Inserting Active Delay Elements to Achieve Wave Pipelining

Derek Wong, Giovanni De Micheli and Michael Flynn *
Department of Electrical Engineering
Stanford University
Stanford, California 94305

Abstract

Wave pipelining is a technique for pipelining digital systems that can increase clock frequency 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 set-up 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 by inserting a minimal number of active delay elements to lengthen short paths. This method can be combined with delay balancing by adjusting gate speeds [11] to design wave-pipelined circuits.

1 Introduction

Wave pipelining is a design method that can potentially boost the pipeline rate of practical circuits by 2 to 3 times 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. In contrast, wave pipelining is the use of multiple coherent "waves" of data between storage elements (see Figure 1). This is achieved by clocking the system faster than the propagation delay between registers. The capacitance in the combinational logic circuit is 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 nsec. Instead of operating at 100 Mhz, the multiplier could operate with 3 pipeline waves to achieve a clock frequency of 300 Mhz with the same 10 nsec latency.

To operate at the highest possible clock 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.

To our knowledge, there have been two published circuit implementations of wave pipelining, the IBM 360/91[1, 2] and a recent experimental computer[7], as well as some theoretical work [5, 4]. The IBM 360/91, was implemented in SSI by balancing the circuit delays using manual design techniques. The other wave-pipelined computer also was implemented by balancing the delays without the aid of CAD techniques.

Our ultimate goal is to build a wave-pipelined chip in VLSI, identifying and solving the necessary practical problems enroute. Currently, we are designing new algorithms and the necessary CAD tools to automatically balance the delays in a combinational logic circuit while minimizing power consumption and added circuitry.

We have developed our tools for bipolar design in single-level, ECL/CML technology since this technology is more suitable for wave pipelining than MOS. For more details on the concept of wave pipelining, technological considerations, and previous work in this area, we refer to another paper [11].

In this paper, we discuss algorithms for inserting additional circuit elements to lengthen short paths in a circuit. Although this method can in principle be applied to circuits in any technology, in practice this method operates in conjunction with another method of balancing delays which is primarily designed for ECL/CML circuits. The other technique, adjusting gate parameters that affect current and delay, is described in [11]. Following an explanation of the frequency limit of wave pipelining, we explain the problem of balancing paths in a circuit and the general strategy for solving it. Then we describe the method of inserting additional circuit elements in detail. We conclude with a description of example applications and future research directions.

*This work was supported in part by an NSF Graduate Fellowship and a supplement from Stanford University. This research was also supported by the Center for Integrated Systems at Stanford. The work was performed using equipment provided by NASA under contract NAGW 419. This work was also supported by NSF, DEC, and AT&T under a PYI award.

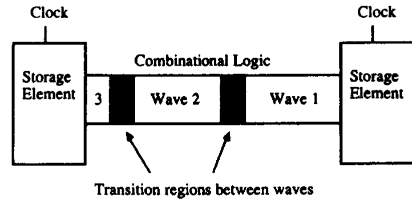


Figure 1: In wave pipelining, multiple coherent waves of data are sent through combinational logic acting as a pipeline.

2 Maximum Pipeline Rate

It can be shown [11] that the minimum clock period at which a wave-pipelined circuit can be clocked is limited by $t_{CP} > \Delta t_P + 2 * \Delta C + t_{SH} + t_{RF}$ where

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

In this paper, we focus on reducing the clock period by reducing the variation in path length Δt_P. We present algorithms to modify circuits to make them balanced under nominal process and temperature conditions. Ideally, if all path delays from input to output are equal under nominal conditions, the clock frequency is limited only by ΔC, t_{SH}, t_{RF}, and Δt_P due to process/temperature variations.

3 The Wave Pipelining Problem

3.1 Problem Definition

To design wave-pipelined circuits, we need to balance the path delays. For practical circuits, we constrain the nominal path delay to be a predefined constant D_{MAX} because the circuit must interface to other components in the system. We therefore define the balancing problem of wave pipelining as follows:

Balancing Problem

Given a combinational logic circuit without feedback, derive an I/O equivalent circuit such that all nominal path delays from input to output are equal to a given constant D_{MAX}.

□

The following modeling assumptions are made about the circuit:

1. Each gate propagates signals one way from inputs to outputs.
2. Gate delay can be adjusted by a parameter (e.g. tail current in ECL/CML) and can range from $T_{MIN}[i]$ to $T_{MAX}[i]$ for each gate i.

CH2805-0/89/0000/0270$01.00 © 1989 IEEE

3. Adjusting the delay of a gate does not affect the capacitative load at its inputs or outputs.
4. Path delays can be increased by inserting active delay elements. These elements are buffers with one input and one non-inverting output. They have adjustable delays between B_{MIN} and B_{MAX}.

