Simultaneous PTL Buffer Insertion and Sizing for Minimizing Elmore Delay

I-Min Liu[†] Tai-Hung Liu[†] Hai Zhou[‡] Adnan Aziz[†]
[†]Electrical and Computer Engineering [‡]Computer Sciences
The University of Texas The University of Texas
Austin TX Austin TX

Abstract

For many digital designs, implementation in pass transistor logic (PTL) has been shown to be superior in terms of area, timing, and power characteristics to static CMOS. One problem with PTL circuits is that the delay through serially connected transistors is quadratic in the number of stages. The objective of this work is to develop an automatic PTL buffer insertion and sizing procedure for minimizing Elmore delay.

1 Introduction

For many complex logic functions, implementation in pass-transistor logic has been shown to be superior to static CMOS [1, 2, 3, 4]. Advantages of PTL include reduced area, faster switching, and lower power consumption. However, PTL designs are largely handcrafted, which is a time-consuming process [5]. At the Boolean level, PTL synthesis based on BDDs is effective [6]. However, synthesis based solely on BDDs is limited; it ignores electrical characteristics of the circuit family. For example, the delay of PTL is quadratically dependent on the number of serial stages [5]. Thus buffers must be inserted into a long series of pass transistors; buffer locations need to be chosen so as to minimize delay and the number of buffers inserted.

Existing buffering algorithms only handle interconnection trees [7, 8, 9]; they lack capability of buffering the circuits represented by directed acyclic graph (DAG), such as PTL. While tailoring PTL circuits into trees may fit PTL buffering problems in the conventional scenario, it reduces the solution space for global optimality. In this paper, we propose an efficient algorithm for simultaneous buffer insertion and sizing of PTL circuits and use Elmore delay [10] as the cost function. Experimental results with test cases show that the proposed algorithm can be used effectively for high-speed PTL circuit designs.

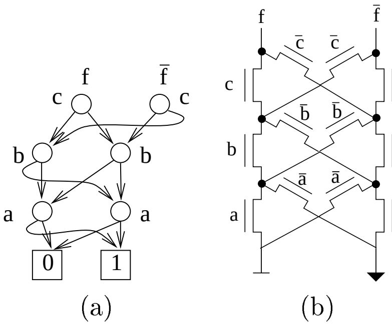

Figure 1: Implementing PTL from BDDs: (a) BDDs for the function $f = a \oplus b \oplus c$ and $\overline{f}$, (b) Mapping BDDs to PTL.

2 Preliminaries

Given a combinational circuit, the primary outputs are Boolean function of the primary inputs. This function can be represented by a single BDD; however it is more efficient to represent it as the composition of a set of BDDs. Afterwards, we convert the BDD to PTL. As shown in Figure 1, there is a mapping from BDD to PTL; a BDD node with a pair of associated edges can be implemented by a pair of pass transistors with complementary control signal, representing a mux. We will call such a pair a *pass transistor pair*. The only pass-transistor circuits we will consider are those defined from BDD. This is because BDD-based PTL is convenient for automated design and has no sneak paths and multiple threshold-voltage loss [6].

We will represent a PTL circuit as DAGs. Specifically, a DAG is an ordered pair (V, E). The vertex set V is the union of the sets SO, SI, IN_P, and IN_B, where SO corresponds to the sources (i.e., V_{DD}/GND), SI corresponds to the sinks (i.e., the primary outputs), IN_P corresponds to pass transistors, and IN_B corresponds

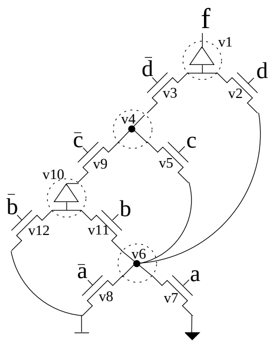

Figure 2: Example of PTL implementing $f = a(b + c + d)$.

to candidate buffer positions in the circuit. (All these sets are assumed to be pairwise disjoint.) In Figure 2, $IN_B = \{v1, v4, v6, v10\}$; $IN_P = \{v2, v3, v5, v7, v8, v9, v11, v12\}$. The set of edges E consists of those vertex pairs (v_i, v_j) for which there is an connection from v_i to v_j.

