Effective and Efficient Approach for Power Reduction by Using Multi-Bit Flip-Flops

Ya-Ting Shyu, Jai-Ming Lin, Chun-Po Huang, Cheng-Wu Lin, Ying-Zu Lin, and Soon-Jyh Chang, *Member, IEEE*

***Abstract*— Power has become a burning issue in modern VLSI design. In modern integrated circuits, the power consumed by clocking gradually takes a dominant part. Given a design, we can reduce its power consumption by replacing some flip-flops with fewer multi-bit flip-flops. However, this procedure may affect the performance of the original circuit. Hence, the flip-flop replacement without timing and placement capacity constraints violation becomes a quite complex problem. To deal with the difficulty efficiently, we have proposed several techniques. First, we perform a co-ordinate transformation to identify those flip-flops that can be merged and their legal regions. Besides, we show how to build a combination table to enumerate possible combinations of flip-flops provided by a library. Finally, we use a hierarchical way to merge flip-flops. Besides power reduction, the objective of minimizing the total wirelength is also considered. The time complexity of our algorithm is $\Theta(n^{1.12})$ less than the empirical complexity of $\Theta(n^2)$. According to the experimental results, our algorithm significantly reduces clock power by 20–30% and the running time is very short. In the largest test case, which contains 1 700 000 flip-flops, our algorithm only takes about 5 min to replace flip-flops and the power reduction can achieve 21%.**

***Index Terms*— Clock power reduction, merging, multi-bit flip-flop, replacement, wirelength.**

I. Introduction

DUE to the popularity of portable electronic products, low power system has attracted more attention in recent years. As technology advances, an systems-on-a-chip (SoC) design can contain more and more components that lead to a higher power density. This makes power dissipation reach the limits of what packaging, cooling or other infrastructure can support. Reducing the power consumption not only can enhance battery life but also can avoid the overheating problem, which would increase the difficulty of packaging or cooling [1], [2]. Therefore, the consideration of power consumption in complex SOCs has become a big challenge to designers. Moreover, in modern VLSI designs, power consumed by clocking has taken a major part of the whole design especially for those designs using deeply scaled CMOS technologies [3]. Thus, several methodologies [4], [5] have been proposed to reduce the power consumption of clocking.

Manuscript received February 1, 2011; revised August 22, 2011; accepted February 16, 2012. Date of publication April 5, 2012; date of current version March 18, 2013. This work was supported in part by the National Science Council of Taiwan under Grant 100-2220-E-006-005.

The authors are with the Department of Electrical Engineering, National Cheng-Kung University, Tainan 70101, Taiwan (e-mail: kkttkkk@sscas.ee.ncku.edu.tw; jmlin@ee.ncku.edu.tw; gppo@sscas.ee.ncku.edu.tw; lcw@sscas.ee.ncku.edu.tw; tibrius@gmail.com; soon@mail.ncku.edu.tw).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2012.2190535

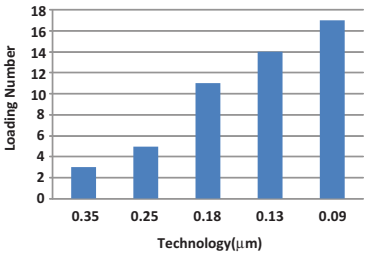


Fig. 1. Maximum loading number of a minimum-sized inverter of different technologies (rising time 250 ps).

Given a design that the locations of the cells have been determined, the power consumed by clocking can be reduced further by replacing several flip-flops with multi-bit flip-flops. During clock tree synthesis, less number of flip-flops means less number of clock sinks. Thus, the resulting clock network would have smaller power consumption and uses less routing resource.

Besides, once more smaller flip-flops are replaced by larger multi-bit flip-flops, device variations in the corresponding circuit can be effectively reduced. As CMOS technology progresses, the driving capability of an inverter-based clock buffer increases significantly. The driving capability of a clock buffer can be evaluated by the number of minimum-sized inverters that it can drive on a given rising or falling time. Fig. 1 shows the maximum number of minimum-sized inverters that can be driven by a clock buffer in different processes. Because of this phenomenon, several flip-flops can share a common clock buffer to avoid unnecessary power waste. Fig. 2 shows the block diagrams of 1- and 2-bit flip-flops. If we replace the two 1-bit flip-flops as shown in Fig. 2(a) by the 2-bit flip-flop as shown in Fig. 2(b), the total power consumption can be reduced because the two 1-bit flip-flops can share the same clock buffer.

However, the locations of some flip-flops would be changed after this replacement, and thus the wirelengths of nets connecting pins to a flip-flop are also changed. To avoid violating the timing constraints, we restrict that the wirelengths of nets connecting pins to a flip-flop cannot be longer than specified values after this process. Besides, to guarantee that a new flip-flop can be placed within the desired region, we also need to consider the area capacity of the region. As shown in Fig. 3(a), after the two 1-bit flip-flops f_1 and f_2 are replaced by the 2-bit flip-flop f_3, the wirelengths of nets net_1, net_2, net_3, and net_4 are changed. To avoid the timing violation caused by the replacement, the Manhattan distance of new nets net_1, net_2, net_3, and net_4 cannot be longer than the specified values.

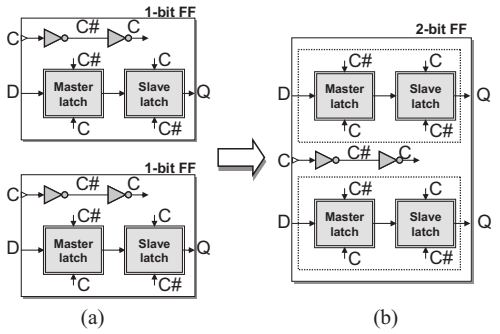


Fig. 2. Example of merging two 1-bit flip-flops into one 2-bit flip-flop. (a) Two 1-bit flip-flops (before merging). (b) 2-bit flip-flop (after merging).

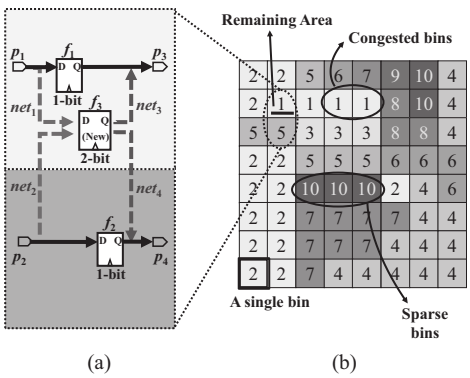


Fig. 3. (a) Combination of flip-flops possibly increases the wire length. (b) Combination of flip-flops also changes the density.

In Fig. 3(b), we divide the whole placement region into several bins, and each bin has an area capacity denoting the remaining area that additional cells can be placed within it.

Suppose the area of f_3 is 7 and f_3 is assigned to be placed in the same bin as f_1. We cannot place f_3 in that bin since the remaining area of the bin is smaller than the area of f_3. In addition to the considerations mentioned in the above, we also need to check whether the cell library provides the type of the new flip-flop. For example, we have to check the availability of a 3-bit flip-flop in the cell library when we desire to replace 1- and 2-bit flip-flops by a 3-bit flip-flop.

A. Related Work

Chang *et al.* [6] first proposed the problem of using multi-bit flip-flops to reduce power consumption in the post-placement stage. They use the graph-based approach to deal with this problem. In a graph, each node represents a flip-flop. If two flip-flops can be replaced by a new flip-flop without violating timing and capacity constraints, they build an edge between the corresponding nodes. After the graph is built, the problem of replacement of flip-flops can be solved by finding an m-clique in the graph. The flip-flops corresponding to the nodes in an m-clique can be replaced by an m-bit flip-flop. They use the branch-and-bound and backtracking algorithm [8] to find all m-cliques in a graph. Because one node (flip-flop) may belong to several m-cliques (m-bit flip-flop), they use greedy heuristic algorithm to find the maximum independent set of cliques, which every node only belongs to one clique, while finding m-cliques groups. However, if some nodes correspond to k-bit flip-flops that $k \geqq 1$, the bit width summation of flip-flops corresponding to nodes in an m-clique, j, may not equal m. If the type of a j-bit flip-flop is not supported by the library, it may be time-wasting in finding impossible combinations of flip-flops.

B. Our Contributions

The difficulty of this problem has been illustrated in the above descriptions. To deal with this problem, the direct way is to repeatedly search a set of flip-flops that can be replaced by a new multi-bit flip-flop until none can be done. However, as the number of flip-flops in a chip increases dramatically, the complexity would increase exponentially, which makes the method impractical. To handle this problem more efficiently and get better results, we have used the following approaches.

