uncertain delays

$$\Delta D_2 \le \alpha(D_{MAX} - \Delta D) + \Delta D.$$

Proof: If the $d_{MAX}[i, j]$'s were the actual delays, then the shortest path would have length $D_{MAX} - \Delta D$. In this case, the actual delay of each gate along the shortest path is at least $(1 - \alpha)d_{MAX}[i, j]$. Therefore, the true shortest path is at least $(1 - \alpha)(D_{MAX} - \Delta D)$ in length. Subtracting this from the longest path D_{MAX} yields the above constraint on ΔD_2. □

F. Solving the Full Fine Tuning Problem

When rising delays do not equal falling delays, the power minimization method is a reasonable heuristic for a difficult problem. Since each gate has only one power parameter which controls a rising and falling delay per output, solutions with nearly balanced delays may not even exist in some cases.

The following procedure defines the use of power minimization to solve the full fine-tuning problem.

1) Given a combinational logic circuit with fine-tuning information, define

- $d_{MAXRF}[i, j]$
  = max(rising delay, falling delay) of arc (i, j)
- $d_{MINRF}[i, j]$
  = min(rising delay, falling delay) of arc (i, j).

These delays are defined from the delay functions $f[i, j, 0](P[x], L[j])$ and $f[i, j, 1](P[x], L[j])$ once the corresponding gate drive $P[x]$ and output capacitance $L[j]$ are known.

2) Assume that each arc (i, j) has delay $d_{MINRF}[i, j]$ bounded by

$$(1 - \alpha)d_{MAXRF}[i, j] \le d_{MINRF}[i, j] \le d_{MAXRF}[i, j]$$

$$\text{for some } \alpha, \quad 0 \le \alpha \le 1.$$

3) Perform the tuning procedure on this circuit. During rough tuning, the d_{MAXRF}'s are used as the delays. During fine tuning, power minimization is performed on the circuit using both the rising and falling delays.

Theorem 5.4: Full Fine Tuning: If $\hat{P}$ is the solution to the power minimization problem in the final fine-tuning pass of the above procedure, then $\hat{P}$ also solves the full fine-tuning problem with

$$\Delta D \le \alpha(D_{MAX} - \Delta D_S) + \Delta D_S$$

where ΔD_S is the remaining imbalance which would result from solving the corresponding symmetric fine-tuning problem using the $d_{MAX\,RF}[i, j]$'s for both rising and falling delays.

Proof: Let us examine the solution of the symmetric fine-tuning problem which uses the $d_{MAX\,RF}[i, j]$'s for both rising and falling delays. Each path can keep the same length or become shorter if we substitute the actual nonsymmetric rise–fall delays and keep the power settings constant. Therefore, all gates in the full fine-tuning solution should have the same or lower power than in the symmetric fine-tuning solution.[6]

In the worst case, we assume that no gates are set at any lower power. The shortest path in the symmetric fine-tuning problem has length $D_{MAX} - \Delta D_S$. In the worst case, this path delay is composed entirely of the larger of the rising/falling delays on each arc, i.e., only $d_{MAX\,RF}[i, j]$'s. The shortest path in the full fine-tuning solution must therefore be at least $(1 - \alpha)(D_{MAX} - \Delta D_S)$ in length. Subtracting this from the longest path D_{MAX} yields the stated limit on ΔD. □

Remarks:

- If negative unate logic is used, then the rise–fall differences tend to cancel out over a number of gate levels, so the actual ΔD can be much lower than the bound indicates.
- ECL emitter-followers have more discrepancy between rising and falling delays at low power levels. Setting the minimum power P_{MIN} relatively high for emitter-followers can, therefore, help to reduce ΔD.

VI. The Overall Tuning Procedure

Consider the overall tuning procedure described earlier. The first step is an optional initial fine-tuning pass which sets the gate drives using the linear program described above with estimated capacitances. The second step is rough tuning, which yields an estimate of the remaining imbalance ΔD_R in the circuit based on the arcs which have unimplementable lengths. Then the circuit is laid out, and actual capacitances are extracted. A final fine-tuning pass adjusts gate drives, taking into account the extracted capacitances.

Let ΔD_A denote the worst-case difference in path delays using the fine-tuning gate delay model to analyze the circuit after rough tuning. The calculations are performed using the estimated capacitances used in rough tuning. Any difference between ΔD_A and ΔD_R solely reflects the use of the fine-tuning rather than rough-tuning, delay model.

Ideally, we would be able to guarantee that ΔD after the final fine tuning pass would always be less than ΔD_A after rough tuning. Unfortunately, for the following reasons it is difficult to guarantee tight bounds on ΔD under general circumstances:

- Since the inserted buffers might cause some formerly shorter paths to have maximal length D_{MAX}, the final fine-tuning solution might be different from the initial fine-tuning solution created prior to rough tuning. In the worst scenario, the revised solution could have new short paths which cause the imbalance ΔD to be greater than ΔD_A.
- The actual capacitances used in the final fine-tuning pass may differ significantly from the estimated capacitances used in the initial fine-tuning pass and rough tuning. In the worst case, some gates might not have a sufficient tuning range to balance the circuit with the changed capacitances.

Under these circumstances, we can consider two alternative power minimization algorithms. The first is normal power minimization, i.e., with the weight $\beta = 1$. This method performs well heuristically.

The second method is to use a weighted sum of the powers as the goal function to reduce the likelihood of new short paths. In this case, β is set to a small positive number much less than 1. This causes the linear program to prioritize so that it minimizes the power of all regular gates first. The delay buffer powers will be minimized afterwards only to "fill in" the short paths. The delay buffer powers will each be reduced until a delay or power constraint becomes active. For sufficiently small β, no constraints that were active before rough tuning will be deactivated during the final fine-tuning pass, due to the addition of buffers. In this sense, the second method is an improvement over the first method. In practice, β should be small, e.g., 0.001, but not so small as to cause numerical problems. If actual capacitances could be assumed to be equal to the original estimates, this method would guarantee that ΔD is less than or equal to ΔD_A.

When capacitances change substantially, the results are heuristic. One method that can improve the result is to leave power margins during the first fine-tuning pass by restricting the gates to a middle subset of their power range. This leaves some margin that can help compensate for capacitances that are different from the estimates. This is discussed in more detail in Section VII-D.