A *buffering* is a function mapping IN_B to a library B containing buffers of various sizes. After buffering, a candidate buffer node will correspond to a specific buffer in the library. A buffer b is a quadruple (p_b, r_b, c_b, d_b), where p_b is the buffer type (inverting or non-inverting), r_b is the output resistance, c_b is the input capacitance and d_b is the intrinsic delay of the buffer. For convenience, we include a dummy buffer, meaning a wire-or, in the library. When a candidate buffer position corresponds to a dummy buffer, it means no buffer is inserted at that position. In Figure 2, $v4$ and $v6$ are dummy buffers. We will use the following symbolism for a buffered PTL circuit:

- $Child(v)$: Set of node v's children; e.g., $Child(v6) = \{v2, v5, v11\}$.
- $Des(v)$: All pass transistors, buffers, or sinks on the paths from node v to the neighboring downstream buffers or sinks. (This set does not include v itself.); e.g. $Des(v6) = \{v1, v2, v3, v4, v5, v10, v11\}$.
- C_v^{down}: Downstream capacitance seen at the downstream side of node v.
- C_v^{up}: Downstream capacitance seen at the upstream side of node v.
- q_v^{down}: Required arrival time at the downstream side of node v.
- q_v^{up}: Required arrival time at the upstream side of node v.

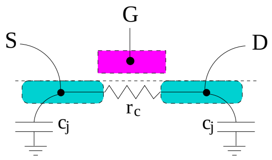

Figure 3: Equivalent RC model for a pass transistor.

2.1 Delay Model

In order to keep the problem tractable, yet realistic, we model a PTL circuit as a distributed switch-level RC network [5] and use Elmore delay model [10] to capture the quadratic delay along a series of pass transistors. Elmore delay model is widely adopted in interconnection trees for delay optimization. It is not obvious that it can be applied to PTL circuits; we will prove this in a later lemma.

Given an input vector i to a design D, we'll refer to the set of pass transistors which are turned on as the *conducting set* corresponding to that i/D. *Non-conducting set* consists of those pass transistor nodes not in the conducting set. Each candidate buffer node is fanined by a pass transistor pair, in which one transistor is turned on and the other is off at any instance of time, since the PTL circuits has been assumed to have been derived from BDDs. Hence, if we remove the non-conducting set from N_P, the induced graph is a forest. For example, in Figure 2, given $i = (a, b, c, d) = (1, 0, 1, 1)$, the corresponding non-conducting set is $\{v3, v8, v9, v11\}$. The induced graph consists of two trees; one contains $\{v10, v12\}$ and the other contains $\{v1, v2, v4, v5, v6, v7\}$. This observation generalizes to the following lemma:

Lemma 2.1 *The graph induced by removing non-conducting set from N_p of the DAG representing PTL is a forest.*

Each pass transistor is modeled as a π-type circuit as shown in Figure 3, where c_j is the equivalent source junction capacitance (which by symmetry is equal to equivalent drain junction capacitance) and r_c is the equivalent channel resistance (which is infinite if the pass transistor is turned off). Given a buffered design, the Elmore delay through a pass transistor is computed by

$$Delay(v) = r_c(c_j + C_v^{down}) \tag{1}$$

Each buffer is modeled by a simple switch-level RC model. The signal delay through a buffer is computed by

$$Delay(v) = d_b + r_b C_v^{down} \tag{2}$$

where d_b is the intrinsic delay and r_b is the output resistance of the buffer.

