A New Partitioning Approach for Layout Synthesis from Register-Transfer Netlists

by

Allen C. H. Wu
Daniel Gajski

Technical Report 90-10

Information and Computer Science Department
University of California, Irvine
Irvine, CA. 92717

Abstract

Most of the IC today are described and documented using heiarchical netlists. In addition to gates, latches, and flip-flops, these netlists include sliceable register-transfer components such as registers, counters, adders, ALUs, shifters, register files, and multiplexers. Usually, these components are decomposed into basic gates, latches, and flip-flops, and are laid out using standard cells. The standard cell architecture requires excessive routing area, and does not exploit the bit-sliced nature of register-transfer components. In this paper, we present a new sliced-layout architecture to alleviate the preceding problems. We also describe partitioning algorithms that are used to generate the floorplan for this layout architecture. The partitioning algorithms not only select the best suited layout style for each component, but also consider critical paths, I/O pin locations, and connections between blocks. This approach improves the overall area utilization and minimizes the total wire length.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Surveys of VLSI products reveal that most fabricated chips can be described by register-transfer schematics or netlists. In addition to gates, latches, and flip-flops, schematics include register-transfer components such as registers, counters, adders, ALUs, shifters, multiplexers, and register files. The products in this category include DMA controllers, bus controllers, disc controllers, and programmable I/O interfaces; that is, basically all chips for computer design with the exception of CPUs and memories.

The common used layout strategy for such designs is the use of standard cells. Standard cell methodology, however, does not take into account the regular nature (bit-slice property) of register-transfer components, since those components are decomposed into basic gates, latches, and flip-flops before layout. Greater layout density can be achieved if register-transfer components are laid out in a bit-sliced layout architecture.

The general bit-sliced layout architecture has two weaknesses, however. First, if all sliceable components do not have the same number of bits, then some bit-slices are not used. For example, when a 4-bit counter and a 12-bit register are laid out in the bit-sliced style, 8 bit-slices are wasted since the 4-bit counter occupies only 4 bit-slices. Second, routing of bit-slices with different indices is difficult. For example, if bits 7 and 8 of the register must be connected to bits 1 and 2 of the counter of the previous example, routing across bit-slices must be introduced.

In this paper, we describe a new layout architecture to alleviate the problems of general bit-sliced layout architecture. We also present a top-down layout methodology and algorithms that generate layouts for register-transfer schematics. In most conventional hierarchical layout approaches, placement and routing are carried out in a bottom-up fashion [Ayre90, DaEs89]. In this approach, each predefined block is designed individually, then the blocks are connected using routing channels between blocks. However, our layout methodology is performed in a top-down fashion [UeKi85] that implements three partition phases: (i) partition the register-transfer schematic into bit-slice and glue-logic components, (ii) partition the bit-sliced stack to minimize the layout area of the bit-sliced components, and (iii) partition glue-logic components into blocks based on the target architecture (Figure 1). In addition, pins are assigned to the proper sides of blocks in order to minimize the wire crossing between blocks.

The remainder of this paper is organized as follows. We present a sliced layout architecture in Section 2. Three partitioning algorithms are described in detail in Section 3-5. The experimental results are shown in Section 6. Finally, Section 7 contains the conclusions.

2. Sliced-Layout Architecture

There are two common layout architectures for bit-sliced components: bit-slice abutment [JaJe85, Joha79] and bit-sliced macros or standard cells with routing channels [HsGr87, RoWa87, ThKo87, LuDe89]. The first layout architecture abuts bit-sliced cells with over-the-cell routing for connecting bit-slices with the same indices. This architecture, however, wastes area if units with varying bit-widths are in the same stack or if different indices of the bit-slices must be connected. The second layout architecture stacks bit-sliced macros or standard cells vertically with routing channels between units. Using this layout architecture, several units with smaller bit-widths can be placed in the

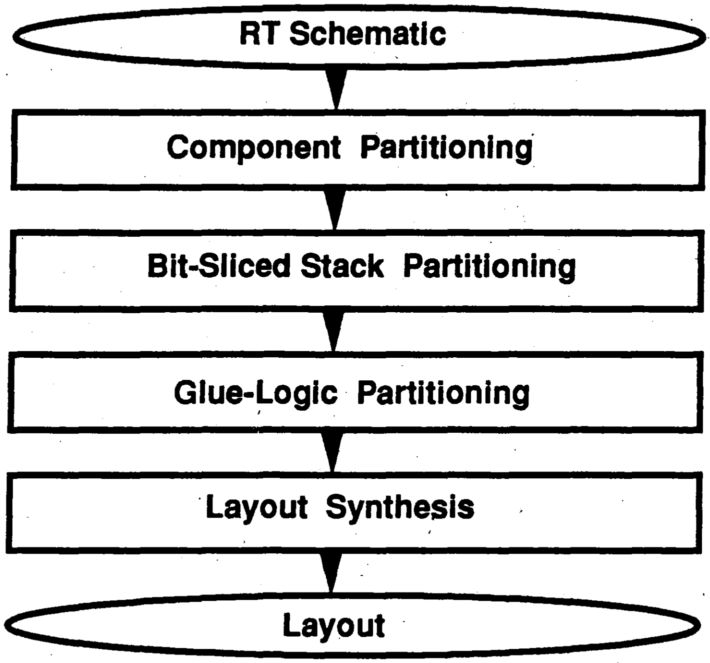

Figure 1. Three partitioning methods for layout generation from RT schematics

same row in order to reduce wasted space. However, routing channels for wire connections between the units contribute to low area utilization.