Although a precise bound on ΔD is not possible when capacitances change, in practice, the tuning procedure performs very well. This will shown by the small ΔD achieved in the demonstration chip described in Section VII-C despite actual capacitances which do in fact vary substantially from the initial estimates.

One additional note is that ECL/CML gates are actually tunable in small discrete steps rather than continuously because resistors have integral dimensions in VLSI. A general property of linear program solutions is that the solution lies at the intersection of linear programming constraints where possible. Therefore, during the construction of the fine tuning linear program the piecewise-linear approximations of each delay versus power curve should be done so that the breakpoints correspond to actually implementable resistor values. Many of the gate drives are then set to implementable values exactly. The remaining rounding error should be small compared to

[6]This claim requires that the power versus delay curves are convex and monotonically decreasing, as assumed earlier.

other components of Δt_P. After rounding, the maximum and minimum paths can be computed again to get a revised Δt_P.

VII. Experiments with Wave-Pipelined Designs

A. Algorithm Implementation

The rough- and fine-tuning graph algorithms have been implemented in about 6500 lines of C code, excluding the netlist parser and linear program solver.

The fine-tuning program reads a circuit and gate library description in a netlist language called Stanford Logic Interchange Format (SLIF). As described earlier, fine tuning produces a linear program for power minimization. This is then input to a standard linear program solver called MINOS [21] that uses the Simplex method to solve for the vector P. The fine-tuning program then produces a modified SLIF circuit file with all the gate drives set.

The rough-tuning program reads the same circuit file, performs the rough-tuning algorithm, and writes a modified circuit file with buffers inserted. As an optional intermediate step, it produces a linear program for repadding to be solved using MINOS. The solution is read to create a rough-tuned circuit with minimal added area.

B. Example Applications

We now present the results of applying this method to four example circuits.

1) Adder Circuits: The first two circuits are a 4-bit carry-lookahead adder slice and a 16-bit carry-lookahead adder using 4-bit slices. To demonstrate the benefits of rough tuning alone, we assume fixed delays of 1 for the gates (corresponding to gates set at high power) so that fine tuning becomes unimportant. The padding elements have adjustable delays between 1 and 3 units, and each padding element is assumed to take 3/4 of the area of an average gate. As shown in Table I, rough tuning is required to balance the circuits to a small ΔD.

2) Multiplier Circuits: The second two circuits are the partial-products generator plus carry-save adder sections of 4 × 4 and 8 × 8-bit combinational CML multipliers. The partial-products generator is an array of AND gates, and the carry-save adder section is a Wallace tree using 3-2 counters without Booth encoding [23]. Each 3-2 counter is implemented using two levels of OR/NOR gating, which therefore requires the use of each input and output in both true and complement.

Approximate CML gate delay and capacitance models were developed based on simulations of ECL circuits in HS3.5, a 1.5-μm Bi-CMOS technology from Signetics. The shortest paths take one nominal-power gate delay (0.4 ns).

The results shown in Table II were achieved using an initial fine-tuning pass followed by a rough-tuning pass.[7]

[7]Table II is a revised version of a table that appeared in [24] and [25]. It contains corrections to the number of buffers added by rough tuning on Mult4×4 and Mult8×8. The earlier numbers were unfortunately too high due to a programming error.

TABLE I
Example Rough Tuning Results

Circuit	Add4[1]	Add16[1]
Size	30	134
Padding Elements	11	86
Depth (gate levels)	4	8
Estimated Increase In Area	29.1%	48.1%
D_{MAX}	4	8
ΔD (before)	75%	87.5%
ΔD (after)	0%	0%
Total Runtime (uVAX 3200)	<0.01 h	0.01 h
Rough-Tuning Run Time	<0.01 h	0.01 h

[1]The run-times for Add4 and Add16 are estimated.

TABLE II
Example Combined Tuning Results

Circuit	Mult4×4	Mult8×8
Size	90	498
Padding Elements	10	49
Depth (gate levels)	5	9
Estimated Increase In Area	8.3%	7.4%
D_{MAX}	2.0 ns	4.0 ns
ΔD (before)	80%	90%
ΔD (after)	6.6%	12.2%
Power (after)	90 mW	377 mW
Total Runtime (DEC 5000/125)	0.049 h	2.911 h
Rough-Tuning Run Time	0.005 h	0.025 h

The final fine-tuning pass was not performed since the circuits were not laid out.

The runtime of the combined tuning procedure is dominated by fine tuning which solves a large linear program to set the gate currents. The rough tuning procedure takes only a tiny portion of the runtime because it solves a smaller linear program.

In most technologies, including CML, rising delays are not exactly equal to falling delays. This causes some difference in path lengths that is difficult to completely remove. In this case, about 5%–10% of D can be added to the given ΔD to include the difference between rising and falling delays in CML.

The power consumption is the global minimum for the specified delay D_{MAX}. Increasing the allowed delay would reduce the power. The run time is substantial for Mult8×8 but may be reduced for large circuits using hierarchical design techniques and possibly by using a solution method other than the Simplex algorithm.

The increase in area is substantially smaller for the Wallace tree circuits than the adder circuits due to the basic regularity of Wallace trees plus the use of tunable gates and delay buffers.

Rough and fine tuning make these circuits wave-pipelineable. In all four examples, ΔD is changed from almost D_{MAX} to nearly zero.

