

# A Novel Delay Balancing Methodology for Wave Pipelined Circuits

Rui Tang

Dept. of Electrical and Computer Engineering  
Northeastern University  
Boston, MA 02115  
rtang@ece.neu.edu

Yong-Bin Kim

Dept. of Electrical and Computer Engineering  
Northeastern University  
Boston, MA 02115  
ybk@ece.neu.edu

**Abstract**—This paper presents a novel effective methodology to balance the delays based on Wong’s algorithm and Klass’s algorithm. In order to reduce the area overhead and the number of delay elements added, a new re-padding technique is introduced and included in this novel delay balancing methodology. Several computational blocks such as adders, multipliers are tested, and the simulation results show that the area overhead is reduced up to the factor of 3 comparing with the previous algorithms.

## I. INTRODUCTION

Wave pipelining is a design method that can be used for high-speed digital systems design[1]. It can achieve higher throughput without inserting registers. Instead, it uses the delay of the combinational logic block itself as a storage element and synchronizes data by equalizing path delays. The minimal clock period, or the maximal frequency, is determined by the difference between maximal and minimal propagation delays of the combinational logic. Therefore to achieve the highest possible wave-pipelining frequency, all path delays from inputs to outputs should be at its best equalized. Timing constraints limit the performance of wave-pipelining circuits, consequently, it is an essential issue in wave pipelined circuits design. In wave pipelined system, the minimum clock period at which a wave pipelined circuit can be clocked is bounded by [4]:

$$t_{clk} > \Delta t_p + 2 * \Delta C + t_{SH} + t_{RF}, \quad (1)$$

where

$t_{clk}$  = clock period

$\Delta t_p$  = maximum difference between longest and shortest path delays over worst-case design, process, and environment

$\Delta C$  = worst case clock skew existing over the whole circuit

$t_{SH}$  = setup plus hold time for edge-triggered registers

$t_{RF}$  = worst-case rise or fall time (10%-90% voltage swing) at the output of the combinational logic block.

In most cases,  $t_{clk}$  can be made very small by reducing the  $\Delta t_p$  to a small fraction of  $t_p$  (the longest path delay)[4]. The path variation  $\Delta t_p$  results from several sources such as fabrication process, coupling capacitance, supply voltage drift and circuit design[3]. In this paper, we concentrate on reducing the delay variations due to circuits design using the balancing techniques.

## II. PREVIOUS RESEARCHES

In order to balance combinational circuits, several algorithms have been proposed in the past few years. The basic idea of these algorithms is modelling the circuits using Directed Acyclic Graph(DAG) and balancing the circuits by equalizing the paths in the DAG. Klass proposed an algorithm to balance the circuits in gate-level[5]. In Klass’s algorithm, the nodes which represent the gates are put into different levels according to the connectivity of the graph. After level weight is set as the maximal weight of the nodes on this level, every path is balanced according to level weights. After delay balancing, the delays of every path from inputs to outputs are equal to the delay of the longest path. However, now the length of the longest path is equal to the sum of every level’s weights, and it may be larger than the length of the original critical path. Therefore some redundant delay elements are added to balance the circuit. A more efficient algorithm is proposed in [4]. In Wong’s algorithm, a spanning tree is constructed and delay elements are added to balance the fundamental loop. Fewer delay elements are added using Wong’s algorithm than Klass’s algorithm. However in Wong’s algorithm, two nodes are used to represent one gate. One node is used to represent the input of the gate and another node is used to represent the output of the gate. Thus Wong’s algorithm uses twice as many nodes as Klass’s algorithm. To almost all the graph algorithms, the computation time is related to the number of nodes and edges. With the increment of the number of nodes and edges, the computation time increases. Therefore, the CPU time of Wong’s algorithm is longer than Klass’s algorithm. In this paper, a novel delay balancing algorithm is proposed based on these two algorithms and a new delay elements re-padding technique is introduced to solve the problems that the previous algorithms have.

## III. PROPOSED ALGORITHM

In this section, the proposed novel delay balancing algorithm is described for deep sub-micron wave pipelined circuits design.

### A. Circuit Modelling Based on Klass’ Algorithm

A weighted Directed Acyclic Graph(DAG) is used to model the circuit. The nodes of the graph represent the gates of the circuit[5].