1) To facilitate the identification of mergeable flip-flops, we transform the coordinate system of cells. In this way, the memory used to record the feasible placement region can also be reduced.
2) To avoid wasting time in finding impossible combinations of flip-flops, we first build a combination table before actually merging two flip-flops. For example, if a library only provides three kinds of flip-flops, which are 1-, 2-, and 3-bit, we first separate the flip-flops into three groups. Therefore, the combination of 1- and 3-bit flip-flops is not considered since the library does not provide the type of 4-bit flip-flop.
3) We partition a chip into several subregions and perform replacement in each subregion to reduce the complexity. However, this method may degrade the solution quality. To resolve the problem, we also use a hierarchical way to enhance the result.

The rest of this paper is organized as follows. Section II describes the problem formulation. Section III presents the proposed algorithm. Section IV evaluates the computation complexity. Section V shows the experimental results. Finally, we draw a conclusion in Section VI.

II. Problem Formulation

Before giving our problem formulation, we need the following notations.

1) Let f_i denote a flip-flop and b_i denote its bit width.
2) Let $A(f_i)$ denote the area of f_i.
3) Let $P(f_i)$ denote all the pins connected to f_i.
4) Let $M(p_i, f_i)$ denote the **Manhattan distance** between a pin p_i and f_i, where p_i is an I/O pin that connects to f_i.
5) Let $S(p_i)$ denote the constraint of maximum wirelength for a net that connects to a pin p_i of a flip-flop.
6) Given a placement region, we divide it into several bins [see Fig. 3(b) for example], and each bin is denoted by B_k.

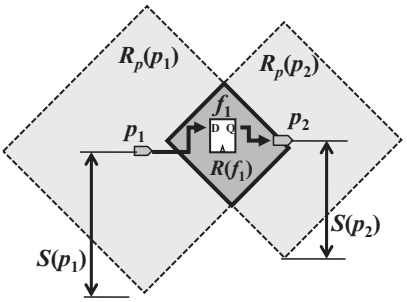


Fig. 4. Defined slack region of the pin.

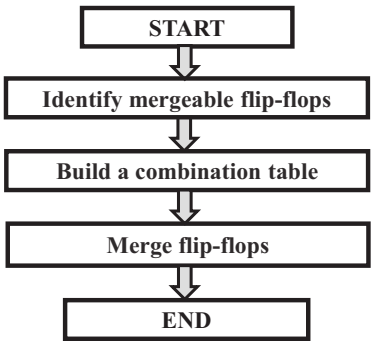


Fig. 5. Flow chart of our algorithm.

7) Let $RA(B_k)$ denote the remaining area of the bin B_k that can be used to place additional cells.
8) Let L denote a cell library which includes different flip-flop types (i.e., the bit width or area in each type is different).

Given a cell library L and a placement which contains a lot of flip-flops, our target is to merge as many flip-flops as possible in order to reduce the total power consumption. If we want to replace some flip-flops $f_{1,\ldots,}f_{j-1}$ by a new flip-flop f_j, the bit width of f_j must be equal to the summation of bit widths in the original ones (i.e., $\Sigma b_i = b_j$, $i = 1$ to $j-1$). Besides, since the replacement would change the routing length of the nets that connect to a flip-flop, it inevitably changes timing of some paths. Finally, to ensure that a legalized placement can be obtained after the replacement, there should exist enough space in each bin. To consider these issues, we define two constraints as follows.

1) Timing Constraint for a Net Connecting to a Flip-Flop f_j from a Pin p_i: To avoid that timing is affected after the replacement, the Manhattan distance between p_i and f_j cannot be longer than the given constraint $S(p_i)$ defined on the pin p_i [i.e., $M(p_i, f_j) \leq S(p_i)$].

Based on each timing constraint defined on a pin, we can find a feasible placement region for a flip-flop $\boldsymbol{f_j}$. See Fig. 4 for example. Assume pins p_1 and p_2 connect to a 1-bit flip-flop f_1. Because the length is measured by Manhattan distance, the feasible placement region of f_1 constrained by the pin p_i [i.e., $M(p_i, f_1) \leq S(p_i)$] would form a diamond region, which is denoted by $R_p(p_i)$, $i = 1$ or 2. See the region enclosed by dotted lines in the figure. Thus, the legal placement region of f_1 would be the overlapping region enclosed by solid lines, which is denoted by $R(f_1)$. $R(f_1)$ is the overlap region of $R_p(p_1)$ and $R_p(p_2)$.

2) Capacity Constraint for Each Bin $\boldsymbol{B_k}$: The total area of flip-flops intended to be placed into the bin B_k cannot be larger than the remaining area of the bin B_k (i.e., $\Sigma A(f_i) \leq RA(B_k)$).

III. Our Algorithm

Our design flow can be roughly divided into three stages. Please see Fig. 5 for our flow. In the beginning, we have to identify a **legal placement region** for each flip-flop f_i. First, the feasible placement region of a flip-flop associated with different pins are found based on the timing constraints defined on the pins. Then, the legal placement region of the flip-flop f_i can be obtained by the overlapped area of these regions. However, because these regions are in the diamond shape, it is not easy to identify the overlapped area. Therefore, the overlapped area can be identified more easily if we can transform the coordinate system of cells to get rectangular regions. In the second stage, we would like to build a **combination table,** which defines all possible combinations of flip-flops in order to get a new multi-bit flip-flop provided by the library. The flip-flops can be merged with the help of the table. After the legal placement regions of flip-flops are found and the combination table is built, we can use them to **merge flip-flops**. To speed up our program, we will divide a chip into several bins and merge flip-flops in a local bin. However, the flip-flops in different bins may be mergeable. Thus, we have to combine several bins into a larger bin and repeat this step until no flip-flop can be merged anymore.

In this section, we would detail each stage of our method. In the first subsection, we show a simple formula to transform the original coordination system into a new one so that a legal placement region for each flip-flop can be identified more easily. The second subsection presents the flow of building the combination table. Finally, the replacements of flip-flops will be described in the last subsection.

A. Transformation of Placement Space

We have shown that the shape of a feasible placement region associated with one pin p_i connecting to a flip-flop f_i would be diamond in Section II. Since there may exist several pins connecting to $\boldsymbol{f_i}$, the legal placement region of $\boldsymbol{f_i}$ are the overlapping area of several regions. As shown in Fig. 6(a), there are two pins p_1 and p_2 connecting to a flip-flop f_1, and the feasible placement regions for the two pins are enclosed by dotted lines, which are denoted by $R_p(p_1)$ and $R_p(p_2)$, respectively. Thus, the legal placement region $R(f_1)$ for f_1 is the overlapping part of these regions. In Fig. 6(b), $R(f_1)$ and $R(f_2)$ represent the legal placement regions of f_1 and f_2. Because $R(f_1)$ and $R(f_2)$ overlap, we can replace f_1 and f_2 by a new flip-flop f_3 without violating the timing constraint, as shown in Fig. 6(c).

However, it is not easy to identify and record feasible placement regions if their shapes are diamond. Moreover, four coordinates are required to record an overlapping region [see Fig. 7(a)]. Thus, if we can rotate each segment 45°, the

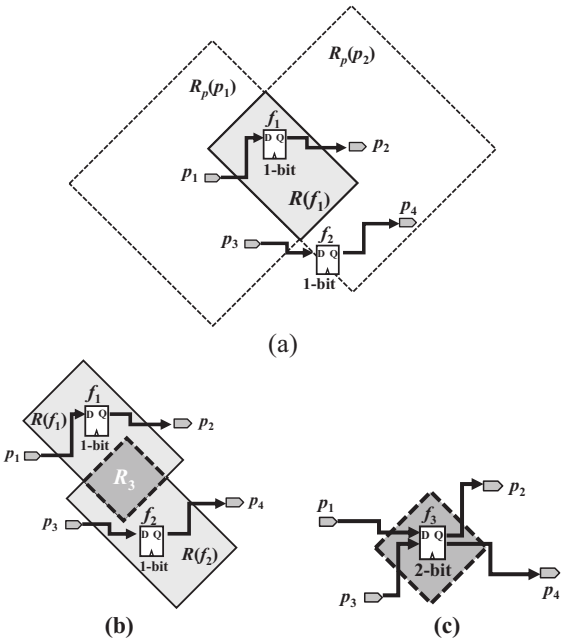


Fig. 6. (a) Feasible regions $R_p(p_1)$ and $R_p(p_2)$ for pins p_1 and p_2 which are enclosed by dotted lines, and the legal region $R(f_1)$ for f_1 which is enclosed by solid lines. (b) Legal placement regions $R(f_1)$ and $R(f_2)$ for f_1 and f_2, and the feasible area R_3 which is the overlap region of $R(f_1)$ and $R(f_2)$. (c) New flip-flop f_3 that can be used to replace f_1 and f_2 without violating timing constraints for all pins p_1, p_2, p_3, and p_4.