C. A 63-Bit Population Counter

In order to fully test the wave-pipelining concept, we have designed a demonstration chip. The physical design

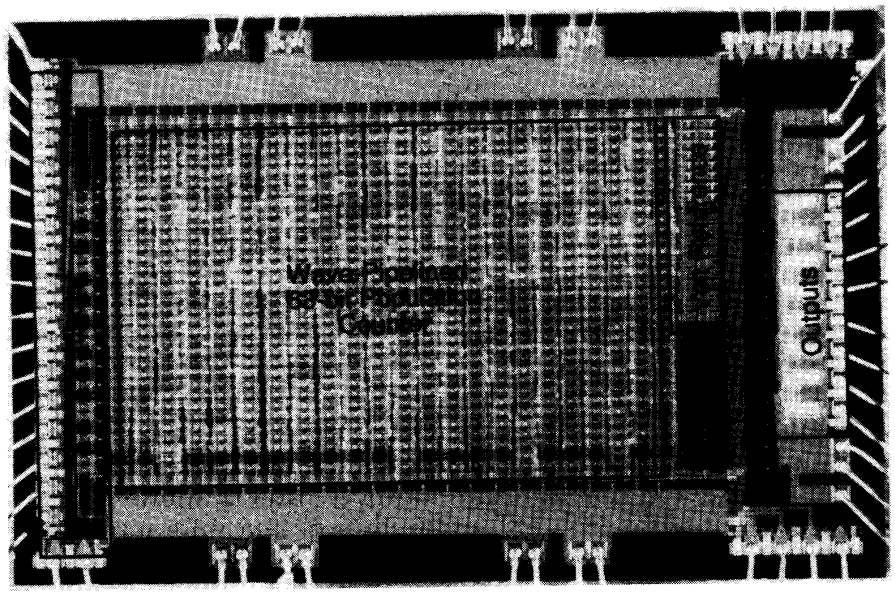


Fig. 9. Photomicrograph of wave pipelining demonstration chip.

of this chip and tests on fabricated chips are described in detail in [27]. In this paper, we primarily describe the results of using the CAD algorithms to design this chip.

The chip was designed to be a demonstration of wave pipelining rather than a commercial circuit. The logic circuit performs a 63-bit population counter function, i.e., it takes 63 parallel inputs and produces the number of 1's in that string as a binary number. This function is similar to a section of a high-speed combinational multiplier. The number of levels of logic is only slightly shorter than that required for a 64 × 64 multiply.

The circuit is a combinational logic circuit with 21 levels of logic and a nominal longest path delay of 8.5 ns plus 1 ns for the input/output drivers. After tuning, the path length difference due to design is about 1.1 ns, excluding the effects of differences in rising versus falling delays and data-dependent delays. The total delay variation Δt_P includes these effects plus process and temperature variation within the chip.

The circuit has been designed in a commercial bipolar process from Signetics called Qubic 1 [6]. The circuit is implemented in single-level CML using a standard cell technique. All the logic cells are OR/NOR gates, designed using a single level of current switches.

The circuit has been fully designed, simulated, fabricated, and tested. A photomicrograph of the chip is shown in Fig. 9. The chip dimensions are approximately 4 × 6 mm. The inputs are at the left, and the outputs are at the right. The number of input pads was reduced from 63 to 16 by wiring a few logic inputs to each pin. This reduction simplifies packaging and testing the prototype chip while still allowing 2^{16} distinct inputs to be applied.

To test wave pipelining, a sequence of 40 000 test vectors has been applied at various frequencies using a commercial chip tester called a Trillium Delta-Master. The tests show that the chips can wave pipeline at up to 250 MHz (clock period = 4.0 ns). Using constraint (1) on maximum pipelining frequency described earlier, we can show that the worst-case difference in path length Δt_P is less than or equal to 2.75 ns. In contrast, the maximum clock frequency using ordinary pipelining would be about 97 MHz (clock period = 10.25 ns).

At 250 MHz, the time between registers for a particular wave is about 10 ns, but new waves are applied every 4 ns. To achieve this, the clock to the ending register is delayed by about 2 ns compared to the clock to the beginning register.

We used the following design flow:

1) First, a schematic netlist and the individual layout cells were designed.
2) An initial fine-tuning pass using estimated capacitances was optionally applied to minimize the imbalances prior to rough tuning.
3) The circuit was rough tuned.
4) A place and routed core logic circuit was made from the circuit schematics using a commercial auto-place and route tool. Unfortunately, the tool was designed to minimize total wiring area and was not intended to control capacitances or minimize total delay.
5) Parasitic capacitances on each net were extracted.
6) A final fine-tuning pass was performed. Since all the power levels of each gate have the same dimensions, the power levels of each gate can be changed without requiring any changes in the place and route.
7) Power buses, bias voltage generators, input/output buffers, and pads were added using manual layout techniques.

Because we used a commercial place-and-route tool

TABLE III
RESULTS OF VARYING D_{MAX} FOR A 63-BIT CML POPULATION COUNTER (ALL DELAYS ARE IN PS)

D_{MAX}	Longest Path	Shortest Path	ΔD	Current
8500	8561	7447	1114 (13.0% of D_{MAX})	207.5 mA
8500 (updated)	8546	7466	1080 (12.6% of D_{MAX})	206.6 mA
9000	9028	7860	1168 (12.9% of D_{MAX})	187.7 mA
10 000	10 027	8568	1459 (14.6% of D_{MAX})	162.2 mA
11 500	11 449	9698	1751 (15.3% of D_{MAX})	137.1 mA

whose optimization goal is total area, the length of each particular net is not optimized or even well controlled. In fact, lines with the same fanout could vary by as much as a factor of 5 or so in capacitance. The achievement of a small delay variation due to design despite the wide distribution of capacitances demonstrates the flexibility of these CAD techniques.

D. Tuning Experiments on Population Counter Design

Two trials were performed of steps 2 to 6 of the design flow. The first trial used a two-step method, where the circuit was first fine tuned using estimates of capacitances derived from a simple floor plan of the sections of the chip. Then the circuit was rough tuned to fill out the remaining imbalances. The placement was performed according to the floor plan, and the routing was done using an auto-router.

The second trial performed rough tuning using an integer gate delay model without using an initial fine-tuning pass. During rough tuning logic gates had delays of 1, and buffers could have delays between 1 and 3. The layout was performed using automatic placement and routing without a floorplan but using constrained locations for the global inputs and outputs.

The second trial uses more buffers but empirically achieved a better balancing of the circuit. This might be explained by two effects: residual imbalances and tuning flexibility. In the first trial, fine tuning balances the circuit as much as possible, so that some loops may have a residual imbalance which is smaller than the minimal buffer delay and therefore cannot be balanced (from the rough tuning algorithm's point of view).

