Chapter 3

Core-based Incremental Placement Algorithm

Based on the literature reviews, knowledge of design automation, FPGA applications, and software development, an incremental placement algorithm is investigated, a guided placement methodology is presented, and a cluster merge mechanism is discussed in this chapter.

3.1 Incremental Placement Algorithm

As examined in the literature review, minor changes in a design will result in a complete reprocessing of the traditional FPGA design flow. Incremental techniques have great potential to shorten the FPGA development cycle. The higher the density of the logical gates in a device, the more important the incremental techniques. This section presents an implementation of an incremental placement algorithm. Different from other incremental placement algorithms including [Cho96] and [Tog98], this algorithm is based on pre-defined cores and is demonstrated using the JBits API.

Advanced silicon technology provides the capability of supporting more complex designs. The ability of System-on-Chip (SoC) and deep sub-micron technology to enhance the performance of semiconductor chips while lowering total system cost has made SoCs highly sought after in the electronics industry; it has also put high pressure on

silicon providers and tool designers to provide more efficient methods to meet the constant demand of reduce the time-to-market. Intellectual Property (IP) reuse is thus becoming an essential consideration in advanced chip designs [Oli01][Zha01]. As the gate counts grow faster and faster, it is impractical to always build an application from scratch. Employing pre-defined functional units (also known as cores) as the design elements not only can shorten the development cycle, it can also improve the design performance because these cores are generally optimized. IP reuse strategy removes the gap between the available semiconductor technologies and their implementation in chip designs, and it is becoming an important way to ensure the design efficiency, especially for large-scale chip designs [Ayc00].

Cores can be standard library components that can be presented as rectangles or other shapes according to the designer's assignment. Pre-optimized, they are considered as functional units, and are moved and placed without breaking their hierarchy in this incremental placement algorithm.

3.1.1 Basic Implementations

There are two objectives established for this algorithm. The first objective is to reduce the processing time. Time used to place a newly added core should be orders of magnitude smaller than the conventional placement algorithms [Kur65][Bre77], which provides the possibility of making the placement tool more user-interactive. This is also the general goal for all of the incremental algorithms. The second objective is to place a newly added core to a desired position with the minimal possible shift of the existing designs. The desired position for a core has the shortest path with its connected cores. Placing a core in such a position may cause the shift of other existing cores. If the cost of moving some pre-placed cores is larger than the advantages of placing a new core to its desired place, it is not necessary to make this placement. How to minimize the shift, in other words, how to decide which core has to be moved is also a problem that needs to be considered. To achieve these objectives, there are two criteria used in this algorithm:

- the added core will be placed in a group that has the highest connectivity with the new core, and

- the number of cores along the shifted path must be minimized.

According to the first criterion, the device is divided into several clusters depending upon the interconnectivity between the cores similarly as the contemporary bottom-up clustering placement techniques [Ger98]. A core in a cluster does not connect to any of the cores in another cluster. An empty device is evenly divided into four clusters when the first core is placed. More clusters will be inserted between the existing clusters if necessary. When a core is placed, it is put into an existing cluster if it connects to a placed core in this cluster, or it is put into a new cluster if it has no connection with any of the placed cores in the device.

Figure 3.1 illustrates the clustering method. In this example, unconnected Cores A, B, C and D are initially placed in four different clusters. When Core E is added, it is located into the same cluster as Core A because these two cores are connected to each other. Core F is inserted into a new cluster between Core A and Core B because Core F does not connect to any of the existing cores.

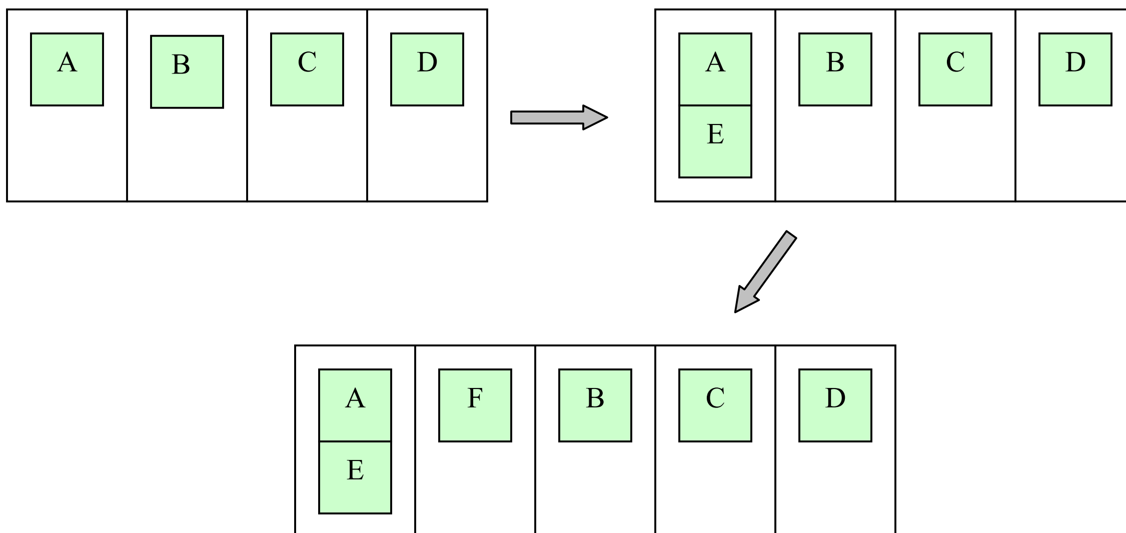


Figure 3.1 Illustration of the clustering method

The clustering method has two advantages. First, it is easy and fast to find a highest connectivity group for an added core. Second, it decreases the possibility for unrelated

cores to be placed in a newly added core's desired position; thus reducing the chances of shifting the existing cores and cutting down the placement time.

As discussed in Chapter 2, the purpose of placement is to place a design in a minimal area while satisfying design constraints (position and size of cells), technological constraints (design rules, number of layers), and performance constraints (longest path allowable, timing) [Chi99] . Wire length, the total distance between connected modules, should be minimized to reduce the capacitive delays associated with longer nets to speed up the operation of the chip [Sha91]. In addition, the shorter distance among the connected modules makes it more likely that the resulting design would be smaller in area. Therefore, wire length is used as the metric to drive the goal of the placement algorithm.

There are several ways to estimate wire length. Typical methods include the minimum rectilinear spanning tree [Sha91], half perimeter [Ger98], and the squared Euclidean distance [Ger98]. Steiner tree is the most efficient wiring scheme, while finding the rectilinear connections of minimum length is NP-hard [She95]. The half perimeter method provides an estimation of Steiner tree length, but it is only accurate for small (less than 5) terminal nets [Sha91]. Euclidean or Manhattan metric provides another good estimation via adding the distances between all pairs of pins in the net. Because routing is generally performed horizontally and vertically in FPGAs, Manhattan geometry is employed as the measure for the distance between cores. The wire length between two cores, for example C1 and C2, is measured using the Manhattan Distance (MD) as

$$MD(C1, C2) = |(r_1 - r_2)| + |(c_1 - c_2)| \ , \tag{3.1}$$

where (r_1, c_1) and (r_2, c_2) are the row and the column on the center of Cores C1 and C2, respectively. The total wire length of a placed design is calculated using

$$W = \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{M}\left(|r_i - r_j| + |c_i - c_j|\right) \ , \tag{3.2}$$

where N is total number of cores, M is the number of cores that connects to the ith core, and (r_i, c_i) is the center position of the ith core.

Figure 3.2 exemplifies the calculation of the wire length using Formula 3.2. In this example, Core A connects to Cores B and C, and Core B connects to Core D.