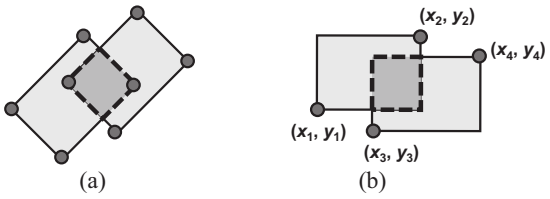


Fig. 7. (a) Overlapping region of two diamond shapes. (b) Rectangular shapes obtained by rotating the diamond shapes in (a) by 45°.

shapes of all regions would become rectangular, which makes identification of overlapping regions become very simple. For example, the legal placement region, enclosed by dotted lines in Fig. 7(a), can be identified more easily if we change its original coordinate system [see Fig. 7(b)]. In such condition, we only need two coordinates, which are the left-bottom corner and right-top corner of a rectangle, as shown in Fig. 7(b), to record the overlapped area instead of using four coordinates.

The equations used to transform coordinate system are shown in (1) and (2). Suppose the location of a point in the original coordinate system is denoted by (x, y). After coordinate transformation, the new coordinate is denoted by (x', y'). In the original transformed equations, each value needs to be divided by the square root of 2, which would induce a longer computation time. Since we only need to know the relative locations of flip-flops, such computation are ignored in our method. Thus, we use x'' and y'', to denote the coordinates of transformed locations

$$x' = \frac{x+y}{\sqrt{2}} => x'' = \sqrt{2} \cdot x' = x + y \tag{1}$$

$$y' = \frac{-x+y}{\sqrt{2}} => y'' = \sqrt{2} \cdot y' = -x + y. \tag{2}$$

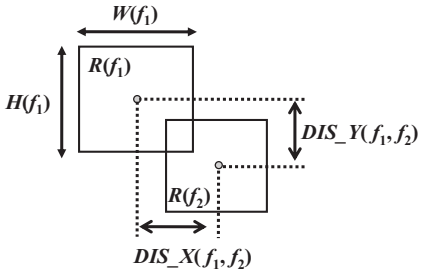


Fig. 8. Overlapping relation between available placement regions of f_1 and f_2.

Then, we can find which flip-flops are mergeable according to whether their feasible regions overlap or not. Since the feasible placement region of each flip-flop can be easily identified after the coordinate transformation, we simply use (3) and (4) to determine whether two flip-flops overlap or not

$$\text{DIS\_}X(f_1, f_2) < \frac{1}{2}(W(f_1) + W(f_2)) \tag{3}$$

$$\text{DIS\_}Y(f_1, f_2) < \frac{1}{2}(H(f_1) + H(f_2)) \tag{4}$$

where $W(f_1)$ and $H(f_1)$ [$W(f_2)$ and $H(f_2)$] denote the width and height of $R(f_1)$ [$R(f_2)$], respectively, in Fig. 8, and the function DIS_$X(f_1, f_2)$ and (DIS_Y(f_1, f_2)) calculates the distance between centers of $R(f_1)$ and $R(f_2)$ in x-direction (y-direction).

B. Build a Combination Table

If we want to replace several flip-flops by a new flip-flop f_i' (note that the bit width of f_i' should equal to the summation of bit widths of these flip-flops), we have to make sure that the new flip-flop f_i' is provided by the library $\boldsymbol{L}$ when the feasible regions of these flip-flops overlap. In this paper, we will build a combination table, which records all possible combinations of flip-flops to get feasible flip-flops before replacements. Thus, we can gradually replace flip-flops according to the order of the combinations of flip-flops in this table. Since only one combination of flip-flops needs to be considered in each time, the search time can be reduced greatly. In this subsection, we illustrate how to build a combination table.

The pseudo code for building a combination table $\boldsymbol{T}$ is shown in Algorithm 1. We use a binary tree to represent one combination for simplicity. Each node in the tree denotes one type of a flip-flop in $\boldsymbol{L}$. The types of flip-flops denoted by leaves will constitute the type of the flip-flop in the root. For each node, the bit width of the corresponding flip-flop equals to the bit width summation of flip-flops denoted by its left and right child [please see Fig. 9(e) for example]. Let n_i denote one combination in $\boldsymbol{T}$, and $b(n_i)$ denote its bit width. In the beginning, we initialize a combination n_i for each kind of flip-flops in $\boldsymbol{L}$ (see Line 1). Then, in order to represent all combinations by using a binary tree, we may add **pseudo types**, which denote those flip-flops that are not provided by the library, (see Line 2). For example, assume that a library only supports two kinds of flip-flops whose bit widths are 1 and 4, respectively. In order to use a binary tree to denote a

Algorithm 1 **Build Combination Table.**

```
1  T = InitializationCombinationTable(L);
2  InsertPseudoType(L);
3  SortByBitNumber (L);
4  for each n_i in T do
5     InsertChildrens (n_i, NULL, NULL);
6  index = 0;
7  while index != size(T) do
8        range_first = index;
9        range_second = size(T);
10       index = size(T);
11       for each n_i in T
12            for j = 1 to range_first do TypeVerify(n_i, n_j, T);
13            for j = i to range_second do TypeVerify(n_i, n_j, T);
14 T = DuplicateCombinationDelete(T);
15 T = UnusedCombinationDelete(T);

InsertPseudoType(L):
1  for i = (b_min+1) to (b_max-1)
2    if (L does not contain a type whose bit width is equal to i )
3      insert a pseudo type type_j with bit width i to L;

InsertChildrens(n, n_1, n_2):
1  n.left_child ← n_1;
2  n.right_child ← n_2;

TypeVerify(n_1, n_2, T):
1  b_sum = b(n_1) + b(n_2);
2  if (L contains a type whose bit width is b_sum)
3    insert a new combination n whose bit width b_sum to T;
4    InsertChildrens( n , n_1, n_2);
```

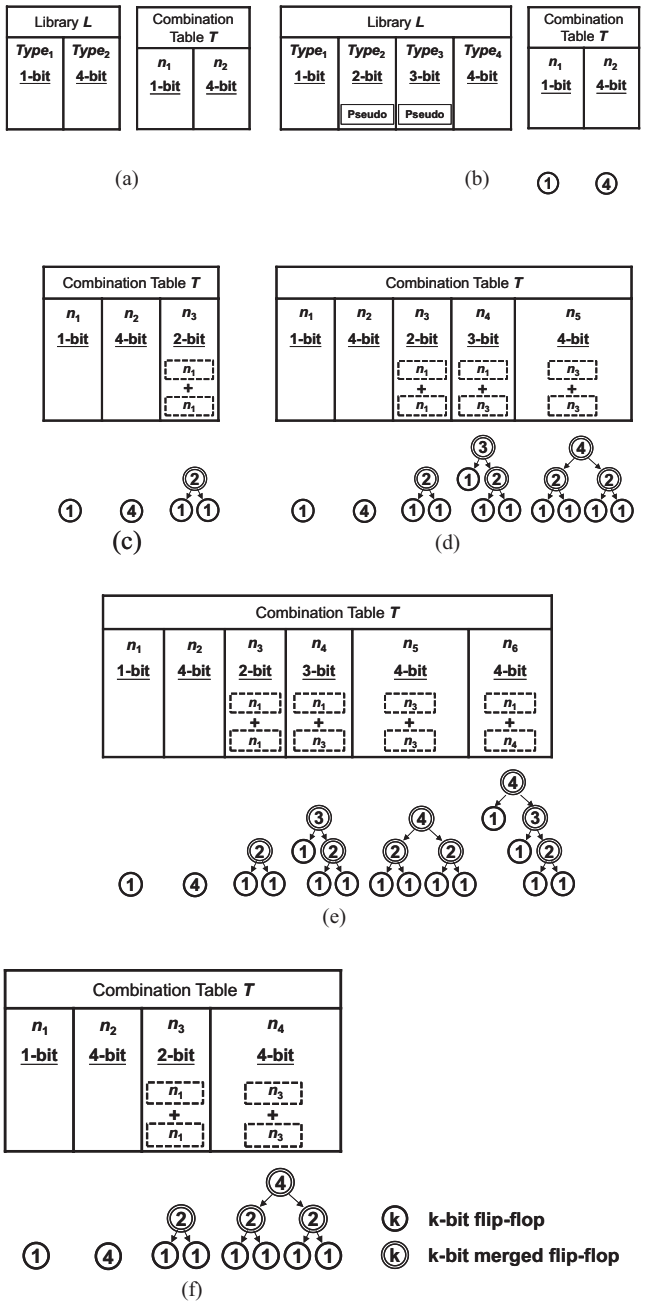


Fig. 9. Example of building the combination table. (a) Initialize the library L and the combination table T. (b) Pseudo types are added into L, and the corresponding binary tree is also build for each combination in T. (c) New combination n_3 is obtained from combining two n_1s. (d) New combination n_4 is obtained from combining n_1 and n_3, and the combination n_5 is obtained from combining two n_3s. (e) New combination n_6 is obtained from combining n_1 and n_4. (f) Last combination table is obtained after deleting the unused combination in (e).