Secondly, the initial fine-tuning pass in the first trial may set some gates at near maximum or minimum power. Denote one of these gates by X. If X's output has an actual capacitance after place and route which is not close to the original estimate, it might be impossible to achieve the delay assumed for X during rough tuning. Since the gate was set close to minimum or maximum power originally, it has limited tuning in one direction. Our implementation of the algorithm increases the tuning flexibility by limiting the power levels to a central subset of the entire range during the first fine-tuning pass.

The second trial has even greater tuning flexibility since all gates were assumed to have unit delay. No initial fine-tuning pass was used prior to rough tuning.

In the second trial, several fine-tuning iterations were made using the different values of D_{MAX}. Four runs were made, and one additional run using $D_{MAX} = 8500$ ps was later made using a slightly updated version of the program. The results are shown in Table III. The actual longest path does not exactly equal D_{MAX} and sometimes slightly exceeds it because the specified gate powers from the linear program solution are rounded to the nearest power level for each gate.

The empirical results highlight the fact that although rough and fine tuning are individually optimal (in a previously defined sense) in minimizing the imbalances in path delays, the tuning process including the effects of actual versus estimated capacitances is heuristic and may be amenable to experimentation.

VIII. SUMMARY AND FUTURE DIRECTIONS

Wave pipelining can potentially increase a system's clock frequency by 2 to 3 times without using additional pipeline registers. To maximize clock rate, we must minimize the variation in path length.

We have developed both rough-tuning and fine-tuning algorithms and implemented these in computer programs.

The rough-tuning algorithm inserts delay elements such that the circuit can be balanced by setting gate parameters. Rough tuning constructs a spanning set of loops in a graph representation of a circuit, then balances the loops by inserting delay elements. By building the loops from a longest path spanning tree, the loops can always be balanced independently. A linear program performing repadding minimizes the number of delay elements required. Rough tuning is guaranteed to balance the circuit within some ΔD_R depending on the available delay elements.

The fine-tuning algorithm uses a linear program to solve for a set of gate drives that balances the circuit delays. The linear program corresponds to the problem of minimizing the power consumption of the circuits subject to delay constraints. Under certain assumptions, the algorithm can guarantee that the final imbalance ΔD is less than or equal to the remaining imbalance after rough tuning. In practice, rough tuning operates in conjunction with fine tuning to design wave-pipelined circuits that have both minimal added area and minimal total power consumption.

The more complex case where the rising delay does not equal falling delay is a difficult problem for both rough and fine tuning. Well-balanced solutions might not even exist. The best that our methods can do is to guar-

antee a reasonable bound on the difference ΔD in propagation time under nominal conditions. In practice, ΔD is usually small compared to t_P, so an efficient wave-pipelined implementation can still be designed.

As shown in Appendix II, some technologies are easier to use than others for wave pipelining. Static CMOS has substantial pattern-dependent delay variations which require special techniques to circumvent. CML and superbuffered ECL have good delay properties. ECL has the advantage of small individually adjustable emitter-followers with the disadvantage of unequal rise–fall delays.

Using the tuning algorithms, a demonstration wave pipelining chip has been successfully designed, fabricated, and tested. After tuning, the variation ΔD in path delay due to design was much less than the nominal critical path delay D_{MAX}. The actual fabricated chips show an improvement in clock frequency of 2.5× using wave pipelining.

This prototype chip plus chips from other researchers [11], [18], [3] show that wave pipelining is a potentially powerful and beneficial technique for increasing pipeline rate. Future research for wave pipelining might include additional logic resynthesis and layout techniques, novel testing techniques, and larger wave-pipelined systems. In addition, substantial benefits may arise from developing gate structures that are designed specifically for wave pipelining in both CMOS and bipolar.

Appendix I
Deriving the Minimum Clock Period Relation

The maximum pipeline rate is affected by technological parameters. Here we derive relations for minimal clock period as a function of variations in path length, clock skew, rise–fall times, and the setup and hold time of the storage elements. These relations are extensions upon those stated in [4], [9], [8], and [15] in order to include clock skew and rise–fall times. Reference [26] contains a detailed clocking analysis of both wave and ordinary pipelining.

A. Definitions

- t_S Setup time for registers or latches
- t_H Hold time for registers or latches
- t_{TRANS} Length of transparent period for latches
- t_P Propagation time of the longest path in the combinational logic
- Δt_P Maximum time difference between longest and shortest paths over worst-case design, process, and environment
- ΔC Worst-case uncontrolled clock skew
- t_{CP} Clock period
- t_{RF} Worst-case rise or fall time (10%–90% voltage swing) at the last logic stage.

B. Wave Pipelining Using Edge-Triggered Registers

1) Interference Fault Constraints:

One type of constraint arises because one wave cannot race ahead and reach the end of the combinational logic before the previous wave has been clocked into the ending storage elements. Otherwise, the two waves would interfere and the data of the previous wave would be lost.

At time 0, a clock pulse nominally arrives at the beginning register A, and a new data wave leaves. Note that a storage bit within register A may have a clock skew of up to ΔC compared to another storage bit in A.

We denote the time that a particular wave has to propagate from A to the ending register B as the *latency interval* t_L. The clock pulse to capture the wave at B arrives nominally at time t_L but could be skewed early or late. Clock skew is defined as potentially occurring between individual storage bits in A and B. The minimum latency interval between the clocking of any bit in A and any bit in B is $t_L - \Delta C$, and the maximum clocking interval is $t_L + \Delta C$.

Due to the propagation delay of the combinational logic, the earliest that the data wave could begin to arrive at register B is $t_P - \Delta t_P$ after starting from register A. The latest that the data wave could arrive is t_P after starting from A.

For simplicity, we begin by assuming that all transitions occur with zero rise and fall times.

Since the longest possible propagation delay plus setup time must be less than the minimum latency interval (see Fig. 10):

$$t_L - \Delta c > t_P + t_S. \tag{2}$$

This long-path constraint on the latency interval t_L is identical to the long-path constraint on clock period for ordinary pipelining (i.e., not wave pipelining) in edge-triggered register systems.