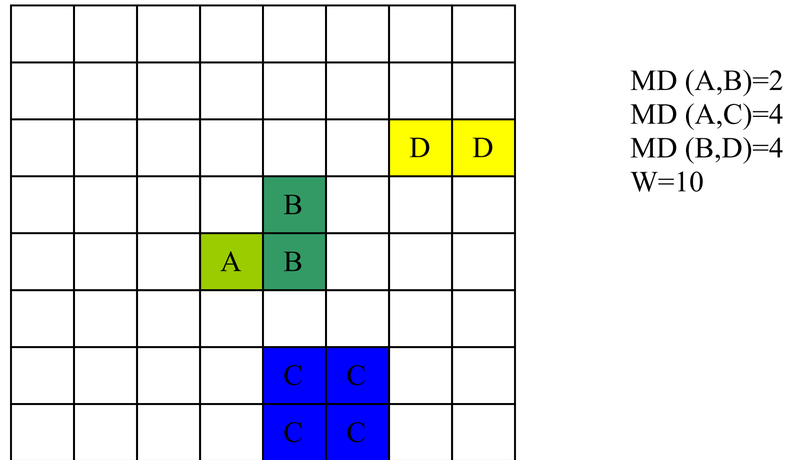


Figure 3.2 Example of counting the wire length

There are four steps in this algorithm:

1) Find an affiliated group for an added core.

An affiliated group is found according to the newly added core's connectivity information. If its connected core has already been placed on the device, the group that the connected core belongs to is the affiliated group for the added core. If both cores are new, they will be placed in a new cluster.

2) Search for a desired position for an added core.

The desired place for a core is the place that has the shortest path with its connected cores. Because this algorithm works incrementally, and it only processes one core at a time, a newly added core only has one connected core at the moment it is added. This core, which is either a placed core or a new core, is named as the target core of a newly added core, and is used to calculate the desired position for the added core.

The shortest distance between two cores is the wire length between them measured by the Manhattan distance. Because this algorithm is demonstrated using JBits, and as introduced in Section 2.3.5, the position at the bottom left corner of a core is required to

place this core in the device using the JBits API. If two cores, C1 and C2, are designed vertically, C1's outputs are connected to C2's inputs, and C2 is a newly added core while C1 has been placed already, the shortest distance between these two cores is presented as

$$MD(C1,C2) = \left|(x_1 + w_1/2) - (x_2 + w_2/2)\right| + \left|(y_1 + h_1/2) - (y_2 + h_2/2)\right|, \tag{3.3}$$

where (x_1,y_1) and (x_2,y_2) are the row and the column of the bottom left corner of Cores C1 and C2 respectively, w and h are the width and height of each core. Core C2 will be placed at

$$\begin{aligned} x_2 &= x_1 + w_1; \\ y_2 &= y_1 + h_1/2 - h_2/2. \end{aligned} \tag{3.4}$$

After finding the desired position for an added core, the empty slot at this position is checked based on its width and the height. Empty slot checking is implemented by the tagged approach. Each CLB has a tag. The tag is set to 0 if it is empty and 1 otherwise. Suppose a desired position for an added Core C is (x,y), then an empty slot is found if

$$\sum_{j=y}^{y+h} \sum_{i=x}^{x+w} tag(CLB(i,j)) = 0, \tag{3.5}$$

where w, h is the width and height of the core.

If the slot is empty, then the newly added core is placed directly at this location. If the slot is not empty, then cores placed in this slot are extracted and their connectivity is examined. If Core A is one of the blocked cores, and if Core A does not have connection with Core C's target core, then it will be moved to a new location. Alternatively, if Core A already has been connected to Core C's target core, it is not necessary to change Core A's position, because doing that will move Core A from its optimum location. In addition, routing will have to be done twice: first route the added core to its target core, then route Core A to its connected cores after it is placed at a new position. Instead of moving Core A, therefore, a search will be made for a new desired position for the newly added core.

Because of the structure of the Virtex device, most of the cores are designed vertically [Xil00]. To save routing resources, a better layout strategy is to place connected cores either left or right of each other. If the desired position for an added core is (x,y), and the

added core's inputs are connected to the target core's outputs, a new desired position is searched for as illustrated in Figure 3.3.

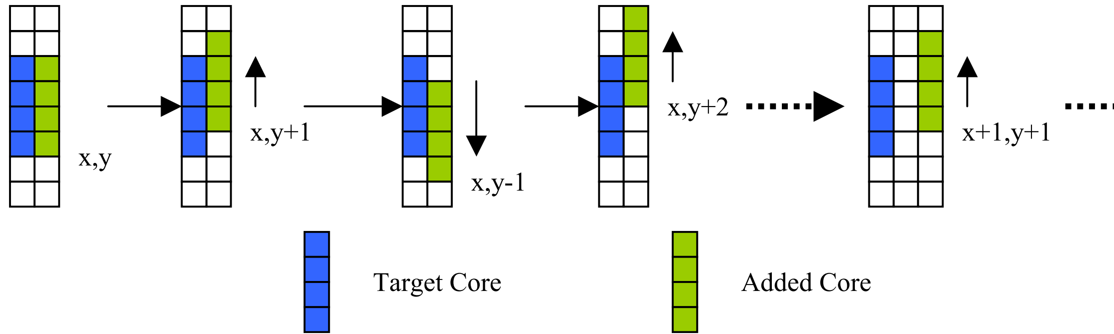


Figure 3.3 Searching the desired position for an added core

The search starts from a previous desired position. It is placed initially one row upward first, then one row downward and two rows upward again. If an empty slot cannot be found in the current column, it will continue from one column to the right. If an added core's outputs are connected to the target core's inputs, then the search will continue from one column left of the previous desired position. This search is repeated until the empty slot is found or the device is fully used.

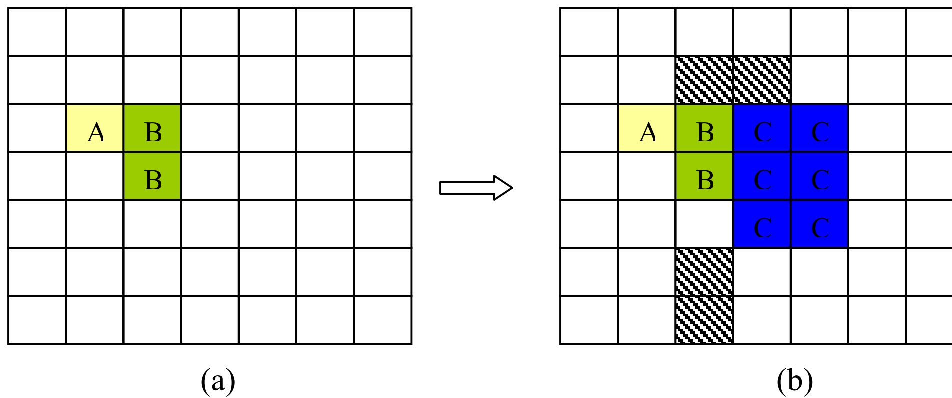


Figure 3.4 A placement example for searching the desired position (a) Place Cores A and B (b) Adding Core C

Figure 3.4 illustrates an example for searching for the desired position for a newly added core. Supposed Core A is connected with Core B; according to the shortest distance calculation, Cores A and B are placed as shown in Figure 3.4a. As more and more

modules are added into the design, some spaces are occupied as displayed in the shadowed areas in Figure 3.4b. When Core C, which also connects to Core A, is added, because its desired position is blocked by another A's connected core (Core B), a nearest possible position is searched for Core C and it is finally placed at a position one column away from Core B as shown in Figure 3.4b.

3) Shift placed cores.

If blocked cores do not have connections with the newly added core's target core, then they are relocated in Step 2. To re-place these shifted cores, they will be considered as newly added cores. For each shifted core, a target position or a target core is chosen and its new desired position is searched for using the algorithm described in Step 2. Three methods are attempted to find the target position or a target core, their implementations are discussed in Section 3.1.2.

4) Place the added core.

In this step, the newly added core is placed at the empty slot, and is registered in the design database.

Figure 3.5 shows a program flow chart of the incremental placement algorithm.