The sliced-layout architecture use a stack of bit-sliced register-transfer units. This bit-sliced stack (Figure 2) combines cell abutment, over-the-cell routing, and switch box alignment to alleviate the problems of previous approaches. Each bit-slice has the same width and has a fixed number of metal2 routing tracks over each cell. Unit heights vary with the unit functionality. Each bit-sliced cell, such as an ALU, multiplexer, register, adder, or shifter, is designed manually. The layout of each unit is produced by a parameterizable generator according to the given bit-width and I/O pin positions.

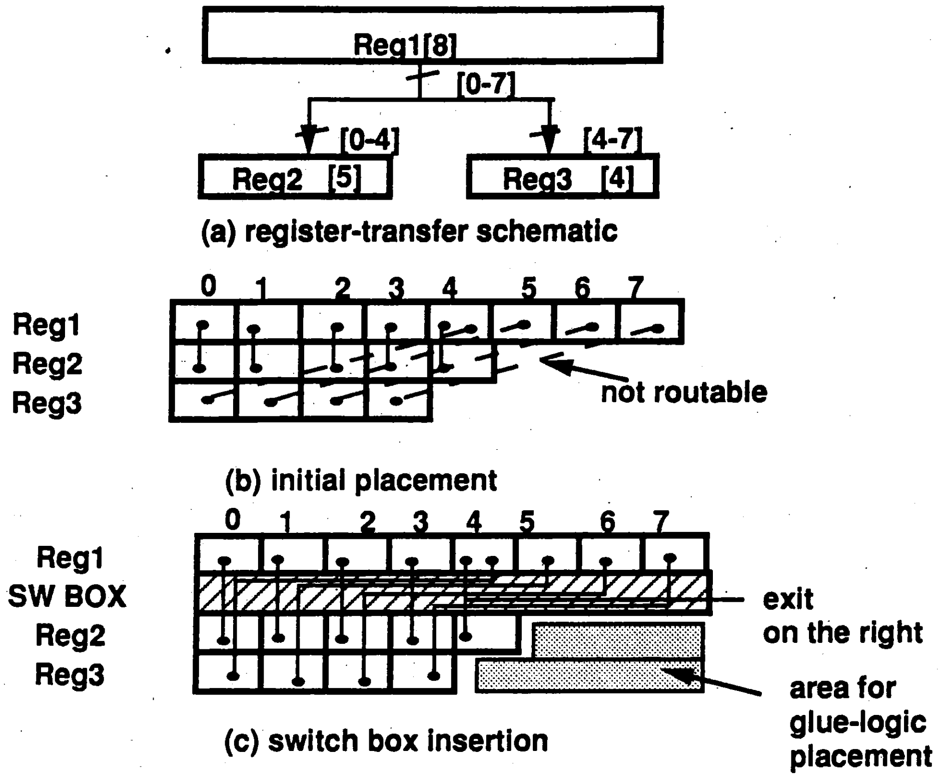

Figure 2. Switch box insertion for wire alignment

The units are stacked in the vertical direction. Using an over-the-cell routing strategy, the data signals run vertically in 2nd metal over the bit-slices. Power, ground, carry, and control lines are routed horizontally in the 1st metal or polysilicon between the bit-slices. An example consisting of three registers (Regl, 2, and 3) is shown in Figure 2(a). Regl has 8 bits, while Reg2 and Reg3 have 5 and 4 bits respectively. The five least significant bits of Regl are connected to Reg2 while the 4 most significant bits of Regl are connected to Reg3. The floorplan of this layout is shown in Figure 2(b). Registers are ordered by bit-width from top to bottom and aligned by the least significant bit.

When several units of different bit-widths are stacked by width as shown in Figure 2(b), a step-shaped triangular area will be created within the stack bounding box. This area can be used for stack folding or for placement of the non-sliceable glue-logic components. Furthermore, a routing channel called a switch box must be inserted in the stack for connecting bit-slices with different indices. In our example, the interconnections between Reg1 (bits 4-7) and Reg3 (bits 0-3) are not routable without a switch box. Therefore, a switch box is inserted as shown in Figure 2(c). Using the same switch box, signals may enter or exit the sliced stack from the left or right as indicated in Figure 2(c).

In the sliced-layout architecture, the stack can be laid out in two different styles. If the netlist contains only a few sliceable components then we use an unfolded stack structure as shown in Figure 3(a). In this case, the glue-logic components are placed into the empty space in the stack bounding box. However, if more space is needed then the glue-logic components are then placed on the left, right, top, and bottom sides of the stack. On the other hand, if the stack contains a large number of sliceable units with highly varying bit widths, the stack structure can be folded as shown in Figure 3(b). The glue-logic components are placed at the sides of the stack bounding box. When the stack height is higher than a given height constraint, the stack must be partitioned into several stacks that can be either folded or unfolded.