We propose two ways of attacking the balancing problem: inserting delay elements and adjusting gate parameters. They are called fine and rough tuning respectively:

- *Fine tuning* adjusts gate parameters so that the circuit is balanced. As a secondary goal, it minimizes power consumption.
- *Rough tuning* inserts a minimal number of delay (padding) elements so that it is possible to balance the circuit by just adjusting the gate parameters. Minimizing the number of inserted elements minimizes the added area.

The overall strategy to balance a circuit is as follows. Before inserting delay elements, we tune the gate parameters using fine tuning to balance the circuit as much as possible, thus minimizing the number of delay elements needed. After this, the rough tuning procedure is run to insert active delay elements where necessary. With the proper type of delay element, a final fine tuning pass will balance the circuit within a tolerance factor proportional to the minimum possible delay B_{MIN} of a delay element.

In this paper, we present the algorithms for rough tuning in detail. Fine tuning is described in another paper [11].

4 Rough Tuning

We model the circuit using a polar, weighted, directed acyclic graph. 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 weights on the two types of arcs represent propagation delays of gates and delays of inserted elements, called *padding elements*. The first set of weights are known, while the second set represents the unknown of the problem. Our technique determines this second set of weights.

A graph is constructed from a circuit using the following steps:

1. A node is associated to each output (inverting and non-inverting) of each gate. An additional 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]'s are zero.
4. A source and sink 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.

This type of delay model is sufficient to accurately represent delays in a single-level ECL/CML gate because the delays are equal from each input.

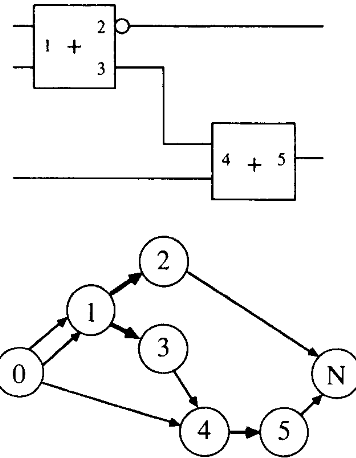
Figure 2 shows an example conversion of a small circuit to our graph representation. Nodes 1 and 4 represent the inputs of the two gates. Nodes 2, 3, and 5 represent the outputs. Heavy arcs are internal to one gate; regular arcs are external.

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

4.1 Problem Formulation for Rough Tuning

Before stating the rough tuning problem, we would like to comment on how gate delay models affect the capability of a rough tuning technique to solve the balancing problem.

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



Light type E arcs represent inserted delays.
Heavy type I arcs represent gate delays.

Figure 2: An example conversion from a circuit to a DAG for Rough Tuning.

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 weights 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.

Defn. Rough Tuning Problem

Given a combinational logic circuit without feedback, find a set of implementable arc weights W to

- Make the longest source-to-sink path have weight D_{MAX}, and
- Minimize ΔD where the shortest path has weight $D_{MAX} - \Delta D$.

□

The parameter ΔD measures the path length difference due to design.

An *optimal* solution is one that minimizes added area, i.e. it minimizes the number of added delay elements $\sum \lceil (W[i]/B_{MAX}) \rceil$.

Using realistic models of delay elements, the rough tuning algorithm guarantees a reasonable bound on ΔD (see section 4.3) that should be less than the worst-case path variation due to the other factors affecting Δt_p (process variations, temperature fluctuations, and data-dependent delays).

4.2 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 weights.

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 weight of each side is the sum of the arc weights including both I and E-type arcs. If the weight of the two sides is 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 weight D_{MAX}. Then, the circuit is balanced if and only if all the loops are balanced. In addition, the circuit is also balanced if and only if a spanning set of linearly-independent loops is balanced, commonly called a set of fundamental loops [3].

Since the number of fundamental loops is linear in the size of the problem, this result enables us to verify efficiently whether a circuit is balanced or not. A spanning set of fundamental loops can be constructed as follows [3]:

1. Construct a spanning tree in the DAG beginning from the source.
2. Let A represent the set of arcs that are not in the tree.
3. Let L be the (initially empty) set of loops.
4. Add one arc (called a *link*) from A to the arcs in the tree. This defines exactly one loop called a *co-tree 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 non-tree arc 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 1: In any fundamental loop constructed from a longest-path spanning tree, the link is always on the side of smaller weight.

Theorem 1. Non-tree Arcs are Type E

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

The proof is reported in [12].

From Proposition 1 and Theorem 1 it follows that every fundamental loop has an adjustable (type E) arc on the side with smaller total weight.

4.3 An Algorithm for Rough Tuning

Based on the above idea of balancing a spanning set of loops, our 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 well-balanced solution, optimizing it, and implementing the weights by using padding elements. The second step may be skipped, if non area-optimal but balanced circuits are sought for.