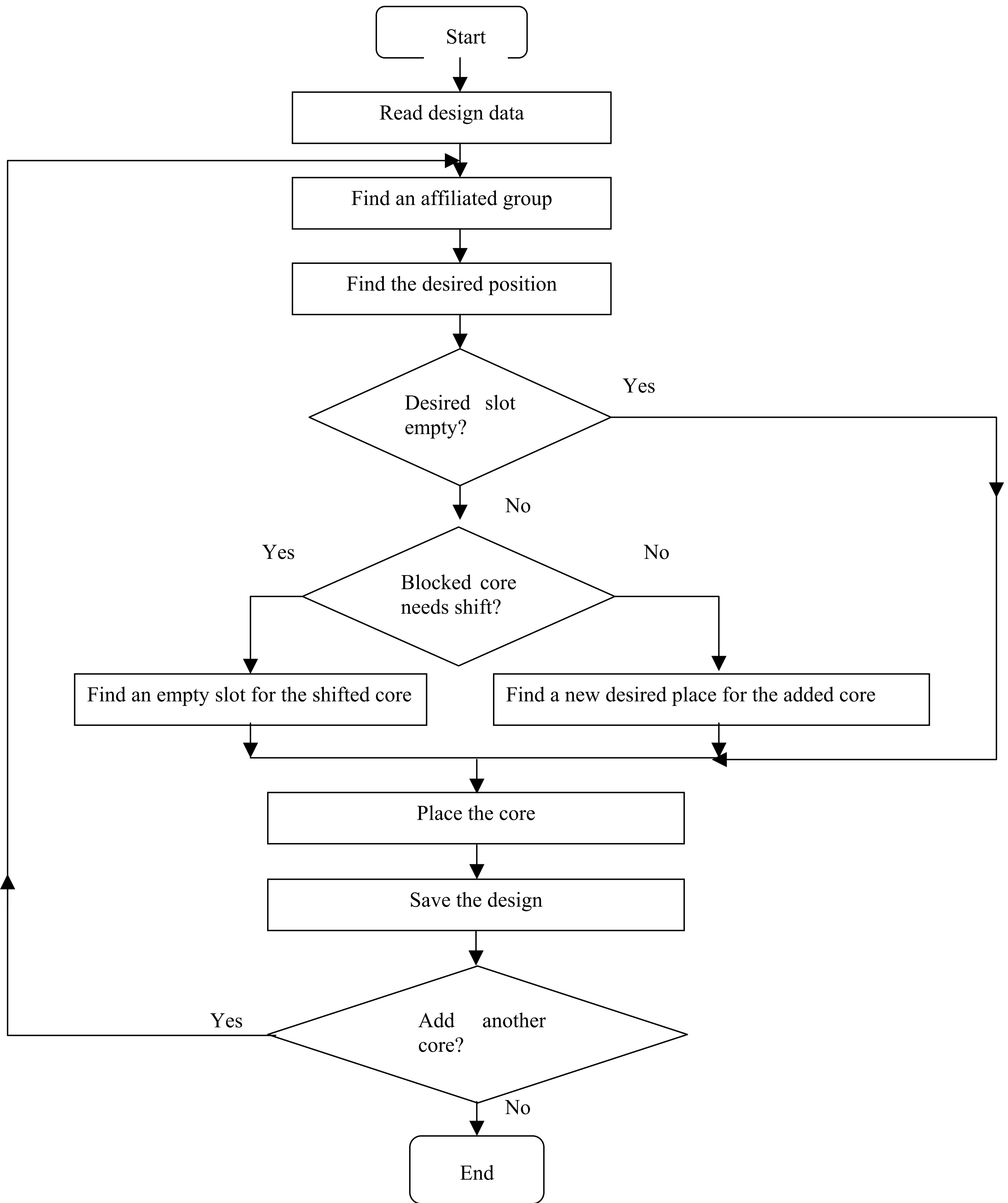


Figure 3.5 Program flow chart of the incremental placement algorithm

3.1.2 Three methods

According to the implementation of the incremental placement algorithms, cores are shifted if they do not have connections with the newly added core's target core. To find a new desired position for these shifted cores using the algorithm described in Step 2, a target position or a target core is chosen first. There are three methods applied to choose the target position/core of a shifted core, and their implementations are described in this section.

3.1.2.1 Method 1: Nearest position

In this implementation, a target core is chosen randomly from the shifted core's connectivity group and the desired position for this shifted core is calculated depending on the location of the selected target core. If the desired position is not available, the shifted core will be placed at a nearest empty position from its desired location for the purpose of saving computation time. Because this method will not move any other existing cores, it is quite possible that there might be no empty space left to exactly fit the shifted core. Placement will fail and the system performance would degrade if only this mechanism is used. To find a balance between the processing time and the system performance, this method is improved by adding further processes for cores that cannot find a nearest empty position.

Under this condition, this shifted core is treated as a newly added core. It is added into the design tool together with its target core and is processed following the incremental placement algorithm illustrated in Figure 3.5. A desired position of for this shifted core is found and the availability of this position is checked. If the blocked cores placed in this desired position do not have connections with the target core of the shifted component, they are to move and leave the place empty for the shifted core. Otherwise, these blocked cores are moved again. The design flow of this method is demonstrated in Figure 3.6 and an example is illustrated in Figure 3.7.