Another constraint must be used to prevent consecutive waves from interfering (see Fig. 11). Suppose that wave 1 nominally starts from A at time 0 and is nominally clocked into B at time t_L. A worst case clock skew between registers A and B and between individual bits of A can be modeled by assuming that some bits of A start at a time ΔC before the nominal. If the last bit of A is clocked at time 0, then the first bit is clocked at time $-\Delta C$. Wave 2 nominally starts at time t_{CP}, but part of wave 2 could potentially start at time $t_{\mathrm{CP}} - \Delta C$ compared to the start of wave 1, due to clock skew within register A. The earliest that wave 2 could begin to interfere at register B is then $t_P - \Delta t_P + t_{\mathrm{CP}} - \Delta C$:

$$t_P - \Delta t_P + t_{\mathrm{CP}} - \Delta C > t_L + t_H. \tag{3}$$

By adding constraints (2) and (3) and solving for t_{CP}, we get the following relation for the clock period:

$$t_{\mathrm{CP}} > \Delta t_P + 2 * \Delta C + t_S + t_H. \tag{4}$$

Now suppose that the stage of logic that is connected to the inputs of register B has a worst-case rise–fall transition time t_{RF}. Inequality (3) is then changed because the earliest signal of the next wave can begin to interfere with the current wave when the transition between waves begins.

This transition begins at a time t_{RF} before the next wave

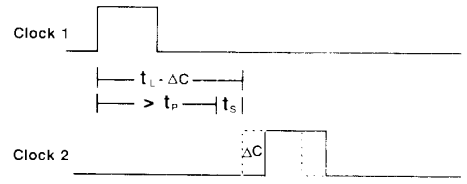


Fig. 10. An illustration of timing constraint $t_L - \Delta C > t_P + t_S$ for wave pipelining using edge-triggered registers. Clock 1 starts the wave from the beginning register, and clock 2 captures the wave at the ending register.

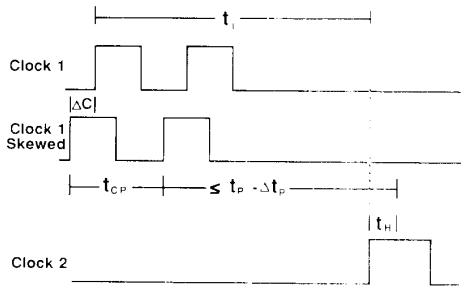


Fig. 11. An illustration of timing constraint $t_P - \Delta t_P + t_{CP} - \Delta C > t_L + t_H$ for wave pipelining using edge-triggered registers. Clock 1 starts the wave from the beginning register, and clock 2 captures the wave at the ending register.

actually arrives at B. Since the earliest signal of wave 2 must not interfere with the clocking of wave 1:

$$t_P - \Delta t_P + t_{CP} - \Delta C - t_{RF} > t_L + t_H. \quad (5)$$

This changes the inequality for t_{CP} to the following.

$$t_{CP} > \Delta t_P + 2 * \Delta C + t_S + t_H + t_{RF}. \quad (6)$$

Another constraint arises because one wave cannot be allowed to interfere with another wave inside a section of combinational logic. The constraint must be satisfied at every signal node X in the circuit (called an *internal signal node*) to prevent two waves from colliding.

The following definitions are required

- t_X Propagation time of the longest path from the global inputs to an internal signal node X.
- t_{RF} Worst-case rise–fall time at X.
- t_{MS} Minimum time that X must be stable for the next stage of logic to operate correctly.
- Δt_X Maximum time difference between the longest and shortest paths leading from the global inputs to node X. Δt_X is always less than or equal to Δt_P.

The latest that a wave could arrive at X is t_X. The earliest that the next wave could begin to arrive at X is $t_X - \Delta t_X - t_{RF} + t_{CP} - \Delta C$. In this case, ΔC is included because the second signal may originate from a different storage bit than the first signal and the two bits may be skewed by as much as ΔC.

To avoid interference, the next wave must arrive at least t_{MS} later:

$$t_X - \Delta t_X - t_{RF} + t_{CP} - \Delta C + t_{MS} \leq t_X. \quad (7)$$

Simplifying yields the second constraint on clock period as

$$t_{CP} > \Delta t_X + \Delta C + t_{MS} + t_{RF}. \quad (8)$$

During the design process, each section of combinational logic is made balanced in delay as far as possible. The goal of minimizing Δt_P includes minimizing Δt_X for all signals X. As a result, the goal of satisfying inequality (8) is usually no more stringent than satisfying inequality (6).

C. Wave Pipelining Using Transparent Latches

Similar constraints can be stated for wave pipelining using transparent latches.

Let t_L denote the beginning of the transparent period for the beginning latch A.

Suppose that minimal restrictions are placed when the signals from the previous logic section arrive at the beginning latch. In other words, the logic section before the beginning latch might produce outputs at any time up to the end of the transparent period minus a setup time, i.e., time $t_L + t_{TRANS} - t_S$. In the worst case, the transparent period minus a setup time must be considered an additional time uncertainty, since a wave could arrive at the beginning latch from the previous logic section between t_L and $t_L + t_{TRANS} - t_S$. Then the waves leaving the beginning latch could depart at any time within a period of length $t_{TRANS} - t_S$. Thus an additional safety margin of $t_{TRANS} - t_S$ must be allowed between waves. This uncertainty may be reduced if the designer can make tighter bounds on the departure time from the beginning latch. For instance, it might be possible to guarantee that the waves arrive at the beginning latch earlier than $t_{TRANS} - t_S$ in the transparent period. See [26] for more details on when this safety margin can be reduced.

1) Interference Fault Constraints: The following constraint ensures that the second wave does not race ahead and interfere with the first wave at the ending latches:

$$t_{CP} > \Delta t_P + 2 * \Delta C + t_S + t_H + t_{RF} + (t_{TRANS} - t_S) \quad (9)$$

which simplifies to

$$t_{CP} > \Delta t_P + 2 * \Delta C + t_H + t_{RF} + t_{TRANS}. \quad (10)$$

The next constraint applies for all internal signals X and ensures that the second wave does not interfere with the first wave at any point within the combinational logic:

$$t_{CP} > \Delta t_X + \Delta C + t_{MS} + t_{RF} + (t_{TRANS} - t_S). \quad (11)$$

In addition, the long-path constraint on the latency intervals between latches is identical to the long-path constraint in ordinary pipelining using single-phase clocking and transparent latches. See [26] for details.

Since t_{TRANS} is included in the clock period, it is advantageous to keep it small.