combination whose bit width is 4, there must exist flip-flops whose bit widths are 2 and 3 in $\boldsymbol{L}$ [please see the last two binary trees in Fig. 9(e) for example]. Thus, we have to create two pseudo types of flip-flops with 2- and 3-bit if $\boldsymbol{L}$ does not provide these flip-flops. Function **InsertPseudoType** in algorithm 1 shows how to create pseudo types. Let $b_{\max}$ and $b_{\min}$ denote the maximum and minimum bit width of flip-flops in $\boldsymbol{L}$. In **InsertPseudoType**, it inserts all flip-flops whose bit widths are larger than $b_{\min}$ and smaller than $b_{\max}$ into $\boldsymbol{L}$ if they are not provided by $\boldsymbol{L}$ originally. After this procedure, all combinations in $\boldsymbol{L}$ are sorted according to their bit widths in the ascending order (Line 3). At present, all combinations are represented by binary trees with 0-level. Thus, we would assign NULL to its right and left child (see Lines 4 and 5). Finally, for every two kinds of combinations in $\boldsymbol{T}$, we try to combine them to create a new combination (Lines 6–13). If the new combination is the flip-flop of a feasible type (this can be checked by the function **TypeVerify),** we would add it to the table $\boldsymbol{T}$. In the function **TypeVerify**, we first add the bit widths of the two combinations together and store the result in b_{sum} (see Line 1 in **TypeVerify**). Then, we will add a new combination n to $\boldsymbol{T}$ with bit width b_{sum} if $\boldsymbol{L}$ has such kind of a flip-flop. After these procedures, there may exist some duplicated or unused combinations in $\boldsymbol{T}$. Thus, we have to delete them from the table and the two functions **DuplicateCombinationDelete** and **UnusedCombinationDelete** are called for the purpose (Lines 14 and 15). In **DuplicateCombinationDelete**, it checks whether the duplicated combinations exist or not. If the duplicated combinations exist, only the one with the smallest height of its corresponding binary tree is left and the others are deleted. In **UnusedCombinationDelete**, it checks the combinations whose corresponding type is pseudo

Algorithm 2 Insert Pseudo Types (optional)

InsertPseudoType(L):
1 **for each** $type_j$ in $\boldsymbol{L}$ **do**
2 **PseudoTypeVerifyInsertion(** $type_j$, L**)** ;

PseudoTypeVerifyInsertion($type_j$, L):
1 **if** (mod ($b(type_j)$ /2) == 0)
2 $b_1 = [b(type_j)/2]$, $b_2 = [b(type_j)/2]$;
3 **else**
4 $b_1 = \lfloor b(type_j)/2 \rfloor$, $b_2 = b(type_j) - \lfloor b(type_j)/2 \rfloor$;
5 **for** i = 1 to 2
6 **if (**($b_i > b_{min}$) &&
($\boldsymbol{L}$ does not contain a type whose bit width is equal to b_i))
7 insert a pseudo type $type_j$ with bit width b_i to $\boldsymbol{L}$;
8 **PseudoTypeVerifyInsertion**($type_j$, $\boldsymbol{L}$);

type in $\boldsymbol{L}$. If the combination is not included into any other combinations, it will be deleted.

For example, suppose a library $\boldsymbol{L}$ only provides two types of flip-flops, whose bit widths are 1 and 4 (i.e., $b_{\min} = 1$ and $b_{\max} = 4$), in Fig. 9(a). We first initialize two combinations n_1 and n_2 to represent these two types of flip-flops in the table $\boldsymbol{T}$ [see Fig. 9(a)]. Next, the function **InsertPseudoType** is performed to check whether the flip-flop types with bit widths between 1 and 4 exist or not. Thus, two kinds of flip-flop types whose bit widths are 2 and 3 are added into $\boldsymbol{L}$, and all types of flip-flops in $\boldsymbol{L}$ are sorted according to their bit widths [see Fig. 9(b)]. Now, for each combination in $\boldsymbol{T}$, we would build a binary tree with 0-level, and the root of the binary tree denotes the combination. Next, we try to build new legal combinations according to the present combinations. By combing two 1-bit flip-flops in the first combination, a new combination n_3 can be obtained [see Fig. 9(c)]. Similarly, we can get a new combination n_4 (n_5) by combining n_1 and n_3(two n_3's) [see Fig. 9(d)]. Finally, n_6 is obtained by combing n_1 and n_4. All possible combinations of flip-flops are shown in Fig. 9(e). Among these combinations, n_5 and n_6 are duplicated since they both represent the same condition, which replaces four 1-bit flip-flops by a 4-bit flip-flop. To speed up our program, n_6 is deleted from $\boldsymbol{T}$ rather than n_5 because its height is larger. After this procedure, n_4 becomes an unused combination [see Fig. 9(e)] since the root of binary tree of n_4 corresponds to the pseudo type, $type_3$, in $\boldsymbol{L}$ and it is only included in n_6. After deleting n_6, n_4 is also need to be deleted. The last combination table $\boldsymbol{T}$ is shown in Fig. 9(f).

In order to enumerate all possible combinations in the combination table, all the flip-flops whose bit widths range between $b_{\max}$ and $b_{\min}$ and do not exist in $\boldsymbol{L}$ should be inserted into $\boldsymbol{L}$ in the above procedure. However, this is time consuming. To improve the running time, only some types of flip-flops need to be inserted. There exist several choices if we want to build a binary tree corresponding to a type $type_j$. However, the complete binary tree has the smallest height. Thus, for building a binary tree of a certain combination n_i whose type is $type_j$, only the flip-flops whose bit widths

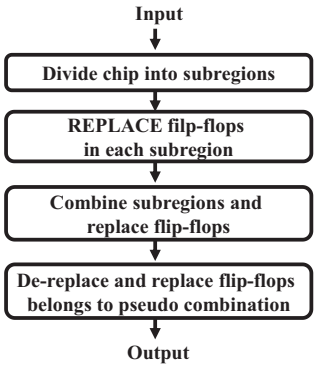


Fig. 10. Detailed flow to merge flip-flops.

are ($\lfloor b(type_j)/2 \rfloor$) and ($b(type_j) - \lfloor b(type_j)/2 \rfloor$) should exist in $\boldsymbol{L}$. Algorithm 2 shows the enhanced procedure to insert flip-flops of pseudo types. For each $type_j$ in $\boldsymbol{L}$, the function **PseudoTypeVerifyInsertion** recursively checks the existence of flip-flops whose bit widths around $\lfloor b(type_j)/2 \rfloor$ and add them into $\boldsymbol{L}$ if they do not exist (see Lines 1 and 2). In the function **PseudoTypeVerifyInsertion**, it divides the bit width $b(type_j)$ into two parts $\lfloor b(type_j)/2 \rfloor$ and $\lfloor b(type_j)/2 \rfloor$ ($\lfloor b(type_j)/2 \rfloor$ and $b(type_j) - \lfloor b(type_j)/2 \rfloor$) if $b(type_j)$ is an even (odd) number (see Lines 1–4 in **PseudoTypeVerifyInsertion**), and it would insert a pseudo type $type_j$ into $\boldsymbol{L}$ if the type is not provided by $\boldsymbol{L}$ and its bit width is larger than the minimum bit width (denoted by $b_{\min}$) of flip-flops in $\boldsymbol{L}$ (see Lines 5–8 in **PseudoTypeVerifyInsertion**). The same procedure repeats in the new created type. Note that this method works only when the 1-bit type exists in $\boldsymbol{L}$. We still have to insert pseudo flip-flops by the function **InsertPseudoType** in Algorithm 1 if the 1-bit flip-flop is not provided by $\boldsymbol{L}$.

For example, assume a library $\boldsymbol{L}$ only provides two kinds of flip-flops whose bit widths are 1 and 7. In the new procedure, it first adds two pseudo types of flip-flops whose bit widths are 3 and 4, respectively, for the flip-flop with 7-bit (i.e., $\boldsymbol{L}$ becomes [1 3 4 7]). Next, the flip-flop whose bit width is 2 is added to $\boldsymbol{L}$ for the flip-flop with 4-bit (i.e., $\boldsymbol{L}$ becomes [1 2 3 4 7]). For the flip-flop with 3-bit, the procedure stops because flop-flops with 1 and 2 bits already exist in $\boldsymbol{L}$. In the new procedure, we do not need to insert 5- and 6-bit pseudo types to $\boldsymbol{L}$.