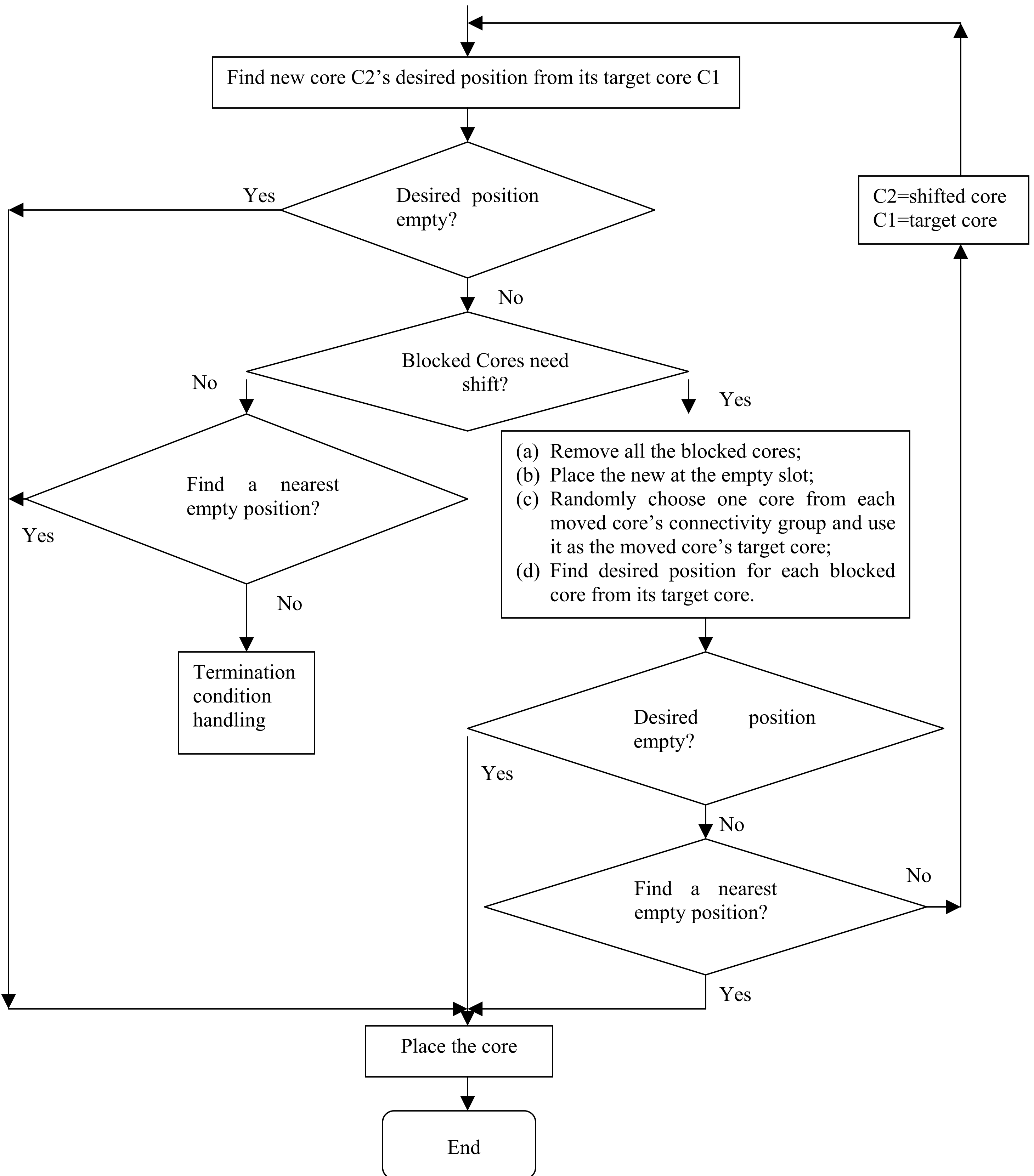


Figure 3.6 Placing a shifted core—Method 1

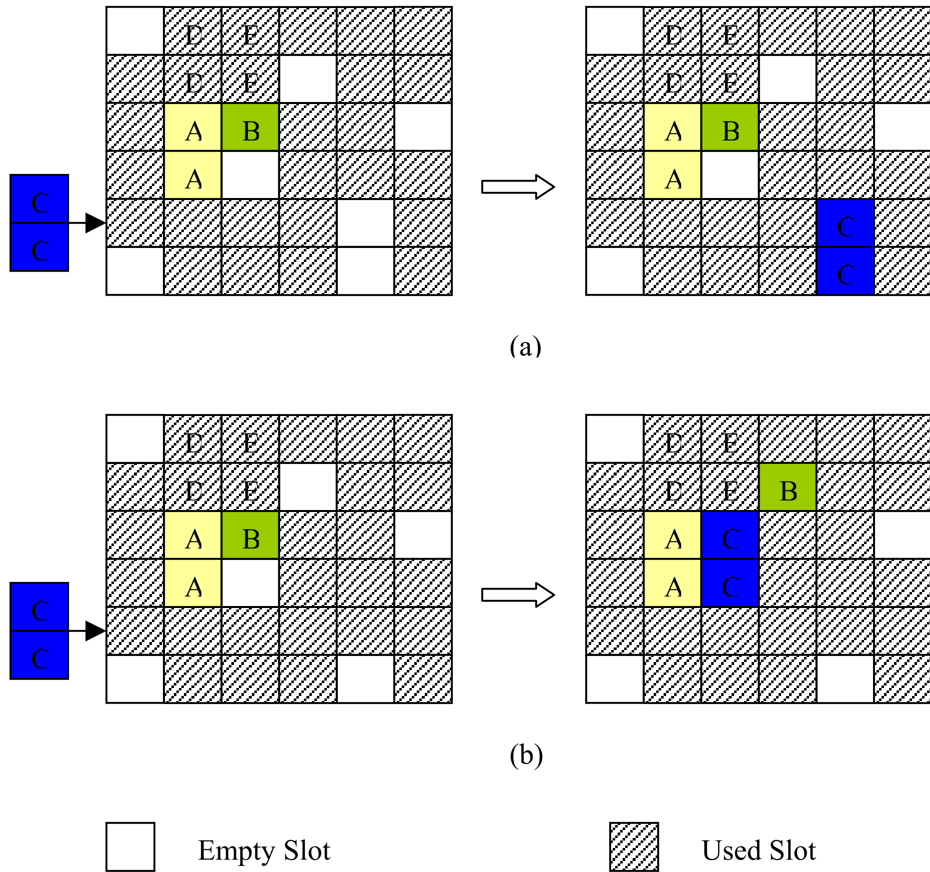


Figure 3.7 Placement example using Method 1 (a) Finding nearest empty position (b) Moving blocked cores if nearest empty position not found

In the example shown in Figure 3.7, Core C is the shifted core and Core A is its target core, which is randomly chosen from Core C's connectivity group. Core B stands in the desired position of Core C, but it does not connect to Core A. According to the algorithm illustrated in Figure 3.6, the nearest empty slot is searched for if a shifted core's desired position is blocked. Core C then finds an empty location as shown in Figure 3.7a. Because such an empty slot is not available in the design in Figure 3.7b, Core C is treated as a newly added core and it is placed following the process described in the Step 2 of the incremental placement algorithm. Under this condition, Core B is about to move to leave the space for C because it does not connect to A. Core B's target, Core D, is then extracted, and its desired position is calculated. Because this desired slot is occupied by

another core, E, that is also in D's connectivity group, Core B starts to find a nearest empty location close to its desired position. Because D's output is connected to B's input, B is placed to the right of D considering the convenience of routing, and finally, Core B finds an empty slot that is one column right to its desired position.

3.1.2.2 Method 2: Recursive search

Similar to Method 1, a target core is also randomly selected from the shifted core's connectivity group in Method 2. Instead of finding a nearest empty position first, the shifted core is directly treated as a newly added core; again, it is added into the design with its target core and is processed following the procedure described in the Step 2 of the incremental placement algorithm whether if a nearest empty slot is available or not. This means cores that block the desired position of a shifted core but do not belong to the same connectivity group as the shifted core would be moved. This procedure is repeated until all the shifted cores find a position.

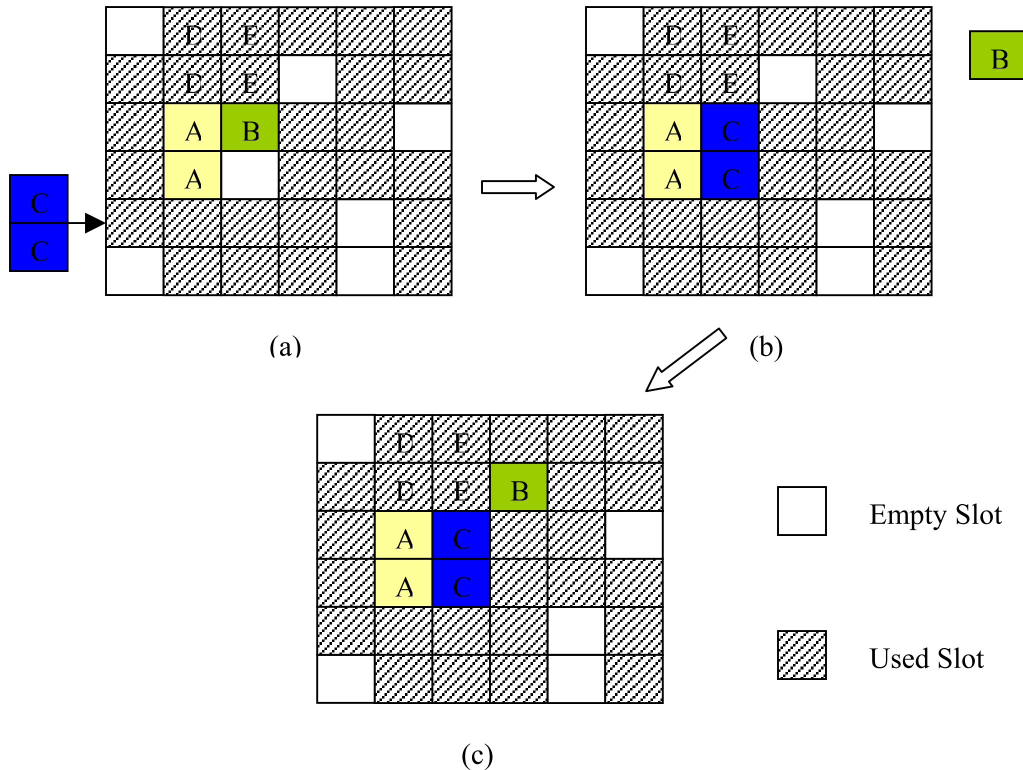


Figure 3.8 Placement example using Method 2—Example 1

Using this method to process the design shown in Figure 3.7a, in spite of an available empty slot two columns away, Core C is still placed at its desired position when forcing Core B to shifted to another place. Core B becomes the shifted core at the second iteration, and Core D is selected as its target core. Because Core E also connects to Core D, Core E stays at its original location while Core B finds a nearest empty place one column to the right as shown in Figure 3.8. If in this design, Core D does not connect to Core E (this could happen because several shifts may have been made, and some connections have been added between existing cores before reaching the stage captured in this example), Core E is also about to move. Figure 3.9 demonstrates the process flow of this situation.

(a)

(b)

(d)

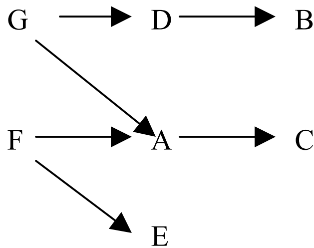
(c)

Empty Slot

Used Slot

Figure 3.9 Placement example using Method 2 –Example2

More cores are involved in the shifting procedures in this example. The connectivity among those related cores is shown as follows:



There are four steps to process the design shown in Figure 3.9.

a. Core C's target core, Core A, is extracted and Core C's desired position is calculated.
b. Core B is moved because it occupies Core C's desired position but does not connect to Core A. Core C is placed at its desired position.
c. Core B's target core, Core D, is found. Similarly as in step (b), Core E is moved to leave the space for Core B.
d. Core E's target core, Core F is extracted and its desired position is calculated. Because the blocked core A also connects to Core F, it does not move. Core E is placed at an empty position closest to its desired position.