3. Component Partitioning

The purpose of component partitioning is to determine the layout style (bit-slice or glue-logic) for each register-transfer component. The component layout style depends on the component type, the component's connectivity, and the overall floorplan. First,

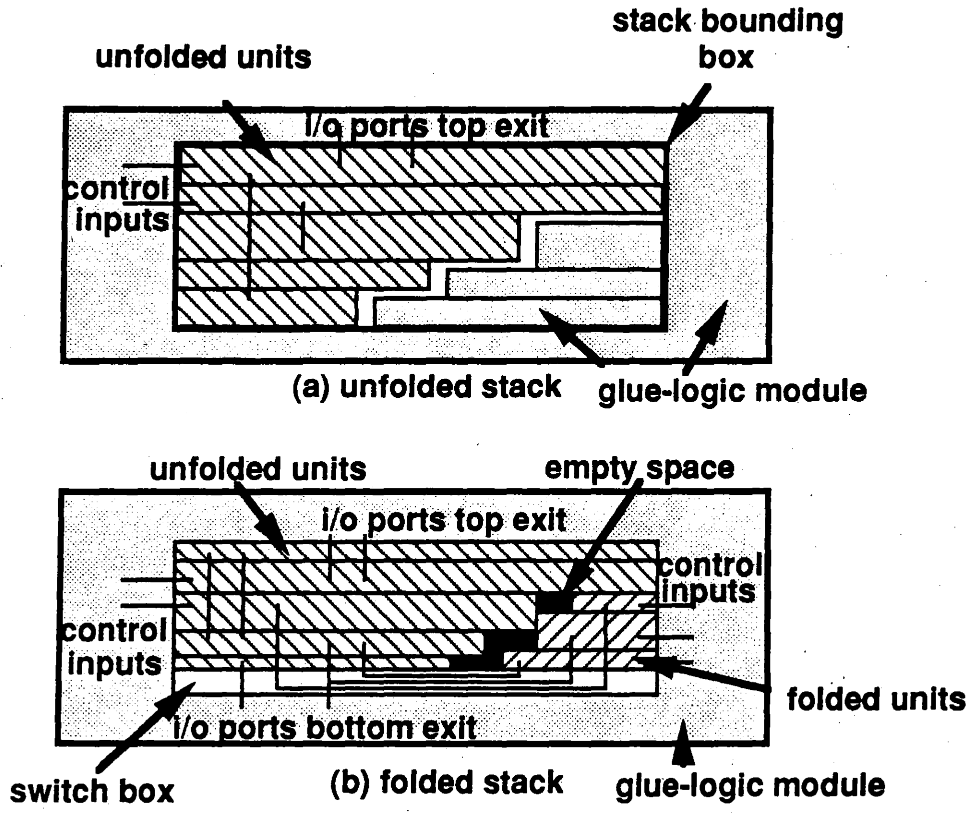

Figure 3. Two sliced layout architectures

components must be partitioned by type since some components such as counters, registers and ALUs are sliceable while others like decoders and encoders are not. Second, small size components can be implemented in two ways. For example, a 2-bit ALU can be implemented using NAND and NOR gates or as a bit-sliced unit. The implementation decision for such a component depends on the component's connectivity. For example, if a component in question is strongly connected to other glue-logic components, then a glue-logic layout style may be more suitable for this component in order to reduce the wiring area between bit-sliced stack and glue-logic module. Third, the component layout style also depends on the final floorplan. For example, using the folded stack architecture, small bit-sliced units are folded into empty space in the stack bounding box. If a small unit doesn't fit into the stack, this unit might better be laid out a glue-logic component in order to reduce the overall layout area. By exploiting the bit-slice property of register-transfer components and selecting the best suited layout style for each component, better area utilization can be achieved.

A weighted and labeled undirected graph **G** is formed from the schematic by a set **U** of nodes and a set **E** of edges. The node $\mathbf{u}_j \in \mathbf{U}$ is the component in the schematic where $\mathbf{j}$ = 1..$\mathbf{n}$, and $\mathbf{n}$ is the total number of components. The sub-node $\mathbf{port(i}_j\mathbf{)} \subset \mathbf{u}_j$ is the port in the component $\mathbf{u}_j$ where $\mathbf{i}$ = 1..$\mathbf{m}$ and $\mathbf{m}$ is the number of ports in the component $\mathbf{u}_j$. The attribute type of a $\mathbf{port(i}_j\mathbf{)}$, $\mathbf{port\_type(i}_j\mathbf{)}$, indicates that $\mathbf{port(i}_j\mathbf{)}$ is a control port or a data port. Let $\mathbf{e(i}_k, \mathbf{j}_l\mathbf{)}$ be the edge between $\mathbf{port(i}_k\mathbf{)}$ and $\mathbf{port(j}_l\mathbf{)}$, where $\mathbf{u}_k$,$\mathbf{u}_l \in$ U, $\mathbf{port(i}_k\mathbf{)} \subset \mathbf{u}_k$, and $\mathbf{port(j}_l\mathbf{)} \subset \mathbf{u}_l$. The weight of an edge $\mathbf{e} \in \mathbf{E}$, $\mathbf{w[e(i}_k\mathbf{,j}_l\mathbf{)]}$, is the number of wires between these two ports. Thus, the edges correspond to connections between ports, while weights are equal to the multiplicity of connections. A graph generated from the schematic in Figure 4(a) is shown in Figure 4(b). There are two components in the schematic, a 4-bit **Reg** and a 4-bit **Mux**. In the graph, **Reg** and **Mux** form two nodes (**U1** and **U2**). Each node has two data ports, ($\mathbf{port(a}_1\mathbf{)}$, ($\mathbf{c}_1$) $\subset$ **U1** and $\mathbf{port(d}_2\mathbf{)}$, ($\mathbf{f}_2$) $\subset$ **U2**), and one control port ($\mathbf{port(b}_1\mathbf{)} \subset$ **U1** and $\mathbf{port(e}_2\mathbf{)} \subset$ **U2**). In Figure 4(b), $\mathbf{e(c}_1, \mathbf{d}_2\mathbf{)}$ is the edge between $\mathbf{port(c}_1\mathbf{)}$ and $\mathbf{port(d}_2\mathbf{)}$ and $\mathbf{w[e(c}_1\mathbf{,d}_2\mathbf{)]}$=4.