- $V$  is the set of graph nodes  $v$ . Each  $v \in V$  represents a gate.  $w(v)$  is the weight of a node which corresponds to the delay of a gate.
- $E$  is the set of graph directed edges  $e$ . Each edge  $e \in E$  representing the interconnection of the gates is weighted with a weight  $w(e)$ . Edges' weights represent the wire delays connecting the gates. An edge  $e$  that goes from node  $v_m$  to node  $v_n$  is denoted as  $v_m \rightarrow v_n$ .
- A source and sink node are added to the graph. The source node  $v_0$  is connected to all primary input pins, like  $a[0]$ ,  $b[1]$ . The sink node  $v_N$  is connected to all primary outputs such as  $s[1]$  or  $p[2]$ .
- A path  $p \in G$  is a sequence of nodes and edges. A path  $p$  that starts from node  $v_1$  and ends at node  $v_k$  is denoted as  $v_1 \Rightarrow v_k$ . For any path  $p = v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_k$ , the path delay is defined as the sum of edge weights and node weights along the path:  $w(p) = \sum_{i=1}^{k-1} w(e_i) + \sum_{i=1}^k w(v_i)$ .

In deep sub micron design, wire delay is dominant. Therefore with the pre-knowledge of interconnection between the gates, wire delays can be considered in this gate level delay balancing algorithm. This delay balancing algorithm has two goals. First goal is to minimize the delay difference of every path from primary inputs to primary outputs and the second goal is to minimize the number of delay elements added or to minimize the area overhead. Therefore in the first step, we should find out the gaps in the graph which need to be balanced. Fig 1 shows a conversion example of a small circuit to our graph representation. It can be seen obviously that the delay from  $a \rightarrow p$  is longer than the path delay from  $b \rightarrow p$ . The delay gap between these two paths is equal to the delay of one inverter. To simplify the problem, we assume that the rising delay and the falling delay of one inverter are the same and there is no delay difference between the paths in the gates due to the different input patterns. The delay of inverter and NAND gate are equal to one delay unit and the delay of NOR gate is equal to three delay units. The longest path from primary inputs to primary outputs are  $c \rightarrow q$  and  $d \rightarrow q$  which are equal to 4. The delay of path  $a \rightarrow p$  is equal to 2 and the delay of path  $b \rightarrow p$  is equal to 1. Therefore there is a gap equals to 1 from  $U_0 \rightarrow U_2$  and a gap equals to 2 from  $U_2 \rightarrow U_N$  need to be balanced.

### B. Using Loops to Detect Gaps Based on Wong's Algorithm

Gap is defined as the delay difference between the two paths starting at the same node and ending at the same node. These two paths can compose a basic loop and can be called two sides of the loop. A spanning tree of a connected graph is a subgraph that contains all of the graph's nodes. That is to say a spanning tree of a connected graph has no loop and contains at least one intact side of a loop. However the other side of the loop is broken in the spanning tree, therefore a gap can be detected at that broken point. Hence the first two steps of our algorithm are as follows[4]:

- 1) Constructing a longest-path spanning tree in the DAG beginning from the source.

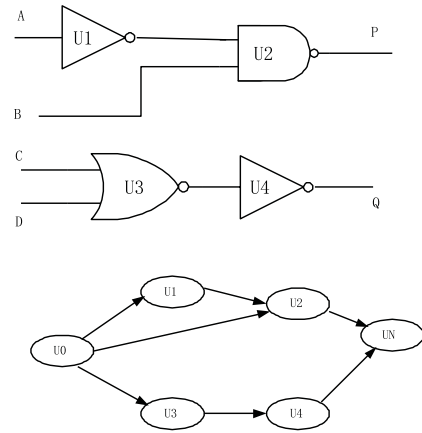


Fig. 1. An example conversion from a circuit to a DAG

- 2) Let A represent the set of edges which are not in the spanning tree, getting edges from A one by one, closing every loop in the spanning tree. If two sides of the loop are balanced, the weight of the gap is equal to zero. Otherwise, the weight of the gap is equal to the length difference of these two sides.