C. Merge Flip-Flops

We have shown how to build a combination table in Section III-B. Now, we would like to show how to use the combination table to combine flip-flops in this subsection. To reduce the complexity, we first divide the whole placement region into several subregions, and use the combination table to replace flip-flops in each subregion. Then, several subregions are combined into a larger subregion and the flip-flops are replaced again so that those flip-flops in the neighboring subregions can be replaced further. Finally, those flip-flops with pseudo types are deleted in the last stage because they are not provided by the supported library. Fig. 10 shows this flow.

1) Region Partition (Optional): To speed up our problem, we divide the whole chip into several subregions. By suitable

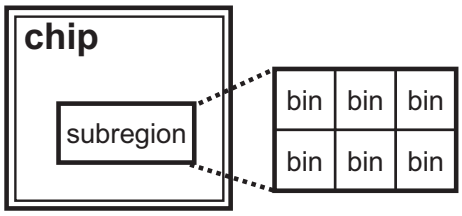


Fig. 11. Example of region partition with six bins in one subregion.

Algorithm 3 Merge all flip-flops in subregion $\boldsymbol{r}$.

```
1  for each combination n in a combination table
2    l_right ← the list for the combination of the right-child of n;
3    l_left  ← the list for the combination of the left-child of n;
4    for each  flip-flop f_i in the list l_left
5      c_best ← ∞
6      for each  flip-flop f_j in the list l_right
7        if ( f_i and f_j can be merged) then
8          Compute cost c;
9          if c < c_best then c_best = c, f_best = f_j;
10       end
11     end
12     Add flip-flop f' merged from f_i and f_best to combination n;
13     Remove f_i from l_left;
14     Remove the flip-flop recorded by f_best from l_right;
15   end
16 end
```

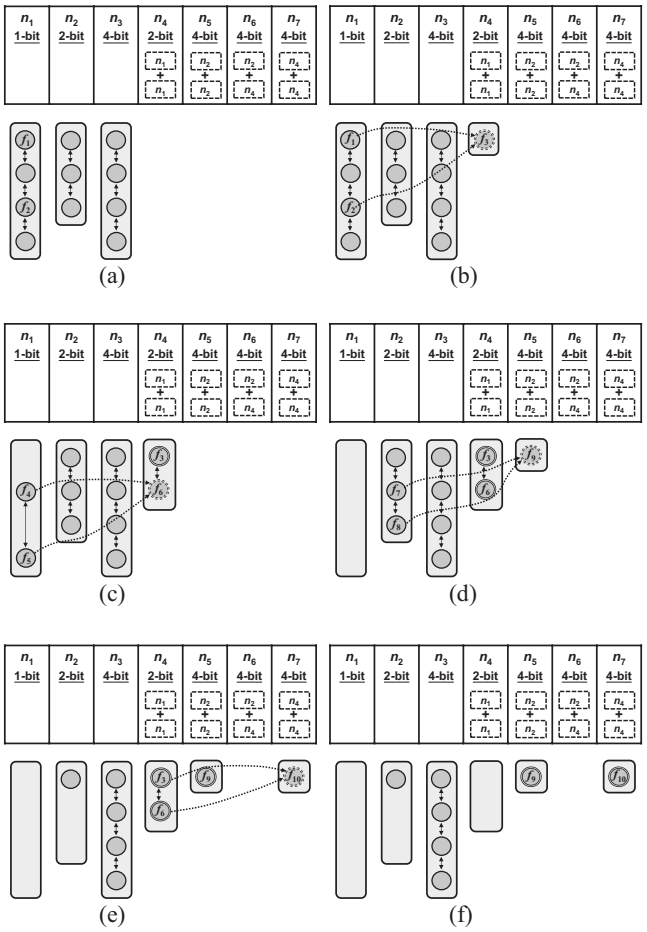


Fig. 12. Example of replacements of flip-flops. (a) Sets of flip-flops before merging. (b) Two 1-bit flip-flops, f_1 and f_2, are replaced by the 2-bit flip-flop f_3. (c) Two 1-bit flip-flops, f_4 and f_5, are replaced by the 2-bit flip-flop f_6. (d) Two 2-bit flip-flops, f_7 and f_8, are replaced by the 4-bit flip-flop f_9. (e) Two 2-bit flip-flops, f_3 and f_6, are replaced by the 4-bit flip-flop f_{10}. (f) Sets of flip-flops after merging.

partition, the computation complexity of merging flip-flops can be reduced significantly (the related quantitative analysis will be shown in Section V). As shown in Fig. 11, we divide the region into several subregions, and each subregion contains six bins, where a bin is the smallest unit of a subregion.

2) Replacement of Flip-flops in Each Subregion: Before illustrating our procedure to merge flip-flops, we first give an equation to measure the quality if two flip-flops are going to be replaced by a new flip-flop as follows:

$$\text{cost} = \text{routing\_length} - \alpha \times \sqrt{\text{available\_area}} \qquad (5)$$

where routing_length denotes the total routing length between the new flip-flop and the pins connected to it, and available_area represents the available area in the feasible region for placing the new flip-flop. α is a weighting factor (the related analysis of the value α will be shown in Section V). The cost function includes the term routing_length to favor a replacement that induces shorter wirelength. Besides, if the region has larger available space to place a new flip-flop, it implies that it has higher opportunities to combine with other flip-flops in the future and more power reduction. Thus, we will give it a smaller cost. Once the flip-flops cannot be merged to a higher-bit type (as the 4-bit combination n_4 in Fig. 9), we ignore the *available_area* in the cost function, and hence α is set to 0.

After a combination has been built, we will do the replacements of flip-flops according to the combination table. First, we link flip-flops below the combinations corresponding to their types in the library. Then, for each combination $\boldsymbol{n}$ in $\boldsymbol{T}$, we serially merge the flip-flops linked below the left child and the right child of $\boldsymbol{n}$ from leaves to root. Algorithm 3 shows the procedure to get a new flip-flop corresponding to the combination n. Based on its binary tree, we can find the combinations associated with the left child and right child of the root. Hence, the flip-flops in the lists, named l_{left} and l_{right}, linked below the combinations of its left child and its right child are checked (see Lines 2 and 3). Then, for each flip-flop $\boldsymbol{f_i}$ in l_{left}, the best flip-flop f_{best} in l_{right}, which is the flip-flop that can be merged with $\boldsymbol{f_i}$ with the smallest cost recorded in c_{best}, is picked. For each pair of flip-flops in the respective list, the combination cost [based on (5)] is computed if they can be merged and the pair with the smallest cost is chosen (see Lines 4–11). Finally, we add a new flip-flop $\boldsymbol{f'}$ in the list of the combination n and remove the picked flip-flops which constitutes the $\boldsymbol{f'}$ (see Lines 12–14).

For example, given a library containing three types of flip-flops (1-, 2-, and 4-bit), we first build a combination table $\boldsymbol{T}$ as shown in Fig. 12(a). In the beginning, the flip-flops with various types are, respectively, linked below n_1, n_2, and n_3 in

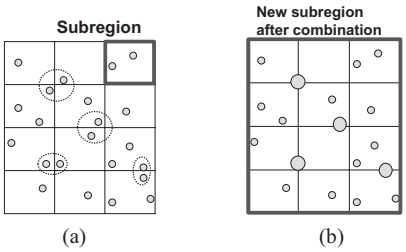


Fig. 13. Combination of flip-flops near subregion boundaries. (a) Result of replace flip-flops in each subregion. (b) Result of replace flip-flops in each new subregion which is obtained from combining twelve subregion in (a).

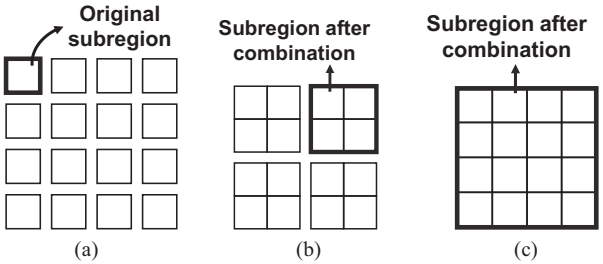


Fig. 14. Combination of subregions to a larger one. (a) Placement is originally partitioned into 16 subregions for replacement. (b) Subregion bounded by bold line is obtained from combining four neighboring subregions in (a). (c) Subregion bounded by bold line is obtained from combining four subregions in (b).