APPENDIX II
WHICH TECHNOLOGY IS BEST FOR WAVE PIPELINING?

Some integrated circuit technologies are better than others for wave pipelining. Ideally, a technology should possess the following properties:

1) Many points of finely controlled speed adjustment which have predictable effects;
2) The same gate delay whether the output is rising or falling;
3) No variation in gate delay depending on input pattern;
4) No variation in gate delay depending on previous input patterns;
5) Zero contribution to Δt_P due to process- and temperature-induced variations within a chip;
6) The standard goals of high noise immunity, low power, high density, and high speed.

Let us examine CMOS, ECL, CML, and super-buffered ECL with respect to these goals.

A. CMOS

Static CMOS (see Fig. 12) is not naturally well suited for wave pipelining because the gate delay depends on the input pattern. Any parallel transistor arrangement will have variable delay, due to the varying amount of resistance. A series transistor arrangement also has variable delay because the capacitance that must be charged depends on which transistor in the series is the last to begin conducting. The number of waves possible in an ordinary static CMOS design would be limited by these effects. Many Bi-CMOS gate structures also cause the same type of data-dependent delays.

Fortunately, other researchers are developing techniques to help alleviate this problem. Gray *et al.* [10], [11] and Klass and Mulder [13], [14] have developed CMOS circuit methodologies for maximizing the number of waves. Their design methods can potentially achieve a high degree of wave pipelining in CMOS, in spite of the delay variations discussed above. Lien and Burleson [18] describe the use of wave pipelining in domino CMOS logic.

B. ECL

A basic ECL gate is shown in Fig. 13. The delay is controlled by adjusting the "tail current" by sizing the resistors proportionately. A gate can have multiple independently adjustable emitter-followers for different fanouts.

The logic section of the gate has fairly balanced rise-fall delay, but the emitter-follower section has rising delays which are much shorter than falling delays. This can be partially overcome by using a modified emitter-follower which slows down the rising delay (see Fig. 14). Higher tail currents also reduce the difference in delays.

An example of a stacked ECL structure is shown in Fig. 15. In stacked structures, the transition speed depends somewhat on the input patterns. The intermediate node x can be charged to different voltages just prior to switching, thus causing variations in delay. For instance,

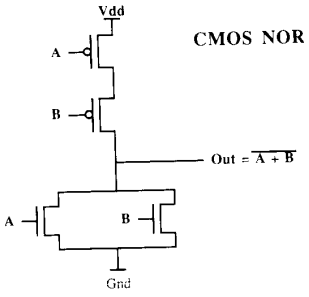


Fig. 12. CMOS gate.

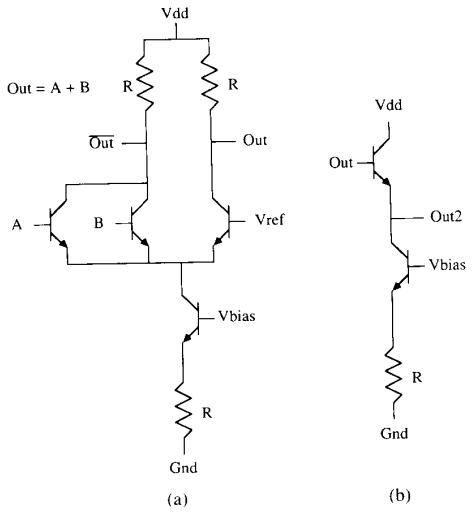


Fig. 13. ECL gate. (a) Logic section. (b) Emitter follower.

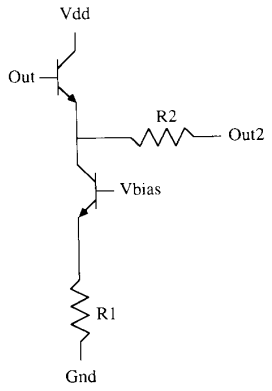


Fig. 14. Modified emitter-follower.

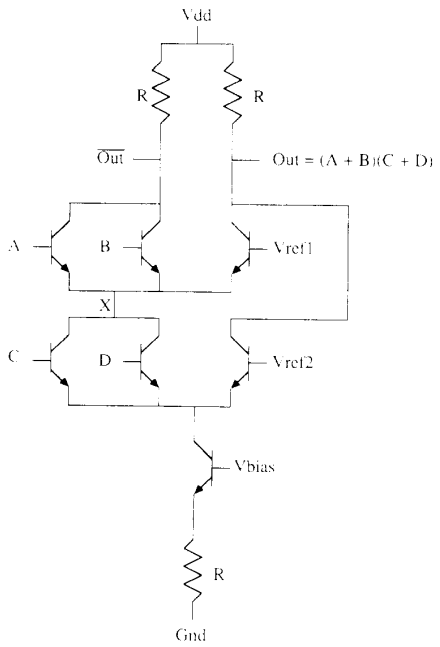


Fig. 15. Logic section of stacked ECL structure.

suppose the voltage were higher on the reference side for both the upper and lower levels. Now suppose that the A input of the upper level rose momentarily. This would charge the intermediate node X to a higher voltage. This would be additional charge to be drained when the lower level switches, thus slowing down the gate.

We simulated some typical stacked structures in Qubic 1, and found that for any particular input–output transition, the best-case delay was at least 82% of the worst-case delay [26], [2]. This 18% variation is substantial, but a reasonable degree of wave pipelining is still possible [26].

C. CML

One way to eliminate the unbalanced rise–fall delay in ECL is not to use emitter-followers. Instead, the output of one ECL logic stage is directly connected to the next in a logic family called CML. In non-wave-pipelined designs, CML is often somewhat slower than ECL because both rising and falling transitions are through limited current devices. In contrast, ECL has a fast rising delay that can be used effectively with inverting logic. However, a faster rising delay is not advantageous for wave pipelining.

Since there are no emitter-followers, one cannot independently adjust each fanout except by using additional gates as buffers. A buffer gate costs three transistors and three resistors, compared to an emitter-follower of two transistors and one resistor. In addition, no wired OR is possible.

D. Super-Buffered ECL

There are high-performance ECL buffers which have balanced delay. By using dynamic techniques, such circuits have fast transitions in both directions, i.e., faster than the static current source can achieve. A circuit presented by Coy *et al.* [5] is repeated in Fig. 16 in a modified form.