3.1.2.3 Method 3: Force-Directed Position

Besides calcualting the desired position from its target core, a force-directed criterion is used to find the desired position in Method 3.

In digital design, subcircuits with known internal layouts are known as cells. Cells that are connected with each other form a net [Ger98]. Those interconnected cells feel an attractive "force" among themselves. The goal of the force-directed criterion is to reduce the total force in the net. In this core-based incremental placement algorithm, the purpose of the force-directed criterion is to reduce the total force among a group of connected cores. To achieve this goal, a moved core will be placed at a position where the core feels the lowest net force.

Suppose (x_{fi}, y_{fi}) represents the location of the left bottom corner of the *i*th core in the shifted core's FROM vector (means the *i*th core's output connects to the shifted core's input), and (x_{tj}, y_{tj}) represents the location of the left bottom corner of the *j*th core in the moved core's TO vector (means the moved core's output connects to the *j*th core's input), to achieve the zero force, we will have

$$\sum_{i=0}^{m-1}[(x+w/2)-(x_{fi}+w_{fi}2)]=\sum_{j=0}^{n-1}[(x_{tj}+w_{tj}/2)-(x+w/2)], \textit{ and}$$
$$\sum_{i=0}^{m-1}[(y+h/2)-(y_{fi}+h_{fi}/2)]=\sum_{j=0}^{n-1}[(y_{tj}+h_{tj}/2)-(y+h/2)], \qquad (3.6)$$

where (*x,y*) is the location of the left bottom corner of the moved core; *w* and *h* represent the width and height of this shifted core, and *m* and *n* are the number of cores in the shifted core's FROM and TO connectivity group, respectively. Thus, the force-directed position of a shifted core is defined as

$$x=\left\{\sum_{i=0}^{m-1}(x_{fi}+w_{fi}/2)+\sum_{j=0}^{n-1}(x_{tj}+w_{tj}/2)\right\}\Big/(n+m)-w/2, \textit{ and}$$
$$y=\left\{\sum_{i=0}^{m-1}(y_{fi}+h_{fi}/2)+\sum_{j=0}^{n-1}(y_{tj}+h_{tj}/2)\right\}\Big/(n+m)-h/2. \qquad (3.7)$$

The shifted core is placed at the force-directed position if it is empty, otherwise it will be located at an empty slot that is closest to this force-directed position.

Because the design is processed incrementally, the connectivity group used to find the force-directed position contains not the full but the partial connectivity information. Thus, the force-directed position calculated during the design might not be a globally optimal solution, but it is still expected to improve the system performance.

3.1.3 Terminating condition

Section 3.1.1 discussed the basic procedures of the incremental placement algorithms and the methods and metric used to find a desired position for a newly added core. Section 3.1.2 presented three methods applied to find new desired positions for shifted cores. It is observed that the placement would fail if the desired position is blocked and a nearest

empty slot is not available after applying the methods described in Sections 3.1.1 and 3.1.2. This section handles cores that cannot find a valid position on the device, and discusses the terminating condition of the incremental placement algorithm.

Because the incremental placement algorithm tries to place a core at a relatively desirable position with a limited perturbation of the existing design, it is possible for the algorithm to not find a valid location for an added core even moving some other cores could achieve that. Figure 3.10 illustrates such a condition.

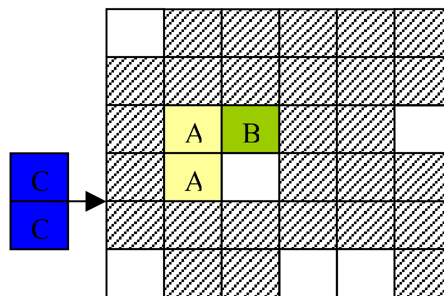


Figure 3.10 A placement example that cannot find a desired position for an added core

In Figure 3.10, Core A connects to both Cores B and C. When Core C, which can be either a newly added core or a shifted core, is added into the placement tool, it cannot find an empty position on the device because a core (Core B) occupies its desired location, and Core B also has a connection with Core A. But when we compare the total number of empty slots and the size of Core C, it can be seen that it is still possible to find a location for Core C if we move some placed cores.

A `CheckIfUnfound()` method is called to collect these "homeless" cores if a valid location on the device can not be found. Those cores are saved in an UNFOUNDCORE database and are processed after all the cores are added into the design. One of the neighbors of this "homeless" core is chosen as the target core, and a desired position is calculated from this target core. The availability of the desired slot is checked first. If it is not empty, all the blocked cores will be removed whether or not they have a connection to the target core or not, forming a small empty region is formed. Those shifted cores

together with the “homeless” core are placed in or close to this region as the descending order of their core height. The incremental placement algorithm will report placement failure if it is still not possible to find a valid location for this “homeless” core. An example demonstrating this method is shown in Figure 3.11, and the flow chart of this method is shown in Figure 3.12.

This method refines the terminating condition of the incremental placement algorithm, and provides the possibility of successfully placing some previously failed designs. As for the design example illustrated in Figure 3.10, it can be successfully placed via =this refined terminating condition as presented in Figure 3.11.

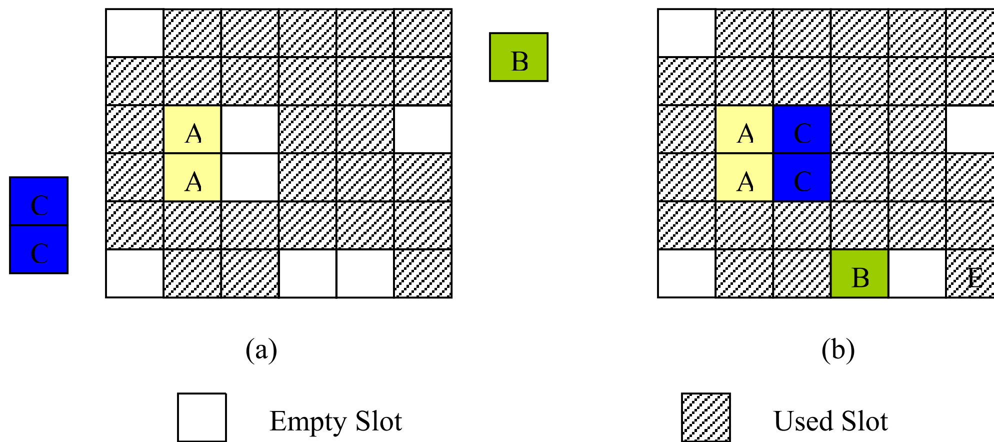


Figure 3.11 Place the design in Figure 3.10 using the refined terminating condition (a) Remove all the block cores (b) Place these cores as the descending order of their height