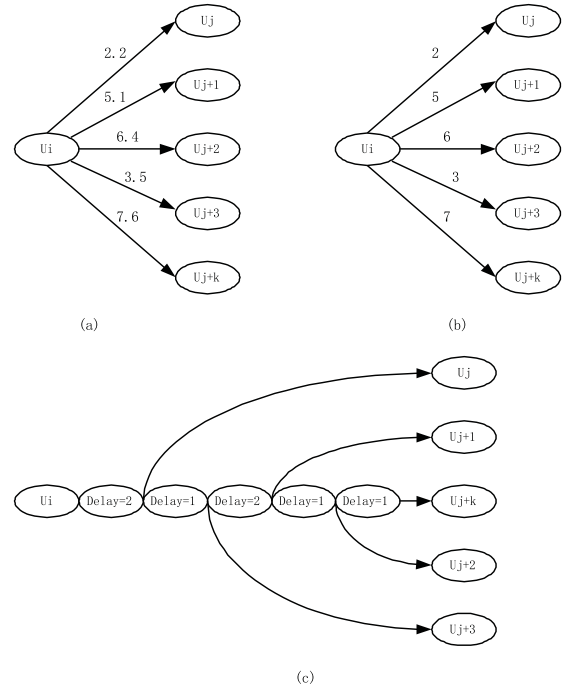


Fig. 2. An example of delay elements sharing

After all the gaps are known, it is time to fill in the gaps with the padding delay elements. Theoretically, if propagation delays of the delay elements have continuous values from zero to infinitely large, any circuits can be balanced perfectly. However in reality, the lengths of padding delay elements are set within a finite range from  $B_{min}$  to  $B_{max}$  and only have the discrete values instead of continuous values. Therefore, the propagation delay difference smaller than  $B_{min}$  can't be bal-

anced in this way. We assume that the minimum delay element is  $B_{min}$  and there are several gaps starting from the same node  $U_i$ . These gaps are named  $gap(i, j)$ ,  $gap(i, j+1)$ ,  $gap(i, j+2)$ , ..... $gap(i, j+k)$ . If the length of the gap is smaller than the minimum delay element  $B_{min}$ , this gap can't be balanced with delay elements, else  $gap(i, j) = C_j \times B_{min} + \text{redundancy}(i, j)$  (where  $\text{redundancy}(i, j) < B_{min}$  which can't be implemented and  $C_j$  is a constant larger than 0). Cutting the length that can't be implemented, the new gap weight  $gap(i, j)'$  becomes  $C_j \times B_{min}$ .

### C. Proposed Delay Elements Sharing and Shifting Techniques

Combining the advantages of Klass's[5] and Wong's[4] algorithm, the following delay element sharing and shifting algorithm is proposed. Gaps starting from the same node can share the delay elements because gaps are times of the minimum delay element (one delay unit). Fig 2 shows an example of delay elements sharing. In the graph, it can be seen that the five gaps starting from  $U_i$  are 7.6, 6.4, 5.1, 3.5, 2.2, assuming the minimum delay element is 1, then the weights of the five gaps become 7, 6, 5, 3, 2 as shown in Fig 2 (b) after the redundant weights are cut. Therefore only the maximum delay gap 7 needs to be implemented, and the gaps to other gates can be obtained from the intermediate points of this longest delay path as shown in Fig 2 (c). The length of the minimum common delay is 2. Delay elements sharing can largely reduce the number of delay elements added. If the minimum common delay can be shifted from the output of one gate to the input of this gate, the number of delay elements can be further reduced. Listed below are some basic definitions:

- Fan-in edge  $FIE_i$  of node  $U_i$  is defined as the incoming edge of one node in the DAG which corresponds to the input wire of one gate in the circuit. Fan-in node  $FIN_i$  of node  $U_i$  is defined as a node which has an incoming edge connected to this node.
- Fan-out edge  $FOE_i$  of node  $U_i$  is defined as the outgoing edge of one node in the DAG which corresponds to the output wire of one gate in the circuit. Fan-out node  $FON_i$  of node  $U_i$  is defined as a node which has an outgoing edge connected to this node.
- Gap is defined as the delay difference of two paths starting from the same node and ending at the same node.  $gap(i, j)$  means there is an edge from  $U_i$  to  $U_j$  in the original graph,  $U_i$  is the fan-in node of  $U_j$  and this edge belongs to an unbalanced path which ends in  $U_j$ .
- Total gap weight equals to  $\sum_{i,j}^N gap(i, j)$
- The minimum common delay  $min_{U_i}$  is defined as the minimum gap starting from  $U_i$ .
- The maximum delay  $max_{U_i}$  is defined as the maximum gap starting from  $U_i$ .
- Local gap weight of gate  $U_i$  is defined as  $local_{U_i}$  which equals to  $max_{U_i} + \sum_{FIN(i)} max_{UFIN(i)}$ , where  $max_{UFIN(i)}$  is the maximum delay of gate  $U_i$ 's fan-in nodes.