$\boldsymbol{T}$ according to their types. Suppose we want to form a flip-flop in n_4, which needs two 1-bit flip-flops according to the combination table. Each pair of flip-flops in n_1 are selected and checked to see if they can be combined (note that they also have to satisfy the timing and capacity constraints described in Section II). If there are several possible choices, the pair with the smallest cost value is chosen to break the tie. In Fig. 12(a), f_1 and f_2 are chosen because their combination gains the smallest cost. Thus, we add a new node f_3 in the list below n_4, and then delete f_1 and f_2 from their original list [see Fig. 12(b)]. Similarly, f_4 and f_5 are combined to obtain a new flip-flop f_6, and the result is shown in Fig. 12(c). After all flip-flops in the combinations of 1-level trees (n_4 and n_5) are obtained as shown in Fig. 12(d), we start to form the flip-flops in the combinations of 2-level trees (n_6, and n_7). In Fig. 12(e), there exist some flip-flops in the lists below n_2 and n_4, and we will merge them to get flip-flops in n_6 and n_7, respectively. Suppose there is no overlap region between the couple of flip-flops in n_2 and n_4. It fails to form a 4-bit flip-flop in n_6. Since the 2-bit flip-flops f_3 and f_6 are mergeable, we can combine them to obtain a 4-bit flip-flop f_{10} in n_7. Finally, because there exists no couple of flip-flops that can be combined further, the procedure finishes as shown in Fig. 12(f).

If the available overlap region of two flip-flops exists, we can assign a new one to replace those flip-flops. Once there is sufficient space to place the new flip-flop in the available region, the algorithm will perform the replacement, and the new generated flip-flop will be placed in the grid that makes the wirelength between the flip-flop and its connected pins smallest. If the capacity constraint of the bin, B_k, which the grid belongs to will be violated after the new flip-flop is placed on that grid, we will search the bins near B_k to find a new available grid for the new flip-flop. If none of bins which are overlapped with the available region of new flip-flop can satisfy the capacity constraint after the placement of new flip-flop, the program will stop the replacement of the two flip-flops.

3) Bottom-Up Flow of Subregion Combinations (Optional): As shown in Fig. 13(a), there may exist some flip-flops in the boundary of each subregion that cannot be replaced by any flip-flop in its subregion. However, these flip-flops may be merged with other flip-flops in neighboring subregions as shown in Fig. 13(b). Hence, to reduce power consumption further more, we can combine several subregions to obtain a larger subregion and perform the replacement again in the new subregion again. The procedure repeats until we cannot achieve any replacement in the new subregion. Fig. 14 gives an example for this hierarchical flow. As shown in Fig. 14(a), suppose we divide a chip into 16 subregions in the beginning. After the replacement of flip-flops is finished in each subregion, four subregions are combined to get a larger one as shown in Fig. 14(b). Suppose some flip-flops in new subregions still can be replaced by new flip-flops in other new subregions, we would combine four subregions in Fig. 14(b) to get a larger one as shown in Fig. 14(c) and perform the replacement in the new subregion again. As the procedure repeats in a higher level, the number of mergeable flip-flops gets fewer. However, it would spend much time to get little improvement for power saving. To consider this issue, there exists a trade-off between power saving and time consuming in our program.

4) De-Replace and Replace (Optional): Since the pseudo type is an intermediate type, which is used to enumerate all possible combinations in the combination table $\boldsymbol{T}$, we have to remove the flip-flops belonging to pseudo types. Thus, after the above procedures have been applied, we would perform de-replacement and replacement functions if there exists any flop-flops belonging to a pseudo type. For example, if there still exists a flip-flop, f_i, belonging to n_3 after replacements in Fig. 9(f), we have to de-replace f_i into two flip-flops originally belongs to n_1. After de-replacing, we will do the replacements of flip-flops according to $\boldsymbol{T}$ without consideration of the combinations whose corresponding type is pseudo in $\boldsymbol{L}$.

IV. Computation Complexity

This section analyzes the timing complexity of this algorithm. The core is to continuously seek suitable combinations, and find the optimized solution among all possibilities. Hence, the timing complexity depends on the operation count of the function of deciding whether two flip-flops can combine together or not. For example, assume all flip-flops are of the same type, 1-bit flip-flop. In the beginning, each flip-flop will try to combine with all the other flip-flops. If the first flip-flop finds the best solution, the two 1-bit flip-flops will form a 2-bit flip-flop and be removed from the list. Then, the second flip-flop will perform identical procedures. Let N represent the number of flip-flops per circuit. For an exhaustive run for all the 1-bit cells, the timing complexity is $O(N^2)$. If the

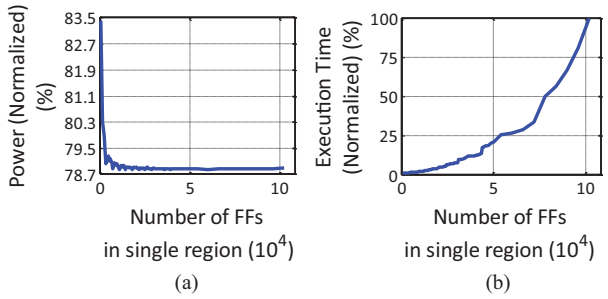


Fig. 15. (a) Influence of the region size on power. (b) Influence of the region size on execution time.

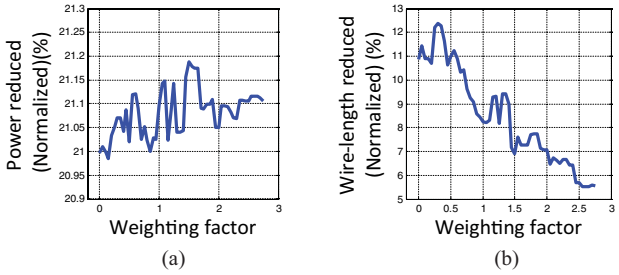


Fig. 16. (a) Influence of the weighting factor on power reduction. (b) Influence of the weighting factor on wirelength reduction.

largest flip-flop the library provided is M-bit, the size of the combination table is $O(M\log_2(M))$ when we use pseudo type flip-flops. The total timing complexity is $O(M\log_2(M) \times N^2)$, equivalently equal to $O(N^2)$ because the value of M is much less than the value of N.

V. Experimental Results

This section shows our experimental results. We implemented our algorithm in C++ language, and all experiments were ran on workstation with a 3.33-GHz Intel Core i7-980X processor with 16-GB memory. Our experiment can be divided into two parts. In the first part, we compare our method with Chang *et al.* [6] and the results are shown in the first subsection. However, some conditions cannot be verified by their test cases. Thus, we provide another set of test cases and the experiment results are shown in the second subsection.

A. Performance Comparison With Chang et al. [6]

In this subsection, we first compare the experimental results with [6]. They used six test cases which were provided by Faraday corporation [7]. Table I shows the information of test cases. The numbers of flip-flops range from 98 to 169 200, and the available types (i.e., 1-, 2-, and 4-bit) of flip-flops in all cases are the same. Table I shows the number of flip-flops in each type in the initial condition.

In our algorithm, there exist two values which would affect our results: the first one is the dimension of a subregion since we would partition a chip into several subregions. The second one is the parameter used in the cost function of (5). Thus, we first do some experiments to explore better values for these two parameters. The results for comparisons with [6] will be shown in the last part of this subsection.

TABLE I
Industry Benchmark Circuits

Circuit	Number of 1-bit FFs	Number of 2-bit FFs	Number of 4-bit FFs
c1	76	22	0
c2	366	57	0
c3	1464	228	0
c4	4378	751	0
c5	9150	1425	0
c6	146400	22800	0

TABLE II
Experimental Results of [6] and Our Approach

Circuit	Approach in [6]			Our approach		
	PR_Ratio (%)	WR_Ratio (%)	Times (s)	PR_Ratio (%)	WR_Ratio (%)	Times (s)
c1	14.8	0.917	0.01	15.9	0.928	0.07
c2	16.9	0.947	0.04	18.0	0.934	0.12
c3	17.1	0.948	0.10	17.8	0.928	0.24
c4	16.8	0.945	0.28	17.6	0.932	0.84
c5	17.1	0.949	0.60	17.8	0.936	1.51
c6	17.2	0.949	78.92	17.9	0.938	30.43
Comp.	0.95	1.01	2.41	1.00	1.00	1.00

1) Influence of Region Size on Performance: In this part, we first determine a suitable size for each subregion during partitioning. Since the execution time is actually dominated by the average number of flip-flops included in a subregion, we use the number of flip-flops in a single subregion to represent the size of a subregion, which can be obtained from multiplying the number of bins in a subregion by the average number of flip-flops in a bin. Fig. 15 shows the simulation results using the circuit ***c6*** in Table I. We sweep the number of flip-flops included in a subregion to observe its effect on power consumption and execution time. The y-axis in Fig. 15(a) and (b), respectively, represent the power reduction and timing improvement ratios relative to the size of a subregion. While a subregion gets larger, the execution time becomes longer. However, the power consumption does not decrease proportionally. On the contrary, if the subregion size becomes very small, the power consumption will increase significantly. To balance execution time and power consumption, we select 600 as the number of flip-flops in a single subregion (the normalized power and execution time are about 83% and 0.8% if the number of flip-flops in a single subregion is 600 in Fig. 15).