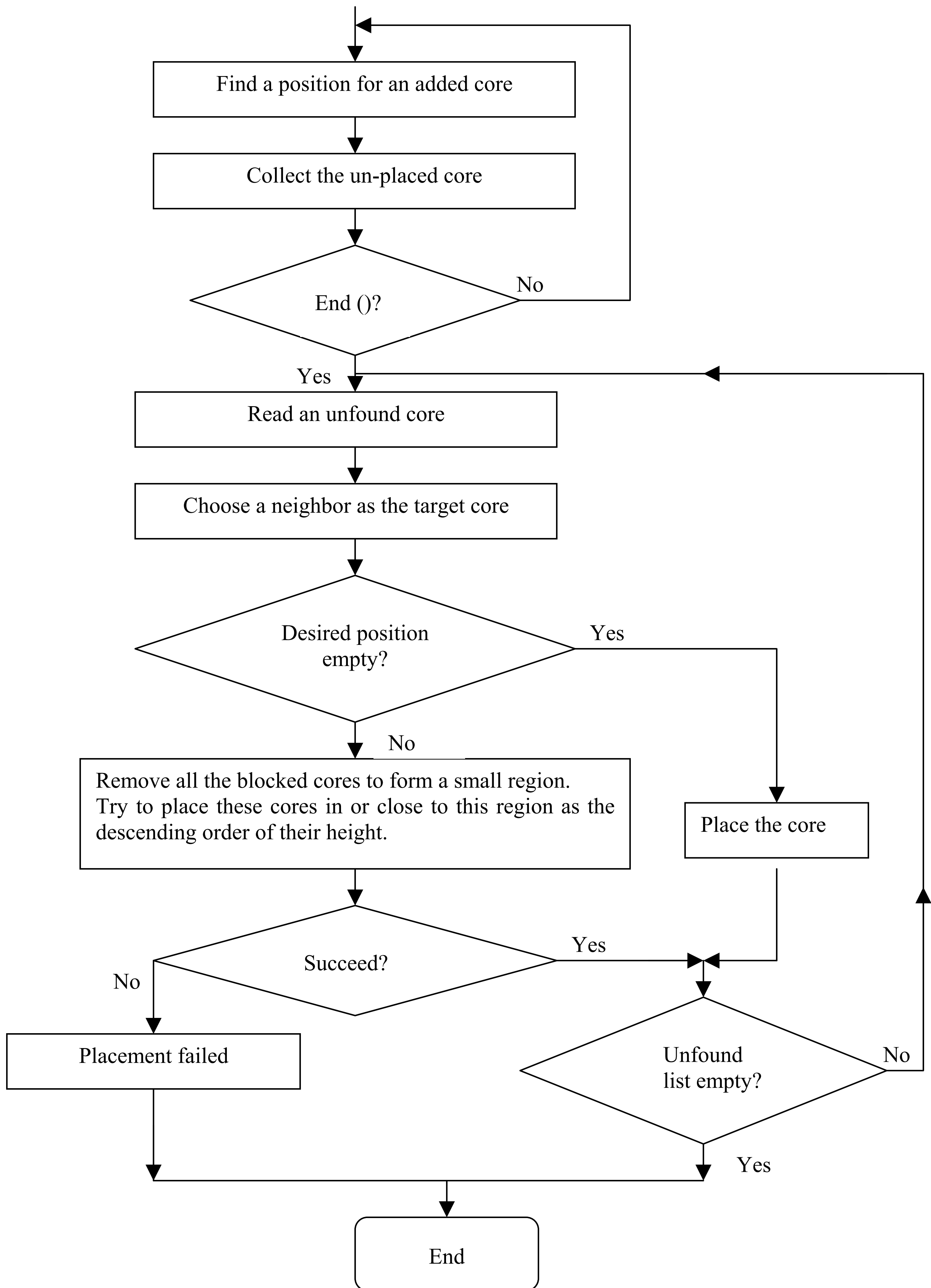


Figure 3.12 Method to place "homeless" cores

3. 2 Guided Placement

The incremental placement algorithm presents a method to place an added core in a desired position with fast processing speed and a small possible shift of the existing design. Because it processes the design incrementally, when the first core is added, it will be placed at the center of the first cluster. Then when the second core is coming, it is put at a desired location that has the shortest distance with the first core if they are connected. Following this routine repeatedly, this method can be used to process a design from scratch.

Engineering changes frequently occur during FPGA development. Modifications are often made to improve our design in design-and-debug cycle. And generally, these changes are small at each time. If only a register is added, or the size of an adder is changed, then it is not hoped to re-process the entire design. Guided placement methodology becomes strongly powerful when minor modifications are made in a design by processing only the changed portions.

Guided placement methodology discussed in this section is intended to solve the problem of how to find changed parts in a design and how to take advantage of the optimized design from previous iterations.

3.2.1 Implementation

To save processing time and use an incremental placement algorithm described in Section 3.1, it is necessary to find a method to determine the changed portions of a design during each modification. Guided placement is a way to provide this function. By comparing the differences between current and previous designs, guided placement can determine and optimize a small changed block(s), and localize the design changes without affecting the remainder of the refined designs.

To implement guided placement, a guide file is used as a template for placing the input design. To increase productivity, the placement information from the last design iteration

is saved in a text file, and is used as the guided design for next design iteration. This guide design can be obtained using the incremental placement algorithm; it can also be a placement created from another placement algorithm such as simulated annealing or a genetic algorithm. An example of using the guide design from a simulated annealing placer can be found in Chapter 5.

When a design is loaded, the existence and creation time of its guide file is verified to determine if the input design has been updated because the guide file was most recently saved. This examination is implemented using Java `exists()` and `lastModified()` functions from the `java.io.File` class.

There are three situations during the examination:

1. A guide file does not exist. This means that the input design is new and it needs to be processed from scratch. Each core in the design is then considered as a newly added core, and is located using the incremental placement algorithm described in Section 3.1.

2. A guide file exists, but the input design has not been changed because the guide file was updated. Under this condition, no placement algorithm is called and the input design is placed according to the information from the guide file.

3. A guide file exists, and the input design is updated. Changed block(s) are distinguished from the unchanged blocks and the design is reprocessed by placing unchanged blocks with the information from guide file and locating changed blocks using the incremental placement algorithm.

Distinguishing changed cores from unchanged cores is based on a signal and component name. It is assumed that each component has a unique name even if it is instantiated from the same RTPCore. It works according to the following rules:

- If a component in a new design has the same name as a component in its guide design, and the two components have the identical width and height, it is placed where it was in the guide design.

- If an unnamed component in a new design is the same type as an unnamed component in its guide design, and the two components not only have the identical width and height, but also the identical connected components, the component is placed where the matching component was in the guide design.

- If a component in a new design has the same name but different width and/or height as a component in the guide design, it is considered as a changed core and is placed using the incremental placement algorithm.

- If a component cannot find a match from the guide design, it is considered as a newly added core and is placed using the incremental placement algorithm.

This guided placement algorithm is expected to save processing time when minor changes are made in a design. If the changed portion is large, then the guided design will not help but may limit the performance of placing the entire design. Therefore, before calling the incremental placement algorithm to process the changed parts, the percentage of the changed portions is compared with the entire design. If the changed part is above 20% of the entire design, instead of the guided placement methodology, the whole design will be placed directly using the incremental placement algorithm discussed in Section 3.1. This threshold is chosen according to the recommendation from the commercial Xilinx and Synplify placement tool [Xil99]. Figure 3.13 shows the program diagram of the guided placement methodology.