A linking cost is used to evaluate the connectivities between components. The linking cost functions are:

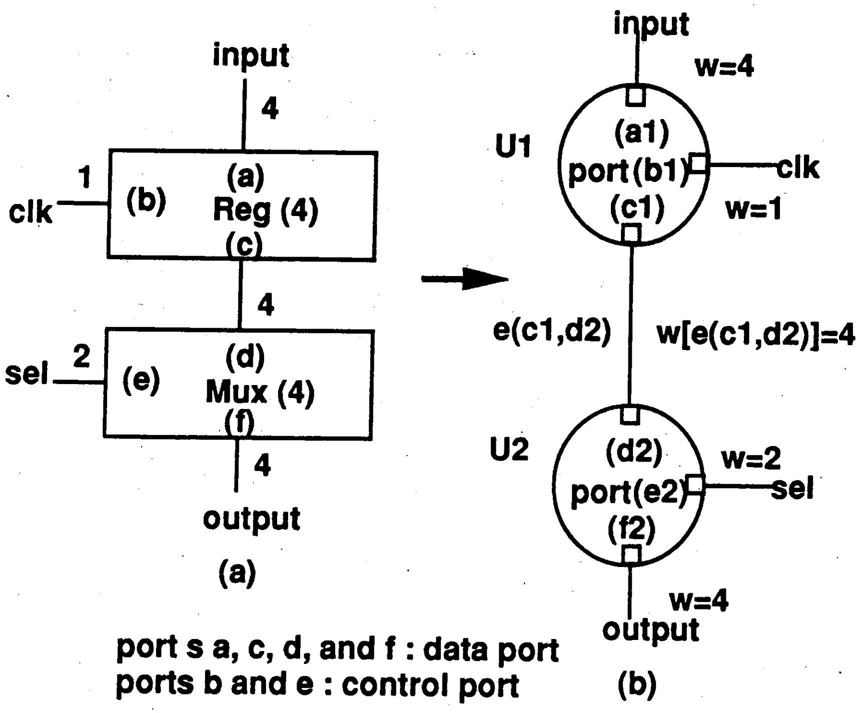

Figure 4. Graph representation of the RT netlist

(1) $W_{control}(u_k) = \sum w[e(i_k, j_l)]$ **for all** u_l where u_l is a **Glue–Logic** or a **Bit–Slice** implementation and **port_type**(j_l) is a control port

(2) $W_{data}(u_k) = \sum w[e(i_k, j_l)]$ **for all** u_l where u_l is a **Bit–Slice** and **port_type**(j_l) is a data port

In the linking cost function, $W_{control}(u_k)$ is the number of wires connected to u_k from other **Glue–Logic** nodes or from control ports of other **Bit–Slice** nodes. $W_{data}(u_k)$ is the number of wires connected to u_k from data ports of other **Bit–Slice** nodes. For example, if U1 in Figure 4(b) is a **Bit–Slice** and **port**(f_2) of U2 connects to another **Bit–Slice** then $W_{control}(u_2)=2$ and $W_{data}(u_2)=8$.

The component partitioning is divided into three steps:

(1) **Initial component type assignments.** The algorithm first labels nodes as **Glue–Logic** if components are not sliceable such as single gates, decoders, encoders, etc. The components that meet the following two conditions are labels as **Bit–Slice:** (i) the component can be laid out as a bit-sliced and (ii) the component's bit widths are larger than a user specified threshold. If these conditions are not meet, the algorithm will label nodes as "**undecided**" component type.

(2) **Component type assignments for the undecided nodes.** The algorithm calculates the linking cost for the undecided nodes. For an undecided node $\mathbf{u}_k$, if $\mathbf{W}_{control}(\mathbf{u}_k) > \mathbf{W}_{data}(\mathbf{u}_k)$ then node $\mathbf{u}_k$ is labeled as **Glue–Logic**, otherwise node $\mathbf{u}_k$ is labeled as **Bit–Slice.**

(3) **Partitioning improvement.** In this step, the algorithm re-evaluates the connectivities among nodes to finalize the component type assignments. The algorithm first calculates the linking cost for all of the nodes. Then the algorithm evaluates the connectivities of nodes that can be laid out as both **Glue–Logic** and **Bit–Slice.** For evaluating a node $\mathbf{u}_k$, there are three possible cases: (i) If $\mathbf{W}_{control}(\mathbf{u}_k) > \mathbf{W}_{data}(\mathbf{u}_k)$ and node $\mathbf{u}_k$ is a **Bit–Slice** then node $\mathbf{u}_k$ is re-labeled as **Glue–Logic,** (ii) If $\mathbf{W}_{data}(\mathbf{u}_k) > \mathbf{W}_{control}(\mathbf{u}_k)$ and node $\mathbf{u}_k$ is a **Glue–Logic** then node $\mathbf{u}_k$ is re-labeled as **Bit–Slice, and** (iii) If conditions (i) or (ii) do not apply, a node $\mathbf{u}_k$ keeps its original component type.

(4) **Floorplan driven component type assignments.** During the stack folding stage, the algorithm may re-label nodes as **Glue–Logic** if the components can not fit into the bit-sliced stack. Details of this phase will be described in the "stack partitioning" section of this paper.