Rough Tuning Algorithm

1. Construct a DAG to represent the circuit. Add one type I arc of weight 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 weight $W[i]$ to balance the corresponding fundamental loop.
4. Apply the repadding algorithm (described in section 4.3.1) to minimize the number of delay elements.
5. Implement the delay weights by inserting delay elements.

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

At step 5, the delay weights are implemented as follows. For a weight $W[i] \geq B_{MIN}$, 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).

For any weights that are smaller than B_{MIN}, one of the following heuristic methods can be used:

- Ignore any weights smaller than B_{MIN}.

  Then ΔD is bounded from above by $\Delta D < nB_{MIN}$ where every input-to-output path has at most n type E arcs that have weight $< B_{MIN}$.
- Implement weights greater than $B_{MIN}/2$ with a single delay element.

  Then ΔD is bounded from above by $\Delta D < nB_{MIN}/2$.

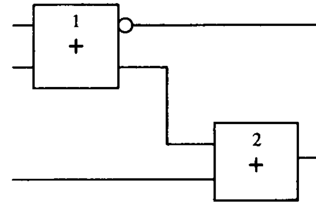
  If D_{MAX} is small as possible (i.e. equal to the critical path with all gates at max power in the original circuit), then the new critical path t_P could exceed D_{MAX} by up to $nB_{MIN}/2$.

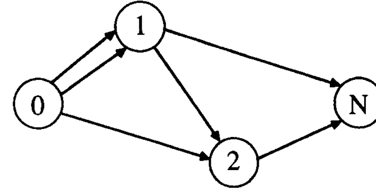
Note that ΔD represents an upper bound on the mismatch of the source-to-sink path delays. Fine tuning [11] can further reduce this mismatch and in some cases it may perfectly balance a circuit.

4.3.1 Achieving an Optimal Solution

Delay weights can sometimes be shifted from one portion of a circuit to another. For instance, if all the type E arcs on one side of a gate (either at the input or output) have positive weight, then some delay can be shifted to the other side. This can reduce the number of delay elements required when the number of arcs is fewer on the other side.

This section presents a method for systematically shifting delay weights to minimize the number of delay elements. We call this method *repadding*.





A Simplified Graph uses one node per gate and only type E arcs.

Figure 3: An example conversion from a circuit to a simplified DAG for Repadding.

It is derived from a method described in [6] for minimizing register count in sequential circuits by moving register boundaries.

We first obtain a simplified graph by contracting all arcs of type I (Internal). An example of this is shown in Figure 3. Nodes 1 and 2 correspond to the two gates. An external arc exists for each connection in the circuit.

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

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

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 weights 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 $\sum R[i](indegree[i] - outdegree[i]))$

subject to:

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

$R[0] = R[N] = 0$ for source 0 and sink N.

The constraints guarantee that the solution does not have negative arc weights [6].

The function $indegree(v) - outdegree(v)$ measures the reduction in $\sum W_X[i]$ 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 $\sum W_X[i]$.

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 weights using

   $W'_X[(i,j)] = W_X[(i,j)] + R[j] - R[i]$.
4. Recompute the full arc weights using $W'[i] = W'_X[i] * B_{MAX} + W_Y[i]$.

4.3.2 Remarks on Repadding

- It can be shown that every basic optimal solution is an integral vector (assuming an optimal solution exists) [6][9]. This ensures that only entire units of delay can be shifted.
- By using the weight pairs, we do not allow the repadding algorithm to shift delays that are smaller than B_{MAX}. It can be shown that shifting delays smaller than B_{MAX} can in some cases increase rather than decrease the number of delay elements needed.
- When an output has multiple fanouts and more than one fanout arc has a positive delay weight, the physical implementation can share delay elements rather than implementing multiple delay chains. Sharing padding elements can still be formulated as a linear program. The details are not reported here, but a similar formulation is reported in [6].

4.4 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 guarantees 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 guarantees to balance a circuit after fine tuning to within $\Delta D < nB_{MIN}$ or $nB_{MIN}/2$ (depending on the heuristic chosen in section 4.3), where every input-to-output path has at most n type E arcs with weight less than B_{MIN}.
3. For circuits using delay elements that have an adjustable delay from B_{MIN} to B_{MAX}, the method locally minimizes the number of added delay elements.
4. 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.

5 Implementation and Applications

The rough tuning algorithm has been implemented by a computer program that interfaces to the format for logic design called SLIF developed at Stanford.

We now present the results of applying this method to four example 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. Rough tuning is required to balance the circuit. If each padding element is assumed to take $3/4$ of the area of an average gate, then the results are as reported in Table 1.