The advantage of super-buffered ECL is that it is very fast and has balanced delay with less power consumption (since the static current sources can be smaller). Unfortunately, each buffer costs four transistors, two resistors, and one capacitor, compared to an emitter-follower of two transistors and one resistor.

E. General Remarks on the Current Steering Technologies

In ECL, CML, and super-buffered ECL, the use of stacked structures increases logic density but reduces the number of waves somewhat. One should consider using a reduced supply voltage to conserve power if stacked structures are not used.

The effect on Δt_p of process and temperature-induced variations within one chip should be similar for all the current-steering technologies.

Careful attention should be paid to the power and ground networks during design. Voltage drops should be small; otherwise, the result would be equivalent to shifting the reference voltages. This would cause a difference in gate speed depending on whether the input is rising or falling.

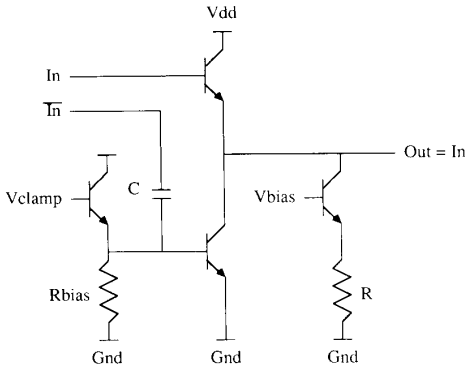


Fig. 16. Super-buffered emitter-follower. The capacitor C boosts the base voltage of the pull-down transistor to provide a dynamic pull-down current on Out.

F. Conclusions and Tradeoffs

If standard design techniques were used, static and dynamic CMOS would not be intrinsically as well suited for wave pipelining as ECL and CML. Using special design techniques to minimize the impact of delay dependencies, other researchers have been designing and fabricating wave-pipelined circuits in CMOS with some promising results [13], [14], [10], [11], [18]. Bipolar ECL, CML, and super-buffered ECL offer a range of tradeoffs between area, absolute speed, wave-pipelining potential, power, and number of independent adjustments:

- ECL has medium area density but has unbalanced rise–fall delay and fairly high power consumption.
- CML has high area density and balanced delays with medium power consumption. However, it offers fewer points of adjustment than ECL, unless additional buffer gates are used.
- Super-buffered ECL is very fast and has balanced delay, medium power consumption, and the same number of adjustments as ECL. However, buffers consume more area than emitter-followers.

To help decide among the various current-switching technologies, one needs to consider the fanout in the circuit. For high-fanout circuits, the need to adjust each fanout independently would favor ECL, since it has small buffers.

ACKNOWLEDGMENT

The other members of our research groups at Stanford have been very supportive. Frederic Mailhot did a wonderful job of building and maintaining the interface tools to the SLIF netlist language. Special thanks to Gary Bewick for much technical help and many enlightening discussions.

The wave pipelining demonstration chips were fabricated at Signetics, Inc. using their Qubic 1 Bi-CMOS process. Special thanks should be given to Ken-Sue Tan, Peter Baltus, Victor Akylas, Charlotte Skeeters, Douglas MacArthur, Frank Feng, Hedy Chen, Henry Wong, Bill Mack, Uzi-Bar Gadda, Joseph Kosteleck, Reda Razouk, Ron Cline, and others at Philips Research and Signetics for helping with the design, simulation, and fabrication of the demonstration circuits.

The physical layout of the demonstration chip was performed using CAD tools provided by the Silicon Design Division of Mentor Graphics, Inc. Rick Sedlak did a superb job of CAD support. The high-speed testing of the chips was performed at Trillium, Inc. Robert Huston, Gerry Labonville, and Thomas Berry at Trillium have helped tremendously.

Michael Saunders, Walter Murray, and Arthur Veinott, Jr. at Stanford were generous in their advice about practical considerations of solving optimization problems. Mark Horowitz was very helpful in pointing out pitfalls and in particular analyzing technology considerations. He thought of the modified ECL emitter-follower and suggested the use of super-buffered ECL.

REFERENCES

[1] S. Anderson, J. Earle, R. Goldschmidt, and D. Powers, "The IBM system/360 model 91 floating point execution unit," *IBM J. Res. Develop.*, vol. 11, no. 1, pp. 34–53, Jan. 1967.

[2] G. Bewick, private communication, 1991.

[3] T. Chappell, *et al.* "A 2-ns cycle, 3.8-ns access 512-kb CMOS ECL SRAM with a fully pipelined architecture," *IEEE J. Solid-State Circuits*, vol. 26, pp. 1577–1585, Nov. 1991.

[4] L. Cotten, "Maximum rate pipelined systems," in *Proc. AFIPS Spring Joint Computer Conf.*, 1969, pp. 581–586.

[5] B. Coy, A. Mai, and R. Yuen, "A 13,000 gate 3 layer metal bipolar gate array," in *Proc. Custom Integrated Circuits Conf.*, 1988, pp. 20.1.1–20.1.3.

[6] J. L. de Jong, *et al.*, "Single-polysilicon layer advanced super high-speed Bi-CMOS technology," in *Proc. IEEE Bipolar Circuits and Technology Meeting*, Minneapolis, MN, Sept. 1989, pp. 182–185.

[7] L. Chua, C. Desoer, and E. Kuh, *Linear and Nonlinear Circuits*. New York: McGraw-Hill, 1987, pp. 695–715.

[8] B. Ekroot, "Optimization of pipelined processors by insertion of combinational logic delay," Ph.D. dissertation, Stanford Univ., Dept. Elect. Eng., Stanford, CA, Sept. 1987.

[9] B. Fawcett, "Maximal clocking rates for pipelined digital systems," Coordinated Science Lab., Univ. of Illinois, Urbana, Rep. R-706, Dec. 1975.

[10] C. T. Gray *et al.*, "A high-speed CMOS FIFO using wave pipelining," Dept. Elect. and Comp. Eng., North Carolina State Univ., Raleigh, Tech. Rep. NCSU-VLSI-91-01, Jan. 1991.