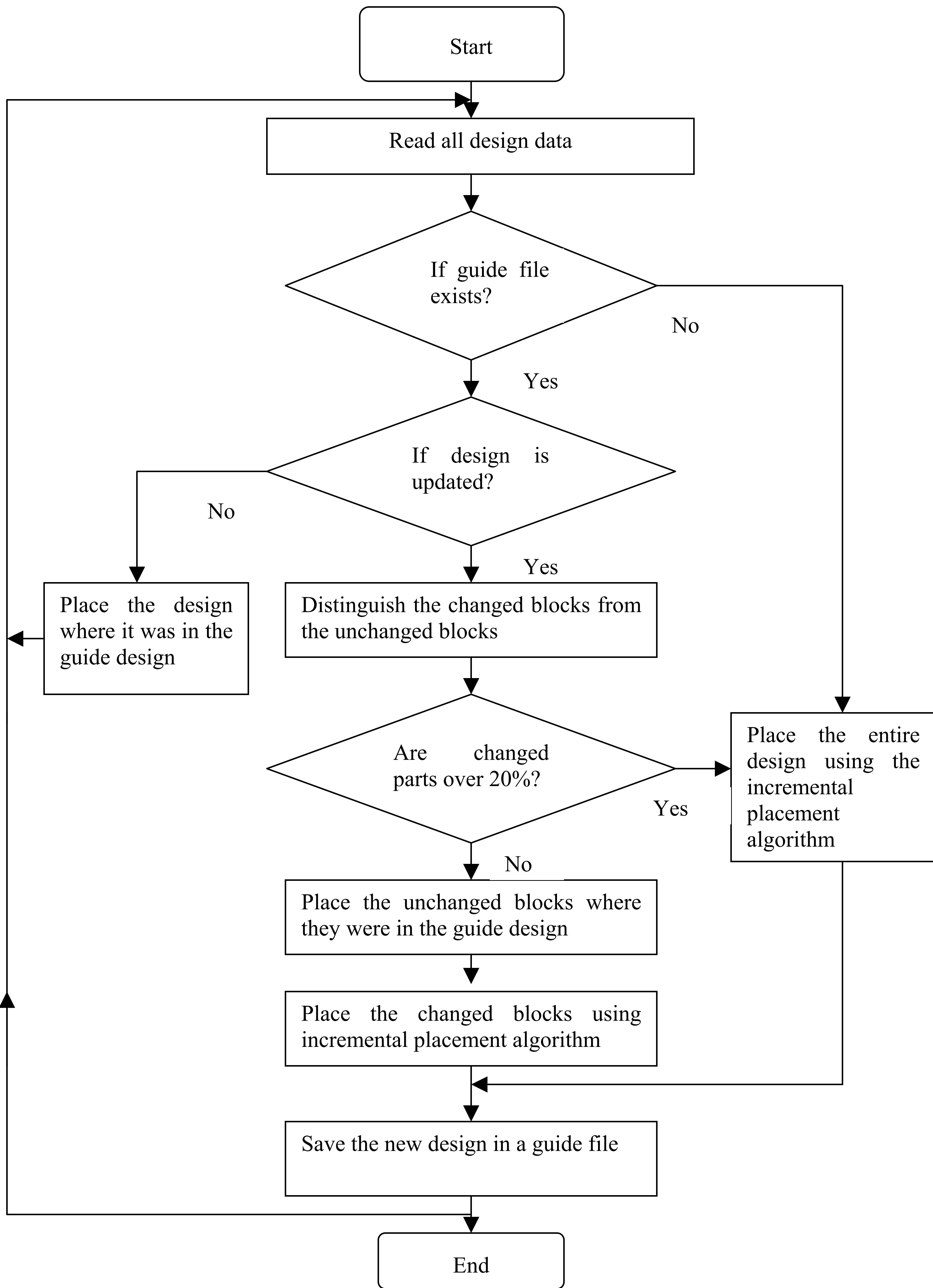


Figure 3.13 Program flow chart of the guided placement methodology

3.2.2 Exception handling in guided placement methodology

There are a few exceptions to when the guided placement methodology is employed. The first occurs when a larger design is used as a smaller design's guide. According to the rules defined for the guided placement, if a match is found between a newly added core and a core from the guide design, the algorithm directly copies the placement information, and places the new core where it was from the guide design. If the current design is smaller than the guide design, then the position that is read directly from the guide design may not be desirable for the input design, leading to bad placement performance, particularly if the guide design is much larger than the input. In this case, the information read from the guide design is useless, and it is better to find a method to avoid performance degradation.

The second exception occurs when the current and the guide design are using different devices, and the capacity of device used in the guide design is larger than that of the current design. Even if the algorithm can find the placement information directly from the guide design, it is not always useful because some positions may have exceeded the boundary of the current device. Using this information may lead to the failure of the placement algorithm.

Two methods are developed to address these exceptions. For the first situation, if a match is found between the current and the guide design, instead of placing the core directly, it is saved in a DIRECTPLACE vector together with its placement information. If no match is determined, then this core will be saved in a TOBEPROCESS vector. After all of the cores are added, the size of the current design and the size of the guide design are compared. If the total device utilization measured in the number of used Configurable Logic Blocks (CLBs) is larger than 80% of the guide design (this criterion can be modified), then cores saved in the DIRECTPLACE vector will be placed where they were from the guide design, and cores saved in the TOBEPROCESS vector will be placed at a relatively desired position using the incremental placement algorithms. If the

current design is smaller than this criterion, then the placement information saved in the DIRECTPLACE vector is abandoned, and all the cores are processed using the incremental placement algorithm.

To solve the problem in the second situation, the device type is recorded. The capacity of the device used in the current design is compared with that used in the guide design, and a flag is set if the guided device is larger than the currently used device. When the guided placement methodology is applied and a match is found between the current and the guide design, the position read from the guide design will be examined to check if it is beyond the boundary of the currently used device. If it is, then this position is discarded, and this core is pushed into in the TOBEPROCESS vector and will be processed using the incremental placement algorithms described in Section 3.1. Otherwise, this core is saved in the DIRECTPLACE vector together with its placement information. Similarly as for the first problem, cores in the DIRECTPLACE and the TOBEPROCESS vector will be processed after all of the cores from the input design have been added.

3.3 Cluster Merge

In the incremental placement algorithm, the design is divided into clusters, and cores placed in one cluster do not have connections with cores in another cluster when they are added. When more and more cores are put into the design, as illustrated in Figure 3.14, it is quite possible that a newly added core connects to cores in different clusters. Putting the added core in one of the clusters will make it far from another, thus increasing the total wire length and possibly the delay between connected cores. In addition, distributing connected cores separately from each other will leave less efficient spaces for the following added cores therefore reducing the performance of the placement algorithm. Under this condition, these clusters should be merged. This section describes the implementation of the cluster merge method.

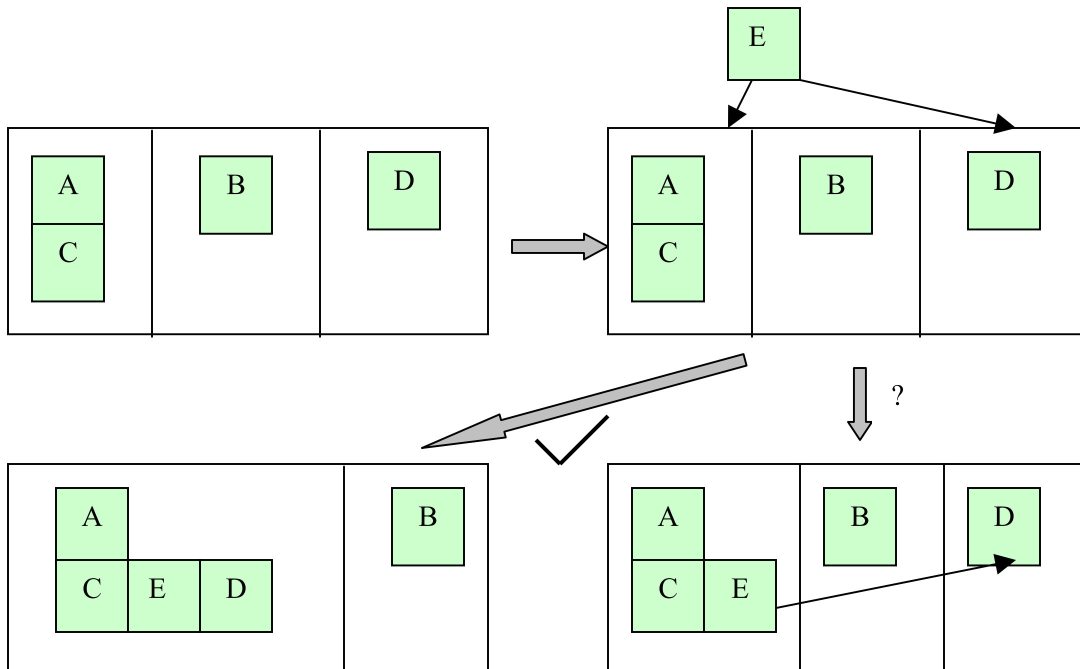


Figure 3.14 Illustration of the cluster merge

3.3.1 Implementation in incremental placement algorithms

The Incremental Placement algorithm can be used to place a design from scratch; it is also effective in processing the minor changes of a design when combined with the guided placement methodology. Different techniques are applied when implementing the cluster merge strategy. This section describes the implementation of the first situation.

Before instigating the cluster merge techniques, let us investigate the clustering placement method developed in this dissertation. When a pair of cores is added into the design incrementally, the design database is checked, resulting one of the four possible.

Case 1: Both of the cores are new. Because these two cores are new and they do not have connections with any existing cores, the will be placed in a new cluster. The first core is located at the center of the new cluster and the second core is placed at its desired position using the incremental placement algorithms discussed in this section.

Case 2: The first core is new and the second core is already placed in the design. The first core will be placed in the same cluster as the second core and its position is determined following the procedure of the incremental placement algorithms shown in Figure 3.5.

Case 3: The first core is already placed and the second core is new. Similarly as Case 2, the second core will be placed in the same cluster as the first core.