In the original graph, if all the fan-out edges of gate  $U_i$  are gaps, the minimum gap can be shifted from the output of  $U_i$  to

the inputs of  $U_i$ . If at least one fan-out edge of gate  $U_i$  is not a gap, shifting can't be operated. In some cases, delay elements shifting can reduce the total weight of delay elements added. Delay elements shifting from the output of  $U_i$  to the inputs of  $U_i$  change the weights of  $U_i$ 's fan-in edges, therefore it may change the weight of longest gap of  $U_i$ 's fan-in nodes. As shown in Fig 3, both  $U_i$  and  $U_j$  are fan-in nodes of  $U_k$  and after delay elements shifting, the weight of the longest fan-out edge of  $U_i$  changes from 3 to 4, however the weight of the longest fan-out edge of  $U_j$  is still equal to 5. And because our goal is to minimize the total weight of delay elements added, whether to do the delay elements shifting or not is based on whether this shifting can reduce the weight of delay elements added or not. If we consider the effectiveness of this single shifting, to simplify the problem, the cost of this shifting can be defined as  $cost_{U_k} = local'_{U_k} - local_{U_k}$ , where  $local_{U_k}$  is the local gap weight of gate  $U_k$  before shifting and  $local'_{U_k}$  is the local gap weight of gate  $U_k$  after shifting. The shifting should be done if  $cost_{U_k} \leq 0$ . Otherwise, the shifting should not be operated.

## IV. THE IMPLEMENTATION OF PROPOSED ALGORITHM

Adder and multiplier are widely used combinational logic blocks. To testify the efficiency of our new algorithm and compare the results with Wong's algorithm and Klass's algorithm, adders and multipliers were chosen as the target designs. For adders, 4,16,32 bit RPL(ripple carry adder), and CLA(carry-look-ahead adder) were run and for multipliers, 4, 8 and 16 bit CSA(carry-save multiplier), NBW(non-booth Wallace tree multiplier) were tested. The whole process of implementing our novel algorithm is as follows:

- 1) Parse the net-list, convert the Verilog code to a DAG:
  - a) Parse the net-list, all the useful information of every gate such as gate's type, gate's inputs, gate's outputs, gate's internal delay, gate's fan-ins, gate's

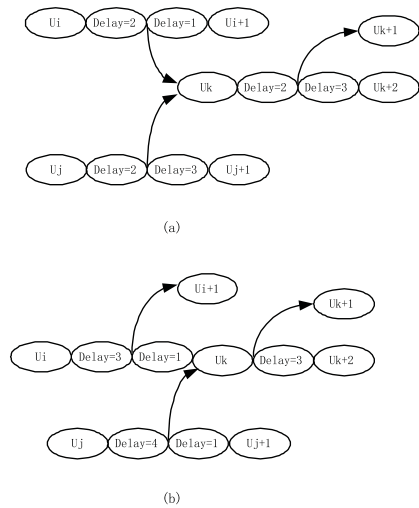


Fig. 3. An example of delay elements sharing

Design	Gate Number	Using The Proposed Algorithm			Using Klass's Algorithm			Using Wong's Algorithm		
		Buffer Added	Max Delay	Area Over-head Ratio	Buffer Added	Max Delay	Area Over-head Ratio	Buffer Added	Max Delay	Area Over-head Ratio
RPL4	74	135	22	45%	316	22	106%	167	22	56%
CLA4	65	92	20	36%	267	20	105%	107	20	42%
RPL16	332	2172	60	144%	5727	60	379%	2277	60	150%
CLA16	272	1940	68	141%	4687	68	340%	2293	68	166%
RPL32	680	8392	106	269%	20611	106	685%	8564	106	275%
CLA32	551	8794	144	272%	18224	144	632%	10271	144	356%
CSA4	146	392	35	55%	865	53	120%	485	35	68%
NBW4	145	418	36	56%	881	50	118%	565	36	76%
CSA16	3328	26419	179	168%	87319	299	554%	51090	179	324%
NBW16	3061	6595	90	51%	21749	144	167%	11948	90	92%