ALGORITHM I. Component Partitioning

```
Component_Partitioning(){
  Build graph from register-transfer netlist;
  /*Initial component type assignment*/
  for component i = 1 to n do{
    if (component i is not sliceable) then{
      type(i) = Glue–Logic;
    }
    else if (component i is sliceable and bit-width(i) > required minimal bit-width for bit-
    sliced units) then{
      type(i) = Bit_Slice;
    }
    else{
      type(i) = undecide;
    }
  }
  /*Assign component type to the undecided nodes*/
  for component j = 1 to n do{
    if (type(j) == undecide) then{
      Calculate linking cost of node j;
      if ($W_{control}(j) > W_{data}(j)$) then{
        type(j) = Glue–Logic;
      }
      else{
        type(j) = Bit–Slice;
      }
    }
  /*Partitioning improvement*/
  for component j = 1 to n do{
    Calculate linking cost of node j;
  }
  for i = 1 to n do{
    if (component i can be laid out as both Bit–Slice and Glue–Logic) then{
      if ($W_{control}(i) > W_{data}(i)$ and type(i) == Bit–Slice) then{
        type(i) = Glue–Logic;
      }
      else if ($W_{data}(i) > W_{control}(i)$ and type(i) == Glue–Logic) then{
        type(i) = Bit–Slice;
      }
```

```
        }
      }
      Floorplan driven component type assignments (see Algorithm II);
    }
```

4. Stack Partitioning

The goal of stack partitioning is to minimize the layout area of the bit-sliced components. Since bit-sliced units often have varying bit-widths, the sliced layout architecture generates an empty space within the stack bounding box. A folding method is used to place small units into the empty space and thus reduce the stack height.

The folding process includes two steps: (i) unit folding and (ii) overlap checking and avoidance. The main constraint of stack folding is that bit-sliced units must not overlap. The algorithm folds one unit at a time. The folding process includes two steps: (i) move the unit $\mathbf{u}_i$ to the right edge of stack's bounding box and rotate it around the center (Figure 5(a)) and (ii) push all of the folded units above the the base-line (Figure 5(b)).

After unit folding, an overlap checking is performed to check whether the units in the folded part overlap with the unfolded part. The bounding box of unit $\mathbf{u}_i$ is defined by the upper-left point $(\mathbf{x}_{ul.i},\mathbf{y}_{ul.i})$ and the lower-right point $(\mathbf{x}_{lr.i},\mathbf{y}_{lr.i})$ of unit $\mathbf{u}_i$ (Figure 5(a)). The overlapping conditions are

(1) There exists a $(\mathbf{x}_{lr.i},\mathbf{y}_{lr.i})$ and $\mathbf{u}_i \in$ {unfolded bit-sliced units};

(2) There exists a $(\mathbf{x}_{ul.j},\mathbf{y}_{ul.j})$ and $\mathbf{u}_j \in$ {folded bit-sliced units};

(3) $\mathbf{x}_{ul.j} < \mathbf{x}_{lr.i}$ and $\mathbf{y}_{ul.j} < \mathbf{y}_{lr.i}$.

If an overlap occurred as shown in Figure 5(b), the folded units will be shifted to the right to avoid the overlap (Figure 5(c)).

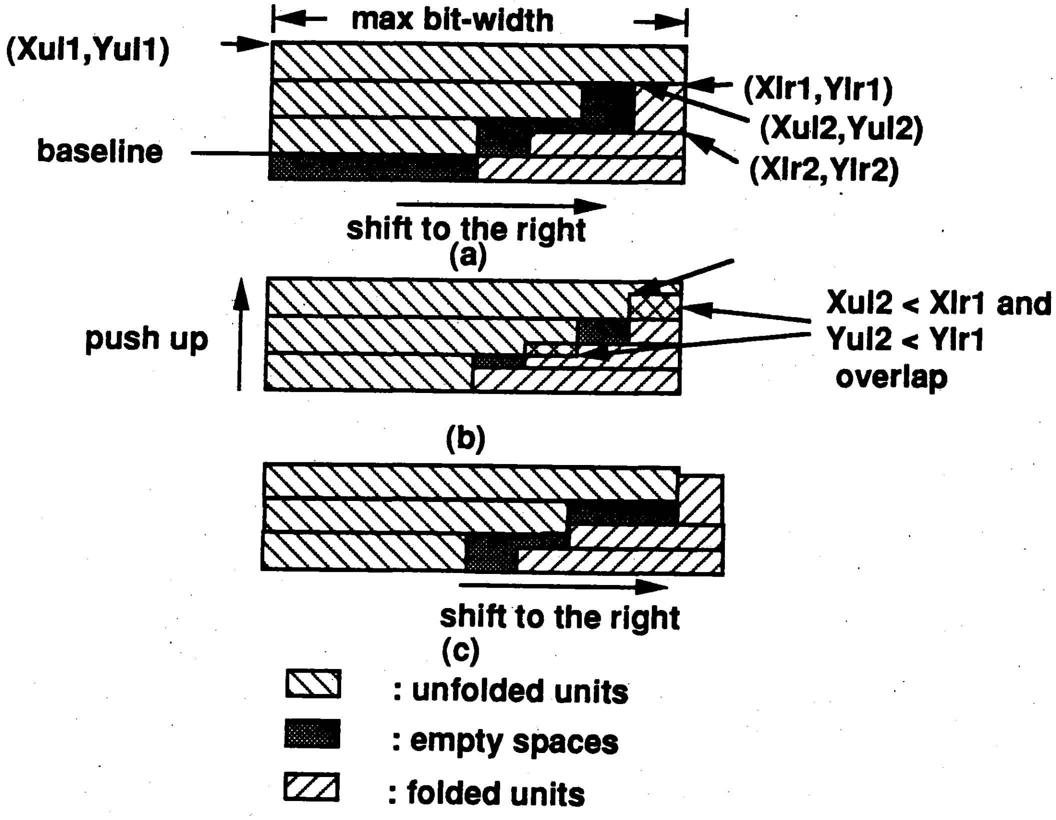

Figure 5. Stack folding process

Using the folding method, we describe a stack partitioning algorithm for minimizing the area of the bit-sliced units as follows:

(1) Calculate the routing area cost between unfolded and folded units. The routing area is in proportion to the number of wires between unfolded and folded units. For instance, the number of wires crossing the cutline between **compA** and **compB** is four (Figure 6(a)). By folding **compB**, the routing area will contain four wires (Figure 6(b)). The number of wires crossing any cutline can be determined after the unit placement and routing. The unit placement and routing consists of three

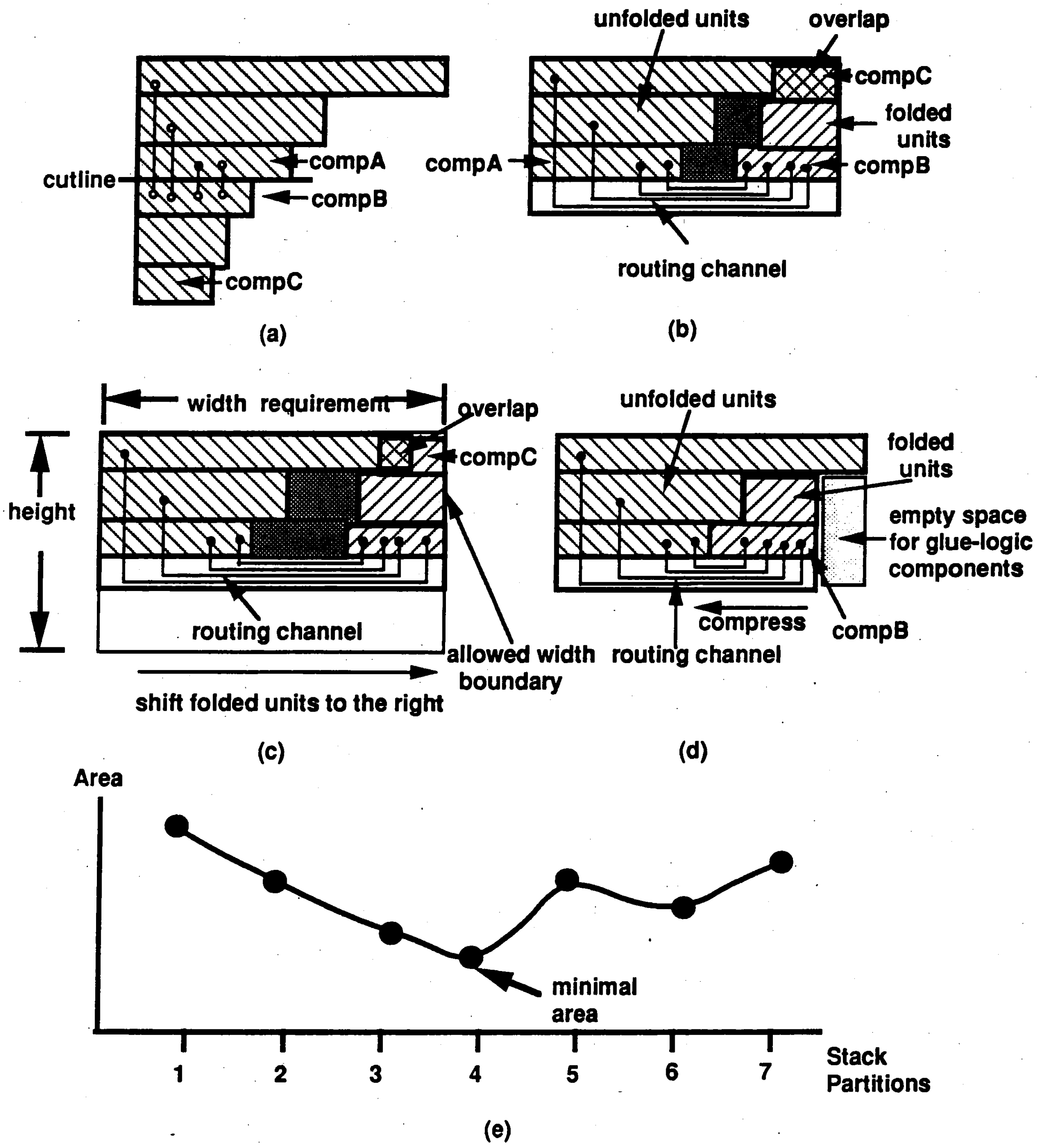

Figure 6. Stack partitioning based on folding

steps: (i) Sort the units by width, (ii) Permute the order of the units of the same width to minimize the track density, and (iii) Assign routing tracks to the units. Thereafter, the number of wires between units is calculated.

(2) The algorithm uses our folding method to minimize the area of the bit-sliced units subject to the given stack height and width requirements. If an overlap occurs during the folding process, the folded units will be shifted to the right to avoid overlap until the required width is reached. However, if some folded units still overlap the unfolded units after shifting, the overlapped folded units will be deleted to form a new stack. In Figure 6(c), **compC** overlaps with unfolded units when allowed width is reached. Thus, it will be moved to form a separate stack. Moreover, the stand-alone or leftover small bit-sliced units that do not fit in any stacks will be moved to the glue-logic module (i.e. it will relabeled **compC** as **Glue–Logic** as mentioned in the previous section "component partitioning" of this paper).