[11] C. T. Gray *et al.*, "Theoretical and practical issues in CMOS wave pipelining," in *Proc. VLSI '91*, Aug. 1991, Edinburgh, U.K., pp. 9.2.1–9.2.10.

[12] D. Joy and M. Ciesielski, "Placement for clock period minimization with multiple wave propagation," in *Proc. 28th Design Automation Conf.*, San Francisco, CA, June 1991, pp. 640–643.

[13] F. Klass and J. M. Mulder, "CMOS implementation of wave pipelining," Delft Univ. of Technology, Delft, The Netherlands, Tech. Rep. 1-68340-44(1990)02, Dec. 1990.

[14] —, "Use of CMOS technology in wave pipelining," in *Proc. 5th Conf. VLSI Design*, Bangalore, India, Jan. 1992, pp. 303–308.

[15] P. Kogge, *The Architecture of Pipelined Computers*. New York: McGraw-Hill, 1981.

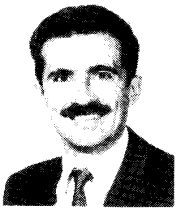
[16] E. Lawler, *Combinatorial Optimization: Networks and Matroids*. New York: Holt, Rinehart, and Winston, 1976, pp. 129–133.

[17] C. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry by retiming," in *Proc. 3rd CalTech Conf. Very Large Scale Integration*, 1983, pp. 87–116.

[18] W. Lien and W. Burleson, "Wave-domino logic: Timing analysis and applications," presented at the IEEE Int. Symp. Circuits and Systems, San Diego, CA, May 1992.
[19] Q. Lin and P. Xia, "The design and implementation of a very fast experimental pipelining computer," *J. Comp. Sci. Technol.*, Beijing, China, vol. 3, no. 1, pp. 1-6, 1988.
[20] D. Marple, "Performance optimization of digital VLSI circuits," Ph.D. dissertation, Dept. Elect. Eng., Stanford Univ., Stanford, CA, Sept. 1986.
[21] B. Murtagh and M. Saunders, "MINOS 5.1 User's Guide," Systems Optimization Laboratory, Operations Research, Stanford Univ., Stanford, CA, Tech. Rep. SOL 83-20R, Jan. 1987.
[22] A. Veinott, Jr., private communication, 1989.
[23] S. Waser and M. Flynn, *Introduction to Arithmetic for Digital Systems Designers*. New York: Holt, Rinehart, and Winston, 1982.
[24] D. Wong, G. De Micheli, and M. Flynn, "Inserting active delay elements to achieve wave pipelining," in *Proc. Int. Conf. Computer-Aided Design '89*, Santa Clara, CA, 1989, pp. 270-273; Dep. Elect. Eng., Stanford University, Stanford, Computer Systems Laboratory Tech. Rep. CSL-TR-89-386, 1989.
[25] ——, "Designing high-performance digital circuits using wave pipelining," in *Proc. VLSI '89*, Munich, Germany, Aug. 1989, pp. 241-252.
[26] D. Wong, "Techniques for designing high-performance digital circuits using wave pipelining," Ph.D. dissertation, Dep. Elect. Eng., Stanford Univ., Stanford, CA, Aug. 1991, also Computer Systems Lab. Tech. Rep. CSL-TR-92-508, Stanford, Univ., Stanford, CA.
[27] D. Wong, G. De Micheli, M. Flynn, and R. Huston, "A bipolar population counter using wave pipelining to achieve 2.5× normal clock frequency," *IEEE J. Solid-State Circuits*, vol. 27, May 1992, pp. 745-753; also in *Proc. 1992 Int. Solid-State Circuits Conf.*, 1992, pp. 56-57.

Derek C. Wong (S'88-M'92) received the B.S. degree in electrical engineering from the University of California, Berkeley, in 1985, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1986 and 1991, respectively.

He is a Post-Doctoral Fellow in the Department of Electrical Engineering at Stanford. Since 1988, he has conducted research on techniques for designing circuits using wave pipelining. He has developed novel CAD algorithms for designing wave pipelined circuits and an LSI chip demonstrating wave pipelining. He has also published an algorithm for high-speed division and an architecture for high-speed logic resolution. His current research interests are in high-speed digital systems, computer organization, CAD, and multimedia.



Giovanni De Micheli (S'79-M'80-SM'89) received the Dr. Eng. degree (summa cum laude) in nuclear engineering from the Politecnico di Milano, Italy, in 1979, and the M.S. and Ph.D. degrees in electrical engineering and computer science from the University of California at Berkeley in 1980 and 1983, respectively.

He is an Associate Professor of electrical engineering and computer science at Stanford University, Stanford, CA. From 1984 to 1986 he was with the IBM T. J. Watson Research Center, Yorktown Heights, NY, as Project Leader of the Design Automation Workstation Group. Previously he held positions at the Department of Electronics, Politecnico di Milano, and Harris Semiconductor, Melbourne, FL. His research interests include several aspects of computer-aided design of integrated circuits, especially automated synthesis, optimization, and verification of VLSI circuits. He is co-editor of *Design Systems for VLSI Circuits: Logic Synthesis and Silicon Compilation* (Norwell, MA: Martinus-Nijhoff, 1987). He was also co-director of the Advanced Study Institute on Logic Synthesis and Silicon Compilation, held in L'Aquila, Italy, under the sponsorship of NATO in 1986 and 1987. He is Associate Editor for the IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS and *Integration: The VLSI Journal*. He was technical and general chairman of the International Conference on Computer Design (ICCAD) in 1988 and 1989, respectively. He has served as a member of the technical committee of the ICCAD, ICCD, and DAC. He has also served as a member of the executive committee of the New York Chapter of the Computer Society in 1985 and 1986.

Dr. De Micheli received the 1987 Best Paper Award of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF ICAS and a Best Paper Award at the 20th Design Automation Conference in June 1983.

Michael J. Flynn (M'56-SM'79-F'80) is a professor of electrical engineering at Stanford University. For ten years, he worked at IBM Corporation in computer organization and design. He was also a faculty member at Northwestern and Johns Hopkins Universities, and the Director of Stanford's Computer Systems Laboratory from 1977 to 1983. He has served as vice president of the Computer Society and was the founding chairman of the Technical Committee on Computer Architecture, as well as ACM's Special Interest Group on Computer Architecture.