2) Influence of Weighting Factor α on Performance: Since the parameter α used by (5) (see Section III-C.2) would affect our results, it is necessary to find a suitable value for getting better results. Similarly, we use circuit ***c6*** to test our program, and the simulation result is shown in Fig. 16. In this experiment, we sweep α from 0 to 3 to get the data of power consumption and wirelength. The y-axis in Fig. 16(a) and (b) respectively represents the wirelength reduction ratio and the power reduction ratio. While the value of α becomes

TABLE III
EXPERIMENTAL RESULTS UNDER DIFFERENT CONDITIONS

	Case 1	Case 2	Case 3	Case 4	Case 5
Library	1, 2, 4	1, 2, 4, 4, 8	1, 2, 4, 6, 13	1, 2, 4, 8	1, 2, 4, 8
Flip-flop number	120	953	5524	60 000	1 728 000
$\text{Power}_{\text{ori}}$ (unit 10^3)	12	95	552	6000	172 800
$\text{Power}_{\text{merged}}$ (unit 10^3)	9	67	430	4208	136 509
PR_Ratio *(%)*	20.97	28.80	22.11	29.87	21.00
WL_{ori} (unit 10^3)	83	577	3563	53 625	1 199 304
WL_{merged} (unit 10^3)	71	506	2189	31 008	1 068 961
WR_Ratio *(%)*	85.62	87.77	61.44	57.82	89.13
Times (s)	0.08	0.24	1.07	36.7	2377
Times of parser	0.07	0.15	0.29	3.8	2153

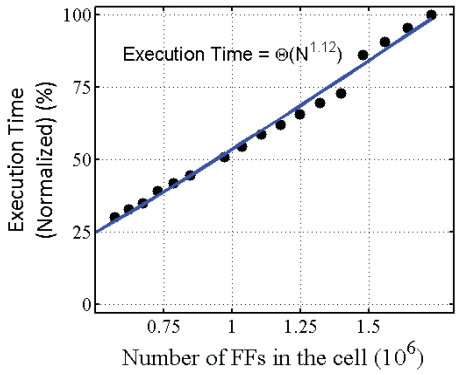


Fig. 17. Average computational complexity of our algorithm.

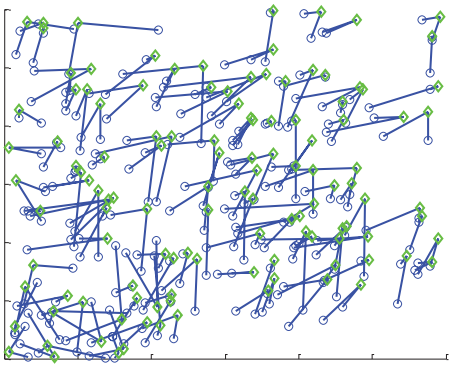


Fig. 18. Distribution of flip-flops in the original design (120 flip-flops, power = 12 000, wirelength = 83 285).

larger, the power reduction ratio gets larger. If α is close to 0, the wirelength reduction ratio will be better than the power reduction ratio. To balance wirelength reduction and power reduction, we use the curves to select a suitable value for α. Because the variation of α has the more apparent effect on wirelength reduction than power reduction, the value of α close to 0 is preferred. In the following experiments, we select 0.1 as the value of α.

3) Comparison Results: The comparison results between [6] and our approach are listed in Table II. Column 1 lists the names of benchmark circuits. In [6], their algorithm was implemented on 2.66-GHz Intel i7 PC under the Linux operation system, and our algorithm was implemented on a 3.33-GHz Intel Core i7-980X processor with 16-GB memory. In Table II, we compare the results of PR_Ratio, WR_Ratio and execution times with [6]. The comparison results are listed in row 8. The values PR_Ratio and WR_Ratio can be computed by the following equations:

$$\text{PR\_Ratio}(\%) = \frac{\text{power}_{\text{original}} - \text{power}_{\text{merged}}}{\text{power}_{\text{original}}} \cdot 100\%$$

$$\text{WR\_Ratio}(\%) = \frac{\text{wire\_length}_{\text{merged}}}{\text{wire\_length}_{\text{original}}} \cdot 100\%$$

where the $\text{power}_{\text{merged}}$ and $\text{wire\_length}_{\text{merged}}$ are the measured power and wirelength after the program is applied, and the $\text{power}_{\text{original}}$ and $\text{wire\_length}_{\text{original}}$ are the measured power and wirelength of the original test case. As shown in Table II, our results of PR_Ratio, WR_Ratio and execution time are all better than the results in [6]. Our execution time of cases with number of flip-flops smaller than about 10 000 is larger than [6], because we have to spend additional time to build the combination table. However, with the help of the combination table, our experimental results of the execution time of *c6* (about 170 000 flip-flops) is much less than [6].

B. Average-Case Performance

In this subsection, we provide another set of cases supported by [9] to test our program. The content of test circuits and experimental results are shown in Table III. Compared to the cases in Table I, the available types of flip-flops are different from Cases 1 to 5. Case 5 is the largest circuit of about 1 700 000 flip-flops. Because the execution time is dominated by the number of flip-flops in the circuit, Case 5 is applied to help to demonstrate the efficiency and robust of our algorithm. Row 1 in the table lists all test cases and row 2 shows types of different flip-flops that can be used in each test case. Rows 3 and 4 respectively, show numbers of flip-flops and total power consumption in original test cases. After some flip-flops are replaced by our algorithm, the power consumption of each design is shown in row 5, and row 6 computes the ratio of power reduction by our algorithm, which is denoted by PR_Ratio. From rows 7 to 9, it shows the wirelength reduction by our algorithm. Rows 7 and 8 show the original wirelength and the wirelength after our program is applied. Finally, the

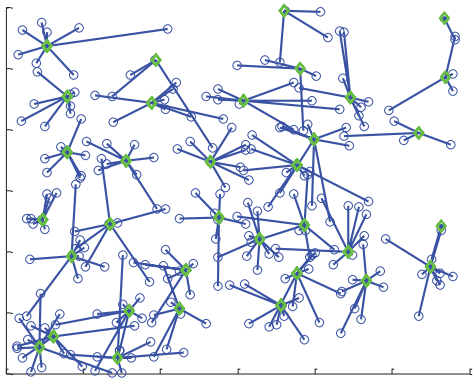


Fig. 19. Resulting distribution of flip-flops (34 flip-flops, power = 9484, wirelength = 71 304)

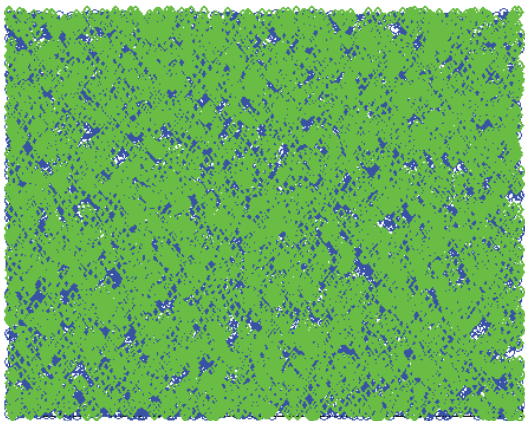


Fig. 20. Distribution of flip-flops in the original design. (5524 flip-flops, power = 552 400, wirelength = 3 562 985).

ratio of wirelength reduction, which is denoted by WR_Ratio, is shown in row 9.

The values of PR_Ratio in all cases are between 20 and 30. Besides, the wirelength are less than the original circuit in all cases, and the best value of WR_Ratio can achieve 42.18% improvement. Row 10 shows the execution time of each case. Because of the long execution time of parser, we show the execution time of parser in row 11.

Fig. 17 displays the curve of the execution time with respect to various flip-flop numbers in a circuit. The test cases are obtained by duplicating Case 1 various times. The x-axis represents the number of flip-flops, and the y-axis denotes the percentage of a execution time compared with the longest execution time. As the number of flip-flops increases, the execution time of parser will be longer than execution time which does not include parser. For this reason, the execution time in Fig. 17 does not include the execution time of parser. The largest case, which contains about 1 700 000 flip-flops, takes the longest execution time (about 10 min). According to Fig. 17, it shows that the timing complexity of our algorithm is $O(N^{1.12})$ instead of $O(N^2)$.