(3) The algorithm calculates the total stack area. There are two cost functions: (i) For one stack, the total area is the minimal bounding box that encloses all of the units and the routing area for connecting the unfolded and the folded units and (ii) For multiple stacks, the total area is the sum of bounding boxes of individual stacks and the routing area between stacks. The algorithm executes repeatedly and selects the best stack partition with the minimal area that also satisfies the stack height and width constraints. For example, Figure 6(e) shows an area curve that was generated by executing the folding process repeatedly. Each data point in Figure 6(e) represents the total area for a particular stack partition. The partition

with the minimal area (partition #4 in Figure 6(e)) is selected as the final stack partition.

(4) After selecting the stack partition, the algorithm performs a width compression step to reduce the empty space between the unfolded and folded units (Figure 6(d)). The empty space in the bounding box can be used for placing the glue-logic components.

ALGORITHM II. Stack Partitioning

```
Stack_Partitioning(){
  Sort bit-sliced units in descending order according to bit-width;
  Perform unit placement and routing;
  Calculate cut-lines between units and empty space in the bounding box;
  if (empty space is too large for glue-logic
    or stack height > the allowed stack height) then{
   for unit 1 to n do{
    Perform unit folding;
    if (unfolded units overlap with folded units) then{
     Shift folded units to avoid overlap;
     if (stack width > the allowed stack width) then{
      Shift folded units to the allowed width boundary;
      Move the overlapping folded units to form a new stack;
     }
    }
    Choose the minimal area partition;
   }
   Move stand-alone and small units into Glue-Logic module;
   Compress unfolded and folded units;
  }
}
```

5. Glue-Logic Partitioning

After forming the bit-sliced stacks, the floorplanner first places the stacks according to a given layout height, width, and aspect ratio. The glue-logic partitioning algorithm places the glue-logic components into the empty space in the stack bounding

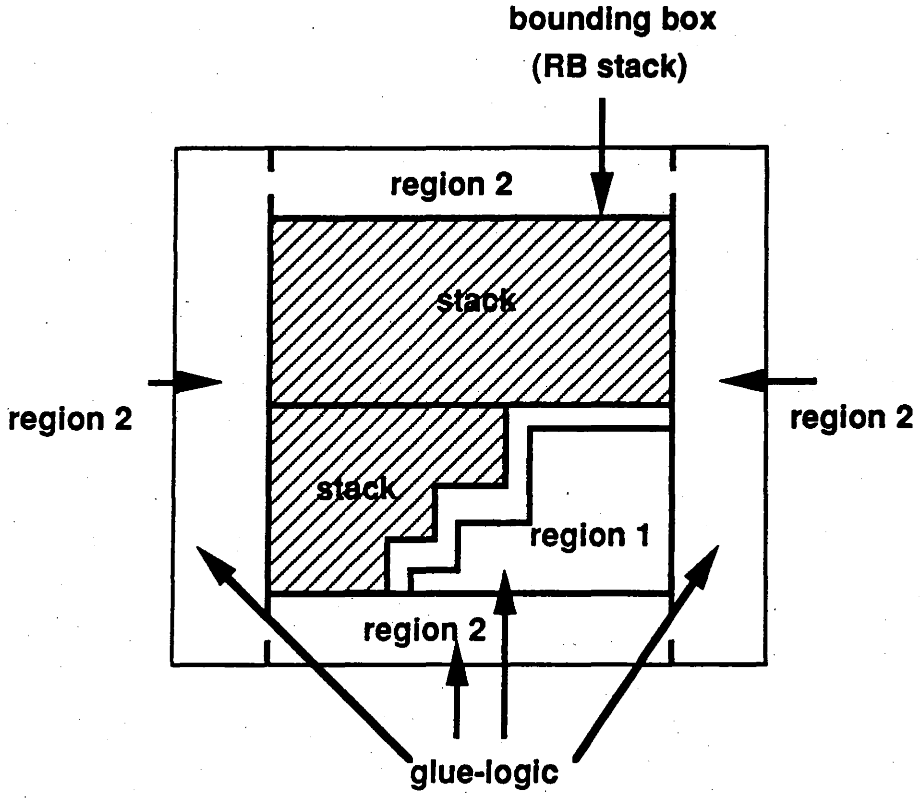

Figure 7. Placement of glue-logic components
in regions 1 and 2

box (region 1) and around the stack bounding box (region 2) according to the given total aspect ratio and I/O pin location (Figure 7).

The glue-logic partitioning algorithm consists of three steps: (i) Block partitioning with layout estimation, (ii) Seed implantation and (iii) Multiway seed partitioning. The block partitioning algorithm partitions the rectilinear area around the bit-sliced stacks into rectangular blocks with estimated block sizes (number of transistors) based on the required aspect ratio and the number of transistors in the glue-logic module. For example, if the glue-logic module is of size n and the given aspect ratio is 1:1 (Figure 8), the algorithm partitions the empty space into block1, block2, and block3 with block sizes of m1, m2, and m3, respectively, where n=m1+m2+m3.