The second two circuits are the partial-products generator plus carry-save adder sections of 4x4 and 8x8-bit combinational CML multipliers. Approximate CML gate delay and capacitance models were developed based on simulations of ECL circuits.

The results were achieved using the combined rough/fine tuning procedure. The final fine tuning pass was not performed since the circuits were not laid out. The run-time 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 only takes a tiny portion of the run-time using a heuristic approximation to retiming (with no linear program solving necessary).

About 5% to 10% of D can be added to the final ΔD to include the difference between rising and falling delays in CML. The increase in cell area is given as two numbers. The first uses a straightforward implementation where single-output CML buffers are 64% as large as a 3-input OR gate. However, since nearly all signals are used in both polarities and the padding is nearly identical for both, the number of buffers can be cut nearly in half by replacing true and complement buffers with slightly larger, dual-output buffers. This results in the reduced area estimate in parentheses.

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

Circuit	Add4	Add16	Mult4x4	Mult8x8
Size	30	134	90	498
Padding Elements	11	86	33	302
Depth (gate levels)	4	8	5	9
Est. Increase In Area	29.1%	48.1%	23 (16)%	39 (25)%
D_{MAX}	4	8	2.0 ns	4.0 ns
ΔD(before)	75%	87.5%	80%	90%
ΔD(after)	0%	0%	5.0%	3.9%
Power (after)	NA	NA	97.7 mW	411.3 mW
Run-Time (uVAX 3200)	NA	NA	0.140 hrs	11.90 hrs
Rough Tuning Run-time	NA	NA	0.01 hrs	0.02 hrs

Table 1: Example Results

6 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 and implemented new algorithms to automatically balance delays in combinational logic circuits. Using our rough tuning algorithm, we insert 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 depending on the available delay elements.

In practice, rough tuning operates in conjunction with fine tuning[11] to design wave-pipelined circuits that have both minimal added area and minimal total power consumption.

Next, we plan to design a test chip demonstrating wave pipelining. We have selected CML as the technology for implementing our sample chip which will be a 32-bit multiplier with possibly some additional datapath logic.

7 Acknowledgements

Mark Horowitz was very helpful in pointing out pitfalls and in particular analyzing technology considerations. Arthur Veinott Jr., Michael Saunders, and Walter Murray were generous in their advice about practical and theoretical considerations of solving optimization problems. Timothy Pinkston reviewed drafts of this paper.

8 Bibliography

1 S. Anderson, J. Earle, R. Goldschmidt, and D. Powers. "The IBM System/360 Model 91 Floating Point Execution Unit." Jan. 1967, IBM Journal of Research and Development, pp. 34-53.

2 L. Cotten. "Maximum Rate Pipelined Systems." 1969 AFIPS Proceedings of Spring Joint Computer Conference, pp. 581-586.

3 C. Desoer and E. Kuh. Basic Circuit Theory, pp. 477-483. 1969, McGraw-Hill(New York, NY).

4 B. Ekroot. Optimization of Pipelined Processors by Insertion of Combinational Logic Delay. Sept. 1987, Ph.D. Dissertation, Electrical Engineering, Stanford University, Stanford, CA.

5 B. Fawcett. Maximal Clocking Rates for Pipelined Digital Systems. Dec. 1975, Report R-706 from Coordinated Science Laboratory, University of Illinois, Urbana, IL.

6 C. Leiserson, F. Rose, and J. Saxe. "Optimizing Synchronous Circuitry by Retiming." 1983, Proceedings of the 3rd CalTech Conference on Very Large Scale Integration.

7 Q. Lin and P. Xia. "The Design and Implementation of a Very Fast Experimental Pipelining Computer." 1988, Journal of Computer Science and Technology, Vol. 3, No. 1 (Beijing).

8 D. Marple. Performance Optimization of Digital VLSI Circuits. Sept. 1986, Ph.D. Dissertation, Electrical Engineering, Stanford University, Stanford, CA.

9 A. Veinott, Jr. Specialist in combinatorial operations research. Private communication in 1989.

10 S. Waser and M. Flynn. Topics in Arithmetic for Digital Systems Designers, p. 88. 1989, Preliminary 2nd edition of: Introduction to Arithmetic for Digital Systems Designers, 1982, Holt, Rinehart, and Winston (New York, NY).

11 D. Wong, G. De Micheli, and M. Flynn. "Designing High-Performance Digital Circuits Using Wave Pipelining." Proceedings of VLSI '89, Munich, W. Germany, August 1989.

12 D. Wong, G. De Micheli, and M. Flynn. "Inserting Active Delay Elements to Achieve Wave Pipelining." 1989, Technical Report CSL-TR-89-386, Electrical Engineering, Stanford University, Stanford, CA.