TABLE I  
THE SIMULATION RESULTS

- fan-outs and the local weight of this gate are recorded in a data structure.
- b) Go through every gate's structure, get the connectivity information of the gates. Build the hash table to store these information.
- 2) Perform balancing and re-padding algorithms on the DAG:
- a) Getting a longest path spanning tree T from the source to the sink. When building the spanning tree, the longest path from the sink node to every node is put into the spanning tree while the weight of the longest path  $D_{MAX}$  is recorded. The weights of other paths from the sink node to this node are subtracted from  $D_{MAX}$ , then the weights of gaps are recorded.
  - b) Visiting the nodes in DAG from the sink node to the source node. Before one node is visited, all his children nodes should be visited.
  - c) When visiting the node  $U_k$ ,  $max_{U_k}$  and  $min_{U_k}$  should be recorded while  $local_{U_k}$  and  $local_{U_k}$  should be compared to decide whether to perform the delay elements shifting or not.
- 3) Convert the resulting DAG to a new gate-level Verilog code:
- a) Generating the new gate-level net-list.
  - b) Verifying the function of the new gate-level net-list.

## V. SIMULATION RESULTS

In the previous section, the basic idea of our proposed novel delay balancing algorithm is described. In this section, we compare this algorithm with Wong's algorithm and Klass's algorithm to demonstrate the efficiency of the proposed algorithm. Klass's algorithm and Wong's algorithm are also implemented in a similar way. To compare these gate-level delay balancing algorithms, we assume the delay of NAND gate and the delay of inverter are equal to one delay unit regardless of input patterns and the delay of NOR gate is three times one delay unit. The weight of minimum redundant

delay element is also equal to one delay unit. Therefore, the paths can be perfectly balanced no matter which algorithm is used. However we can compare the performance of these three algorithms by counting the number of delay elements added and estimating the area overhead. The simulation data are shown in Table I. In this table, CSA16 means 16-bit carry-save multiplier. Column IV (Max Delay) records the length of the critical path which is measured by delay units. Through these tables, it can be seen that comparing with Wong's algorithm and Klass's algorithm, the proposed algorithm save area overhead by factor of 2 to 3 for CSA, NBW and CLA. However for ripple-carry adder, the area overhead is reduced only a little bit. The reason is that carry-save multiplier, Wallace multiplier and carry-look-ahead adder have parallel structures and the possibility of delay elements sharing is larger than the circuit having serial structure such as ripple carry adder.

## VI. CONCLUSIONS

This paper presented a novel delay balancing algorithm based on the combination of Wong's algorithm and Klass's algorithm. The re-padding technique implementing the idea of delay elements sharing and shifting reduces the number of delay elements added or the area overhead. The efficiency of this algorithm has been verified based on the simulation results shown in Table I. The proposed algorithm is proved to be a most effective one among the algorithms under testing.

## REFERENCES

- [1] L. Cotten, "Maximum Rate Pipelined Systems," in Proc. AFIPS Spring Joint Comput. Conf., 1969.
- [2] C. Thomas Gray, Wentai Liu and Ralph K. Cavin III, "Wave Pipelining: Theory and CMOS Implementation," Kluwer Academic Publishers, 1994.
- [3] W. P. Burleson, M. Ciesielski, F. Klass and W. Liu, "Wave pipelining: A Tutorial and Research Survey," IEEE Transactions on VLSI systems Volume 6, Issue 3, Sept. 1998 Page(s):464 - 474.
- [4] D. Wong, G. De Micheli and M. Flynn, "Designing High-Performance Digital Circuits Using Wave Pipelining: Algorithms and Practical Experiences," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol.12, No.1, January 1993, pp.25-46.
- [5] F. Klass, "Balancing Circuits for Wave Pipelining," Technical Report:CSL-TR-92-549 October 1992.