The capacitance C_v^{down} needs to be computed before the delay at a node v can be calculated. Interestingly, although half of the pass transistors in the circuit are turned off at any instant of time, a node v drives all of the nodes in $Des(v)$ in the worst case. This is illustrated in Figure 2. Although node $v3$ is in non-conducting set corresponding to the input $i = (a, b, c, d) = (1, 1, 1, 1)$, node $v7$ needs to charge the downstream junction capacitance of $v3$ through $v2$ and charge the upstream junction capacitance through $v4$. Thus, node $v7$ drives all the nodes in $Des(v7) = \{v1, v2, v3, v4, v5, v6, v10, v11\}$. This observation generalizes to the following lemma:

Lemma 2.2 *Node v drives all the nodes in $Des(v)$ in the worst case.*

Therefore, we approximate C_v^{down} by

$$C_v^{down} = \sum_{u \in Des(v)} c_u \quad (3)$$

The total capacitance seen at the upstream of a node is computed depending on the type of the node: If v is a pass transistor node,

$$C_v^{up} = 2c_j + C_v^{down} \quad (4)$$

If v is a buffered node, the capacitance at the downstream of the node is isolated by the buffer, and therefore

$$C_v^{up} = c_b \quad (5)$$

where c_b represents the input capacitance of the buffer.

We can recursively compute the required arrival times by setting the required arrival time at sinks as specified; the required arrival time at the downstream of node v is the minimum of the required arrival time at the upstream of the nodes in $Child(v)$:

$$q_v^{down} = \min_{u \in Child(v)} q_u^{up} \quad (6)$$

Given q_v^{down} and $Delay(v)$, the required arrival time at the upstream of v is computed by

$$q_v^{up} = q_v^{down} - Delay(v) \quad (7)$$

2.2 Problem Formulation

We define the simultaneous buffer insertion and sizing problem of a PTL circuit for delay minimization as follows:

```
Labelling(v,B)
    if v is labelled then
        return
    for each u ∈ Child(v) do
        Labelling(u,B)
    S^up = S^down = ∅
    while S^up is ∅ do
        S^down = Merge_Solution (v)
        if v ∈ IN_B then
            S^up = Insert_Buffer(S^down,B)
        else
            S^up = Compute_Delay(S^down,v)
        Prune_Solution(S^up)
        if S^up is ∅ then
            Rearrange_Child(v,B)
    return
```

Figure 4: Pseudocode for `labelling` stage.

Given: A PTL circuit with associated required arrival time at sinks.

Find: Positions and sizes of inserted buffers.

Objective: The minimum required arrival time at the sources is maximized.

The legal positions where the algorithm may insert buffers are the downstream of a pass transistor pair. It is practical to insert the type of buffers that output signal is negated from input since an inverting buffer is superior to a non-inverting buffer in both area and delay consideration. We will show how signal polarity is handled in our algorithm.

3 Simultaneous Buffer Insertion and Sizing

Our procedure draws inspirations from the efficient and optimum dynamic programming procedures for trees, which have been applied to buffer insertion [7] and interconnect sizing [8], etc. Intuitively, we start from the sinks, and work our way back to the sources. As each step, buffering and sizing is done as though the design were tree-like. Whenever there is a conflict in the choice of buffer (which can only happen at points leading to reconvergent fanout), we choose the buffer dictated by the more critical path. Although this can be suboptimal, the procedure is fast and yields high quality solutions.

Below we give a detailed description of the algorithm, which we divide into three stages: labelling, generating, and phase assignment.

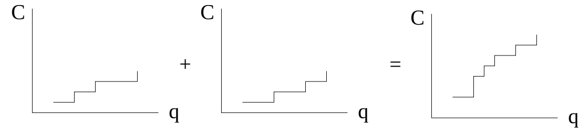

Figure 5: Merge the candidate solutions from children, where q is the required arrival time and C is the total downstream capacitance.