Case 4: Both of the cores have already been placed in the design, and they may or may not be in the same cluster. There is a new connection between these two cores.

Clusters need to be merged only in Case 4 if the new connection is made between cores in two different clusters. It is guaranteed that a pair of cores is placed in the same cluster in the first three cases.

```
connect (register1, multiplierA);
connect (constant, register2);
…
connect (constant, multiplierA);
```

Investigating this simple design example. If `register1` has already been placed in Cluster 1, then `multiplierA`'s position is determined by the information of `register1`. Because `constant` and `register2` do not have connection with any of the existing cores, they will be located at a new cluster, Cluster 2. When the design is processed incrementally and a pair of cores, `constant` and `multiplierA` are added, connections are found between Core `multiplierA` in Cluster 1 and Core `constant` in Cluster 2. Thus, `multiplierA` has connections with cores in two different clusters. Clusters 1 and 2 need to merge to avoid large distances between connected cores.

To achieve the cluster merge, the cluster number for each core is recorded in the design database. When a pair of cores is added to the design and both are already placed, their

cluster number will be extracted and compared. If they are in different clusters, the pair with a higher cluster number will be removed and re-placed in the cluster with a lower cluster number, and the lower cluster number is recorded as its new cluster number. Then the cluster number checking and the cluster merging processing are propagated to all of the cores in the removed core's connectivity group. This procedure continues until there are no connections between the existing clusters. The general framework for the cluster merge algorithm is shown in Figure 3.15.

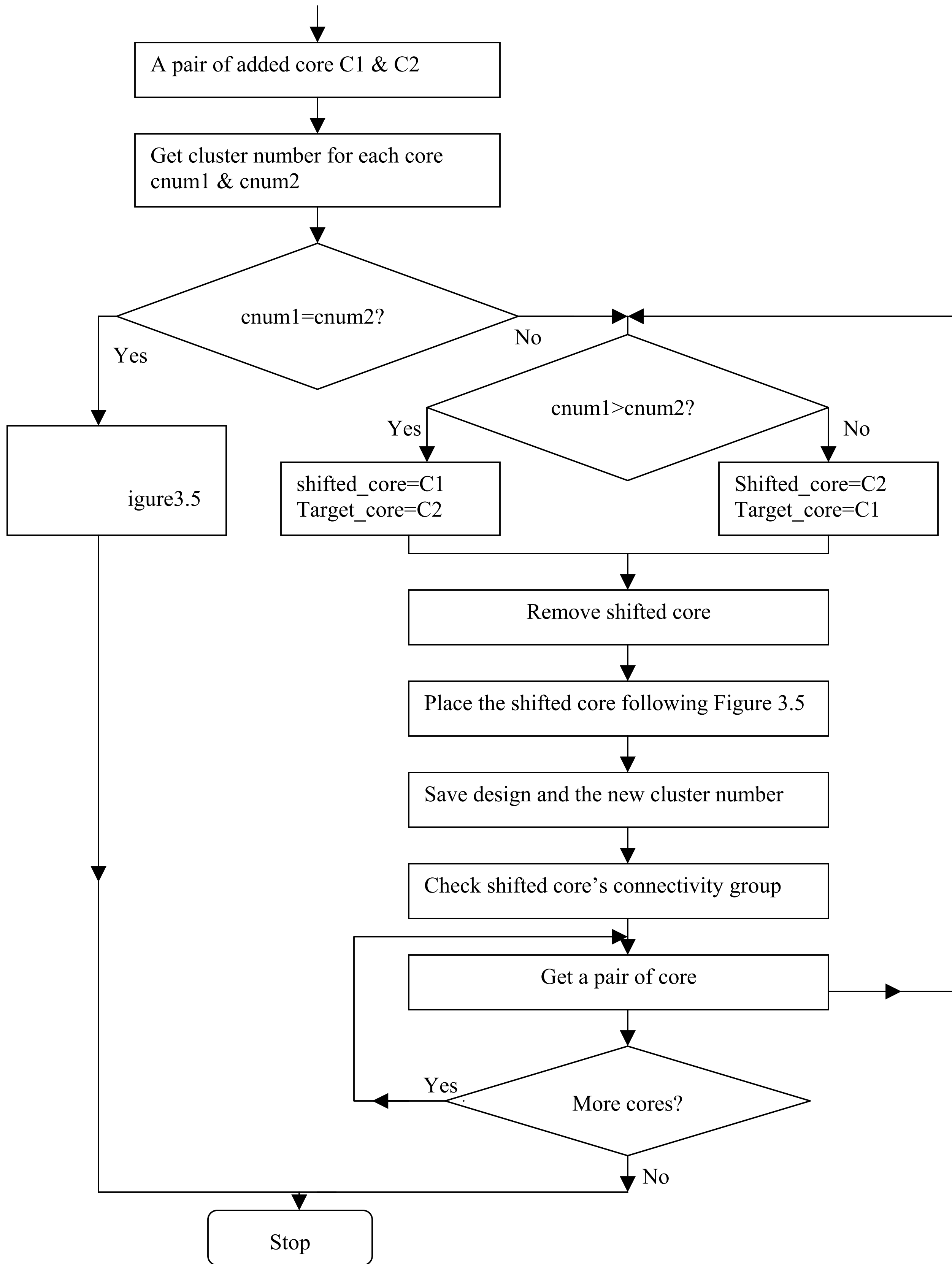


Figure 3.15 Flow chart for cluster merge strategy

3.3.2 Implementation for a changed design

The guided placement methodology is combined with the incremental placement algorithm to process a design with minor changes. During the processing, the input design is compared with the guide template, and the changed cores are distinguished from the unchanged cores. Only the changed cores are processed using the placement algorithm.

Before discussing the implementation of the cluster merge strategy, the definition of the changed portions in the guided placement methodology is investigated. According to the discussion in Section 3.2, a changed core is the one that cannot find a matched core from the guided design with the same name and same size. This definition is expanded when the cluster merge strategy is considered. Besides the changed cores distinguished above, a new connection between two unchanged cores also make it necessary to count one of those two cores changed when implementing the cluster merge strategy.

For example, `register1` and `multiplierA` are unconnected cores placed in two different clusters in the guide design. In the modified input design, both the name and the size of these two cores are unchanged; the only difference is that a connection is added between them. This difference does not affect the guided placement methodology, but it needs to be considered in the cluster merge implementation.

For cores that are considered changed because of the differences of core name and/or size, or because they are new cores compared with the guided design, no special cluster merge strategy is necessary because they are all guaranteed to be placed in the same cluster as their target cores (they belong to the first three cases discussed in Section 3.3.1). For cores that are considered changed because of a new connection between two unchanged cores, the following procedures are employed to ensure the cluster merge.

Step 1: Read the information from the guide file including the cluster number for each core, and save the guide design in the guide design database.

Step 2: When a pair of cores is added to the design, they are saved in the current design database. Because both of the cores have matched cores from the guided design and there is a new connection between them, the cluster numbers of both cores are extracted and compared. If their cluster numbers are the same, no special processing is required, and they are placed according to the guide design. If their cluster numbers are different, then the one with the higher cluster number is moved to the cluster with a lower cluster number using the algorithms developed in Section 3.3.1. The same procedure will be repeatedly performed on all the cores that are in the connectivity group of the moved core.

3.4 Summary

An incremental placement algorithm was presented in this chapter. The detailed process flow was also explained. The three methods employed to find desired positions for shifted cores were discussed. The terminating condition refinement techniques were described as well. The guided placement methodology was investigated to find changed blocks in a design and to take advantage of the optimized design from previous iterations. The exception handling techniques for this methodology were also presented. Finally, cluster merge strategies were described to complete this core-based guided incremental placement algorithm.

Different from other incremental placement algorithms, this algorithm uses pre-defined cores as design elements, and places and moves a core without breaking its original shape. In additional, this algorithm can be employed to not only process minor changes in a design, but can also be used to place a million-gate design from scratch. Aimed to significantly reduce the processing time, this core-based incremental placement algorithm is expected to accelerate the FPGA design-and-debug cycle at a speed orders-of-magnitude faster than the traditional FPGA design flow, thus providing the possibility of developing a more user-interactive integrated FPGA design-and-debug environment.