Figs. 18 and 19 show the original distribution of flip-flops and the resulting distribution of flip-flops after applying our program. In the figures, flip-flops are denoted by green circles and pins by blue circles. Blue lines represent the wires that

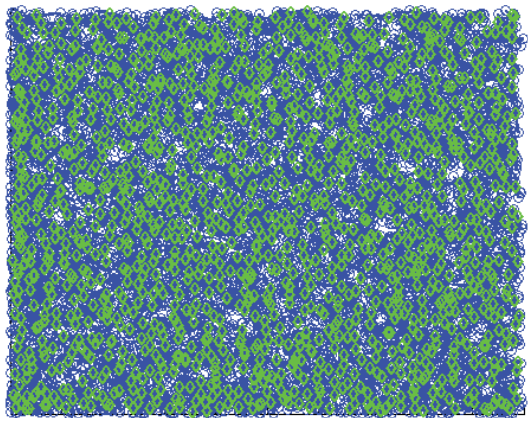


Fig. 21. Resulting distribution of flip-flops. (1378 flip-flops, power = 430 260, wirelength = 2 189 215).

connect pins and flip-flops. In Fig. 18, there are 120 1-bit flip-flops and 240 pins in the original circuit in **Case 1**. After applying our program, there only exist 27 4-bit flip-flops, five 2-bit flip-flops and two 1-bit flip-flops in the new design shown in Fig. 19. In Fig. 20, there exist 5524 2-bit flip-flops and 11 048 pins in the original circuit in **Case 3**. There only exist two 6-bit, 1284 4-bit, 34 2-bit, and eight 1-bit flip-flops for the new circuit shown in Fig. 21 after applying our program.

VI. Conclusion

This paper has proposed an algorithm for flip-flop replacement for power reduction in digital integrated circuit design. The procedure of flip-flop replacements is depending on the combination table, which records the relationships among the flip-flop types. The concept of pseudo type is introduced to help to enumerate all possible combinations in the combination table. By the guidelines of replacements from the combination table, the impossible combinations of flip-flops will not be considered that decreases execution time. Besides power reduction, the objective of minimizing the total wirelength also be considered to the cost function. The experimental results show that our algorithm can achieve a balance between power reduction and wirelength reduction. Moreover, even for the largest case which contains about 1 700 000 flip-flops, our algorithm can maintain the performance of power and wirelength reduction in the reasonable processing time.

References

[1] P. Gronowski, W. J. Bowhill, R. P. Preston, M. K. Gowan, and R. L. Allmon, "High-performance microprocessor design," *IEEE J. Solid-State Circuits*, vol. 33, no. 5, pp. 676–686, May 1998.

[2] W. Hou, D. Liu, and P.-H. Ho, "Automatic register banking for low-power clock trees," in *Proc. Quality Electron. Design*, San Jose, CA, Mar. 2009, pp. 647–652.

[3] D. Duarte, V. Narayanan, and M. J. Irwin, "Impact of technology scaling in the clock power," in *Proc. IEEE VLSI Comput. Soc. Annu. Symp.*, Pittsburgh, PA, Apr. 2002, pp. 52–57.

[4] H. Kawagachi and T. Sakurai, "A reduced clock-swing flip-flop (RCSFF) for 63% clock power reduction," in *VLSI Circuits Dig. Tech. Papers Symp.*, Jun. 1997, pp. 97–98.

[5] Y. Cheon, P.-H. Ho, A. B. Kahng, S. Reda, and Q. Wang, "Power-aware placement," in *Proc. Design Autom. Conf.*, Jun. 2005, pp. 795–800.

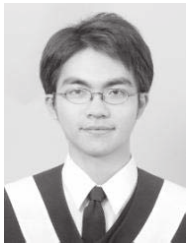
[6] Y.-T. Chang, C.-C. Hsu, P.-H. Lin, Y.-W. Tsai, and S.-F. Chen, "Post-placement power optimization with multi-bit flip-flops," in *Proc. IEEE/ACM Comput.-Aided Design Int. Conf.*, San Jose, CA, Nov. 2010, pp. 218–223.
[7] *Faraday Technology Corporation* [Online]. Available: http://www.faraday-tech.com/index.html
[8] C. Bron and J. Kerbosch, "Algorithm 457: Finding all cliques of an undirected graph," *ACM Commun.*, vol. 16, no. 9, pp. 575–577, 1973.
[9] *CAD Contest of Taiwan* [Online]. Available: http://cad_contest.cs.nctu.edu.tw/cad11

Ya-Ting Shyu received the M.S. degree in electrical engineering from National Cheng Kung University (NCKU), Tainan, Taiwan, in 2008, where she is pursuing the Ph.D. degree in electronic engineering.

Her current research interests include integrated circuit design, design automation for analog, and mixed-signal circuits.

Jai-Ming Lin received the B.S., M.S., and Ph.D. degrees from National Chiao Tung University, Hsinchu, Taiwan, in 1996, 1998, and 2002, respectively, all in computer science.

He was an Assistant Project Leader with the CAD Team, Realtek Corporation, Hsinchu, from 2002 to 2007. He is currently an Assistant Professor with the Department of Electrical Engineering, National Cheng Kung University, Tainan, Taiwan. His current research interests include floorplan, placement, routing, and clock tree synthesis.



Chun-Po Huang was born in Tainan, Taiwan, in 1986. He received the B.S. degree in electrical engineering from National Cheng Kung University, Tainan, Taiwan, in 2008, where he is currently pursuing the Ph.D. degree in electronic engineering.

His current research interests include design automation for high-speed and low-power analog-to-digital converters.

Cheng-Wu Lin received the M.S. degree in electrical engineering from National Cheng Kung University (NCKU), Tainan, Taiwan, in 2006, where he is currently pursuing the Ph.D. degree in electronic engineering.

His current research interests include integrated circuit design, design automation for analog, and mixed-signal circuits.

Ying-Zu Lin received the B.S. and M.S. degrees in electrical engineering and the Ph.D. degree from National Cheng Kung University, Tainan, Taiwan, in 2003, 2005, and 2010, respectively.

He is currently with Novatek, Hsinchu, Taiwan, a Senior Engineer, where he is working on high-speed interfaces and analog circuits for advanced display systems. His current research interests include analog/mixed-signal circuits, analog-to-digital converters, and high-speed interface circuits.

Dr. Lin was the recipient of the Excellent Award in the master thesis contest held by the Mixed-Signal and Radio-Frequency Consortium, Taiwan, in 2005, the Best Paper Award of the VLSI Design/Computer-Aided Design Symposium, Taiwan, in 2008, and the Taiwan Semiconductor Manufacturing Company Outstanding Student Research Award. He received third prize in the Acer Dragon Award for Excellence. He was the recipient of the MediaTek Fellowship in 2009, the Best Paper Award from the Institute of Electronics, Information, and Communication Engineers, and the Best Ph.D. Award from the IEEE Tainan Section in 2010. He was a co-recipient of the Gold Award in Macronix Golden Silicon Design Contests in 2010. He was a recipient of the International Solid State Circuits Conference/Design Automation Conference Student Design Contest in 2011, the Chip Implementation Center Outstanding Chip Design Award (Best Design), and the International Symposium of Integrated Circuits Chip Design Competition.

Soon-Jyh Chang (M'03) was born in Tainan, Taiwan, in 1969. He received the B.S. degree in electrical engineering from National Central University, Jhongli, Taiwan, in 1991, and the M.S. and Ph.D. degrees in electronic engineering from National Chiao Tung University, Hsinchu, Taiwan, in 1996 and 2002, respectively.

He has been with the Department of Electrical Engineering, National Cheng Kung University, Tainan, since 2003, where he is currently a Professor and the Director of the Electrical Laboratories since 2011. He has authored or co-authored over 100 technical papers and 7 patents. His current research interests include design, testing, and design automation for analog and mixed-signal circuits.

Dr. Chang has been serving as the Chair of the IEEE Solid-State Circuits Society Tainan Chapter since 2009. He was the Technical Program Co-Chair of the IEEE Institute for Sustainable Nanoelectronics in 2010, and the Committee Member of the IEEE Asian Test Symposium in 2009, Asia and South Pacific Design Automation Conference in 2010, the VLSI-Design, Automation, and Test in 2009, 2010, and 2012, and the Asian Solid-State Circuits Conference in 2009 and 2011. He was a recipient and co-recipient of many technical awards, including the Greatest Achievement Award from the National Science Council, Taiwan, in 2007, the Chip Implementation Center Outstanding Chip Award in 2008, 2011, and 2012, the Best Paper Award of VLSI Design/Computer-Aided Design Symposium, Taiwan, in 2009 and 2010, the Best Paper Award of the Institute of Electronics, Information and Communication Engineers in 2010, the Gold Prize of the Macronix Golden Silicon Award in 2010, the Best GOLD Member Award from the IEEE Tainan Section in 2010, the International Solid State Circuits Conference/Design Automation Conference Student Design Contest in 2011, and the International Symposium on Integrated Circuits Chip Design Competition in 2011.