3.1 Labelling Stage

In the labelling stage, the algorithm labels each node in the circuit with candidate solutions starting from the sinks. Each candidate solution is a downstream configuration for the node. This is shown in Figure 4. A candidate solution is a quintuple $S_v = (q_v^{up}, C_v^{up}, b, p, S_{Child(v)})$, where b is a buffer in the library, $S_{Child(v)}$ is the set of candidate solutions for $Child(v)$ that are used to construct S_v, and p represents the phase assignment for the node as we will discuss later. Given a $(b, S_{Child(v)})$ combination, we compute the corresponding C_v^{down} and q_v^{down} by Equations 3 and 6. For a pass transistor node, the delay time is computed by Equation 1. The values for C_v^{up} and q_v^{up} can be computed using Equations 4 and 7. At a candidate buffer position, the algorithm finds the optimal downstream configuration to insert the buffer for each buffer in the library and add it to the set of candidate solutions for the node. In this case, we compute the delay time and C_v^{up} of the node using Equations 2 and 5. After the candidate solutions for a node are computed, a pruning procedure is performed to reduce the solution space.

The solution merge and the pruning procedure follows that in [7]. In the merge step, for each possible required time q_v^{down}, we merge the candidate solutions that minimize $\sum_{u \in Child(v)} C_u^{up}$. This procedure is performed like adding functions [7] as illustrated in Figure 5 and is linear time in the number of candidate solutions to merge from. The pruning procedure does it in the way that no quintuple has strictly smaller required arrival time and larger total capacitance than others. These two techniques are indeed greedy in nature for a DAG, i.e., given a q_v^{down}, the configuration that each nodes in $Child(v)$ has smallest possible total downstream capacitance does not necessarily construct the solution for v that total downstream capacitance at v is minimized; this is due to the possible convergence of the downstream descendents of node v. In practice, one buffer is introduced for every few stages of pass transistors in a circuit such that this convergence property is not so significant. From small examples, we saw that this selection rule gives a C_v^{down} equal or close to the optimal choice while significantly reducing the algorithm's complexity.

```
Generating(v,s^up,B) /* s^up is a solution in S^up */
    if the solution of v is fixed then
        return
    while 1 do
        (u, s_u^up) = Find_Unconfiged_Critical_Child(v,s^up)
        if (u, s_u^up) is null then
            Remove_Unfixed_Solution(v)
            Labelling(v,B)
            break
        else
            Config_Node(u,s_u^up)
            Generating(u,s_u^up,B)
            Remove_Unfixed_Solution(v)
            Labelling(v,B)
    fix the solution of v
    return
```

Figure 6: Pseudocode for `generating` stage.

3.2 Generating Stage

Based on the set of solutions returned for the sources, a unique configuration for each candidate buffer node will be decided in the generating stage; this is shown in Figure 6. In this stage, the algorithm iteratively configures the candidate buffer nodes along the paths from sources to sinks as the way they construct the delay-optimal solution for the sources. If there are paths branching off these source-to-sink paths, the algorithm will iteratively configure these paths until all the paths has been configured. A unique solution will be generated for the node when all of its downstream nodes has been configured. When a unique solution for a node is achieved, we "fix" the node to that configuration and never re-compute the solution again during the generating stage. At the end of the generating stage, each node in the circuit is associated with a unique solution.

Since downstream descendents of a node may converge in a DAG, it happens that the solution at source refers to two distinct candidate solutions for a intermediate node along different paths, indicating two different configurations. As illustrated in Figure 7, two candidate solutions for v_1 (i.e., with/without a buffer inserted) are refereed by the source via paths P_1 and P_{cr}. This causes a conflict when one of the solutions needs to be chosen for the node. Noticing that a previously assigned configuration for noncritical paths may introduce an additional buffer delay for the critical path (e.g., Figure 7(a)), we resolve the conflict by iteratively configuring the paths from

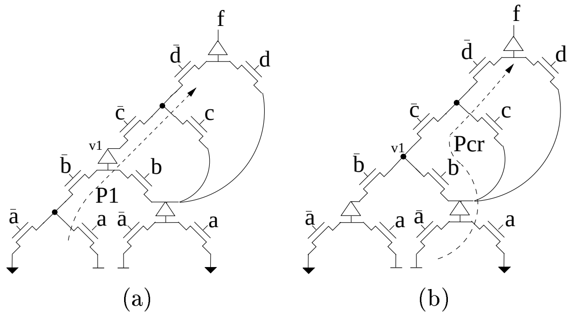

Figure 7: Configure candidate buffer nodes: (a) P_1 first, then P_{cr}, (b) P_{cr} first, then P_1. Here P_{cr} the is critical path.

most to least critical one. As shown in Figure 7(b), the candidate solutions on the critical path are given higher priority. In this way, we try to preserve the optimality for the circuit.

3.3 Phase-Assignment Stage

After the generating stage, each buffer location has a particular buffer from the library. Depending on the node configuration and buffer type, the phase of the output signal of a node may be inverted from the phase of its input signal. Such a node is attributed as an *inverting node*. A node is non-inverting if it is not a inverting node.

We will say the phase assignment of a node is *positive* if it is the same as the assignment of all the sinks; it is *negative* if it is opposite to the sinks; otherwise it is conflicting. A set of nodes is *in-phase* if the signal polarities required at their input are all opposite to (or the same as) the sinks. Clearly, the signal polarity at the input of a node is assignable if all of its children are *in-phase*. For implementability, we require the signal polarity at the input of the sinks are *in-phase*.

Definition 1 (Correct Assignment)
The required signal polarity at the input of assignable node v is assigned be opposite to that of $Child(v)$ if v is an inverting node and is assigned be the same as that of $Child(v)$ if v is a non-inverting node.

To ensure the buffered circuit logically equivalent to its original design, the required signal polarity of each node in the circuit needs to be correctly assigned. During the phase-assignment stage, we assign the incoming signal polarity for each node according to the definition for correct assignment. The required signal polarity is recursively decided based on the configuration of the node and the phase assignment of its children. If the node is non-inverting, the signal polarity of the node is assigned to be the same as its children; otherwise it is assigned to be opposite to its children.

```
Rearrange_Child(v,B)
    if v is V_DD then
        for each u ∈ Child(v) do
            if p_u ≠ p_SI then
                Append_Child(GND,u)
                Remove_Child(v,u)
        Labelling(GND,B)
    else if v is GND then
        for each u ∈ Child(v) do
            if p_u ≠ p_SI then
                Append_Child(V_DD,u)
                Remove_Child(v,u)
        Labelling(V_DD,B)
    else
        v' = Find_Additional_Buffer
        if v' is null then
            v' = Create_Additional_Buffer(v)
        p_cr = Find_Critical_Path_Phase(v)
        for each u ∈ Child(v) do
            if p_u ≠ p_cr then
                Append_Child(v',u)
                Remove_Child(v,u)
        Labelling(v,B)
    return
```

Figure 8: Pseudocode for `phase-assignment` stage.

It is likely that the children require different signal polarities from a node, leading to what we called a "conflicting node". To resolve the conflict, an additional buffered node is introduced and connected at the output of the conflicting node. Based on the required signal polarity, the children are divided into two groups. We identify the group that is less critical in timing, remove them from the child-set of the conflicting node and add them to that of the new buffer. In this way, we may utilize the new buffer to share the load from the conflicting node without directly contributing to the delay time on the critical path. Unlike an intermediate node, additional buffers are not introduced to resolve conflicting V_{DD} and GND since they usually have large fanout that an additional buffer is not sufficient to drive. Those children driven by V_{DD} who require the signal polarity opposite to the sinks are disconnected from V_{DD} and connected to GND. Those children driven by GND who require signal polarity opposite to the sinks are disconnected from GND and connected to V_{DD}. This procedure is shown in Figure 8. In this way, the conflict is resolved without increasing the timing cost.

4 Experimental Results

We have implemented the simultaneous buffer insertion and sizing (SBIS) algorithm under the C/Linux environment on a Pentium-II/266MHz machine. This algorithm is tested on a set of PTL circuits derived from BDDs for MCNC benchmarks [12]. The transistor parameters are based on a $0.5\mu m$ technology [13]. We consider V_{DD} and GND as ideal voltage sources and ignore interconnection effects.

For comparison, we have also implemented a heuristic-based buffer insertion algorithm. More specifically, it inserts buffers such that no node has fanouts more than a constant. It also inserts buffers such that no unbuffered serial chain of pass transistors exceeds a fixed threshold. Table 1 compares SBIS with such a heuristic-based algorithm. In this test, we only allowed one candidate buffer size since the heuristic-based algorithm does not have buffer-sizing capability. The experiment shows that the SBIS algorithm returns favorable results in terms of the delay time while the number of inserted buffers is comparable to that of the heuristic-based algorithm.

Table 2 shows the number of inserted buffers for the SBIS algorithm. We tested the circuits by changing the number of candidate buffers in the library as follows: (a) one candidate buffer (b) three candidate buffers (c) five candidate buffers. The corresponding buffer sizes for the three cases are shown in the table. Our results show that, when more candidate buffers are included in the library, while the total number of inserted buffers may increase, the total buffer area (See Table 3) is actually decreased. This is because the algorithm tends to use more smaller buffers and only insert large buffers at the nodes driving large fanouts when necessary. Table 3 summarizes the performance of the SBIS algorithm. We compare the delay times, total buffer areas, and CPU times for the three cases. Not surprisingly, the delay times are reduced when the algorithm performs optimization with addition flexibility; in many test samples, the difference is substantial. Running times were small in all cases. This efficiency is needed because buffer insertion and sizing are required on a regular basis due to the quadratic delay along long series of pass transistors.

5 Conclusion

In this paper, we report a simultaneous buffer insertion and sizing algorithm for PTL optimization. For algorithmic complexity, we use Elmore delay model to capture the quadratic delay along serial pass transistors and the required time at input of the circuit as a cost function. Preliminary experiments performed indicated the performance gain from simultaneous considering of buffer position and size can be substantial. In the future, we plan to take other important electrical effects into account, namely the threshold voltage loss and the nonlinearity of pass transistors, as well as the shape of the input waveform. We feel that the existing PTL synthesis procedures lack a well-defined characterization of electrical effects of the circuits. We believe we can contribute to an enhanced understand of these issues.

References

[1] K. Yano, T. Yamnaka, T. Nishida, M. Satio, K. Shimohigashi, and A. Shimizu, "A 3.8-ns CMOS 16×16-b Multiplier Using Complementary Pass-Transistor Logic," *IEEE Journal of Solid-State Circuits*, vol. 25, pp. 388–395, Apr. 1990.

[2] F. S. Lai and W. Hwang, "Differential Cascode Voltage Switch with Pass-Gate (DCVSPG) Logic Tree for High Performance CMOS Digital Systems," *International Symposium on VLSI Technology, Systems, and Applications*, pp. 358–362, 1993.

[3] A. Parameswar, H. Hara, and T. Sakurai, "A High Speed, Low Power, Swing Restored Pass-Transistor Logic Based Multiply and Accumulate Circuit for Multimedia Applications," *Proc. Custom Integrated Circuits Conf.*, pp. 278–281, May 1994.

[4] T. S. Sheungm and K. Asada, "Regenerative Pass-Transistor Logic: A Circuit Technique for High Speed Digital Design," *IEICE Trans. Electron.*, vol. E79-C, pp. 1274–1283, Sept. 1996.

[5] N. H. E. Weste and K. Eshragian, *Principles of CMOS VLSI Design*. Addison-Wesley, 1993.

[6] P. Buch, A. Narayan, A. R. Newton, and A. Sangiovanni-Vincentelli, "Logic Synthesis for Large Pass Transistor Circuits," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 663–670, Nov. 1997.

[7] L. P. P. P. van Ginneken, "Buffer Placement in Distributed RC-tree Networks for Minimal Elmore Delay," in *Proc. Intl. Symposium on Circuits and Systems*, pp. 865–868, 1990.

[8] J. Lillis, C.-K. Cheng, and T.-T. Y. Lin, "Optimal Wire Sizing and Buffer Insertion for Low Power and a Generalized Delay Model," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 138–143, 1995.

[9] G. E. Tellez and M. Sarrfzadeh, "Minimal Buffer Insertion in Clock Trees with Skew and Slew Rate Constraints," in *IEEE Trans. Computer-Aided Design*, pp. 333–342, 1997.

[10] W. C. Elmore, "The Transient Response of Damped Linear Network with Particular Regard to Wideband Amplifier," in *Journal of Applied Physics*, pp. 55–63, 1948.

[11] E. Detjens, G. G. R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Technology mapping in mis," in *Proc. Intl. Conf. on Computer-Aided Design*, pp. 116–119, 1987.

[12] S. Yang, *Logic Synthesis and Optimization Benchmarks User Guide*. Microelectronics Center of North Carolina, 1991.

[13] Microelectronics Center of North Carolina, *MCNC Designers' Manual*.

		SBIS		Heuristic-based	
Benchmark	#Levels	Delay(ns)	#Buffers	Delay(ns)	#Buffers
C432	36	8.08	992	8.55	708
C880	43	15.7	9837	16.5	7788
C5135	47	8.80	1298	9.06	1285
C7552	133	23.4	5365	23.9	6386
apex6	20	4.04	484	4.56	413
apex7	22	4.14	243	4.32	216
c8	12	1.69	27	1.69	53
dalu	34	8.60	467	9.37	342
des	19	19.4	4578	19.5	5137

Table 1: Comparison between SBIS and the heuristic-based algorithm before the phase-assignment stage when only one candidate buffer size is allowed. Delay times are computed using Elmore delay.

#Candidate buffers	1	3			5				
Buffer size	4	2	4	6	1	2	4	6	8
C432	1397	1047	335	38	741	529	170	12	0
C880	13511	9948	2863	233	7420	4100	1418	62	48
C5135	1821	1469	321	16	1006	726	183	6	0
C7552	6473	5578	748	78	2329	3258	424	27	2
apex6	589	519	139	14	462	223	61	2	8
apex7	329	307	66	6	279	97	38	2	2
c8	33	65	4	8	77	10	4	4	4
dalu	578	495	62	10	390	188	18	2	6
des	5897	4831	1171	240	3749	2015	659	62	2

Table 2: Numbers of inserted buffers obtained by the SBIS algorithm correspond to: (a) one candidate buffer (b) three candidate buffers (c) five candidate buffers in the library. The buffer sizes are represented by the ratio of the transistor width of the buffer with respect to that of the minimum-size buffer.

	Elmore Delay(ns)			Buffer area(μm^2)			CPU(s)		
Benchmarks	1	3	5	1	3	5	1	3	5
C432	7.10	6.00	5.83	8382	5493	3825	7.3	8.4	11.3
C880	13.2	9.31	8.55	81066	49119	33048	94.9	131.5	211.0
C5135	8.47	7.61	7.43	10926	6477	4839	32.5	36.6	47.2
C7552	23.5	19.9	19.3	38838	21924	16078.5	282.1	291.8	359.6
apex6	4.09	2.72	2.05	3534	2517	1842	16.2	17.3	26.7
apex7	3.39	2.80	2.51	1974	1371	979.5	6.5	7.0	17.7
c8	1.92	1.51	1.33	198	291	253.5	0.8	0.9	1.1
dalu	6.60	4.36	3.72	1734	1728	1347	8.2	9.3	12.2
des	19.9	9.92	6.73	35382	23679	16204.5	224.8	263.9	344.1

Table 3: Delay times, total areas of inserted buffers, and CPU times obtained by the SBIS algorithm correspond to: (a) one candidate buffer (b) three candidate buffers (c) five candidate buffers in the library.