In order to minimize the wire lengths on the critical paths, all components on the critical path must clustered together. Moreover, the components connected to the stack and the external I/O ports should be placed in blocks that are as close as possible to the connecting ports. The components on the critical path as well as the components connected to either the stack or the external I/O ports are called seed components. Seed implantation places the seed components into proper blocks in order to minimize wire lengths. For instance, consider a critical path connected to the control line of a

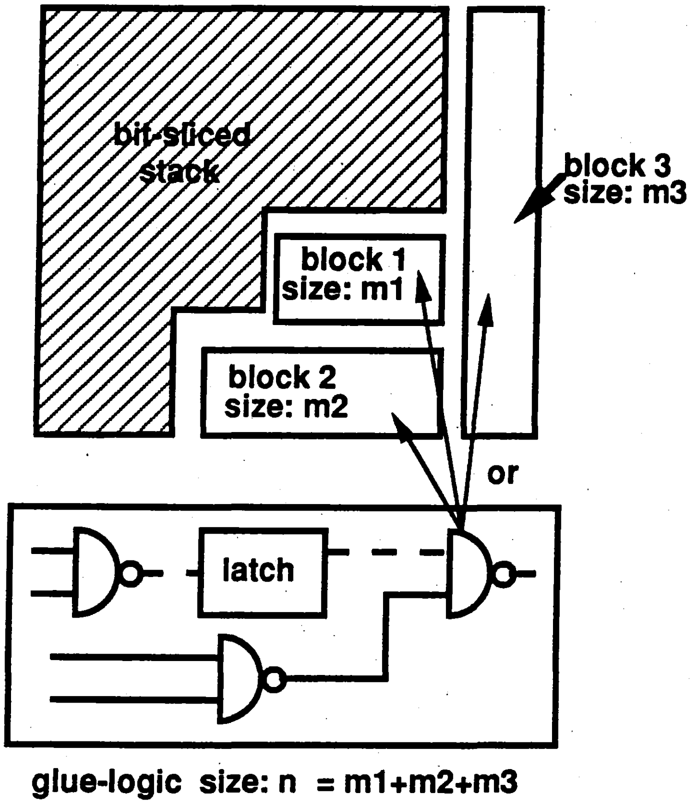

Figure 8. The example of the glue-logic of size n being partitioned into block 1, 2, and 3 with size m1, m2, and m3.

bit-sliced stack is placed in block1, and a glue-logic component, **compA**, connected to the right external I/O port is placed in block2 as shown in Figure 9(a). Without clustering and seed implantation, the critical path could be distributed over both blocks, while **compA** could be placed into block1. That would introduce a long wire which contributes to increase in signal delay (Figure 9(b)). Using seed implantation, however, the seeds are forced to reside in blocks with the shortest distance to the connecting ports. In our example the critical path is forced into block1 and **compA** is forced into block2 (Figure 9(a)).

Finally, the multiway seed partitioning algorithm partitions glue-logic components into pre-defined blocks according to block sizes, I/O pin positions, and critical paths. Moreover, pin assignment for the glue-logic blocks is performed to minimize the wire crossing between blocks.

5.1. Block Partitioning with Layout Estimation

Since the bit-sliced units often have varying bit widths, the empty space in the bounding box forms a ladder-shaped rectilinear geometry as shown in Figure 10. The block partitioning algorithm first partitions the empty space in the stack bounding box into rectangles. This empty space can be decomposed into rectangles of size $\mathbf{w_i} \times \mathbf{h_i}$ where $\mathbf{w_i}$ and $\mathbf{h_i}$ are the width and height of each rectangle **i**, for $\mathbf{1 \leq i \leq n}$, where **n** is the number of rectangles.

The rectangles are sorted in descending order by width. The algorithm first places glue-logic components into the widest rectangle ,REC_A, as shown in Figure 10. Transistors can be placed into rows with vertical or horizontal orientation. Using the horizontal orientation, the **dead-space** indicated as **spaceA** in Figure 10 will be wasted.

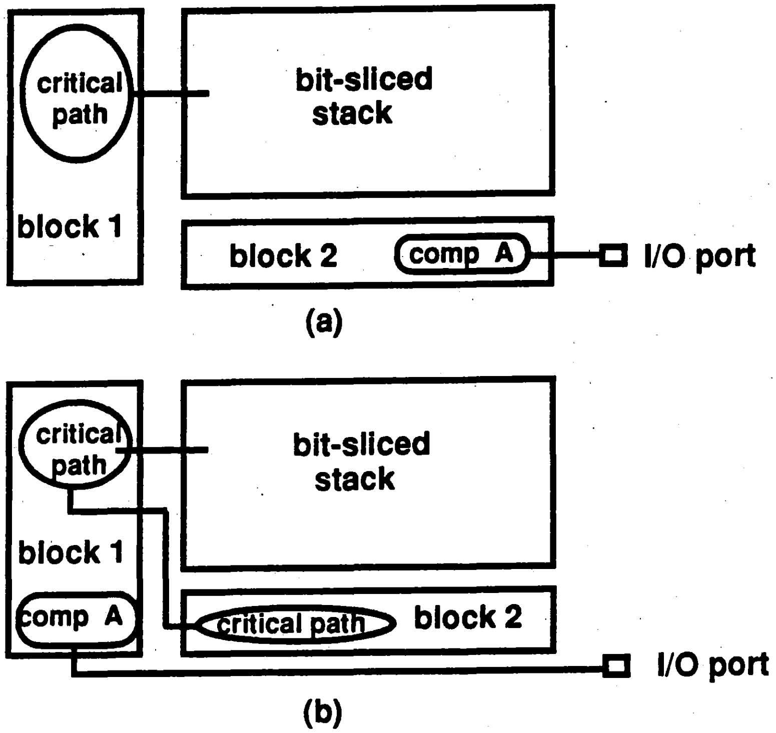

Figure 9. (a) Proper placement through clustering and seed implantation (b) Possible placement without clustering and seed implantation

Using the vertical orientation, however, a portion of life-space indicated by spaceB in Figure 10 can be propagated to the next widest rectangle, REC_B, as shown in Figure 10.

In our implementation, we use LES [LiGa87] for glue-logic block generation. In the LES architecture, transistors are arranged in a horizontal strip and wires are connected between the P and N transistor rows. The area estimation for the LES architecture is provided by an estimator embedded in the database [ChGa90]. By considering the

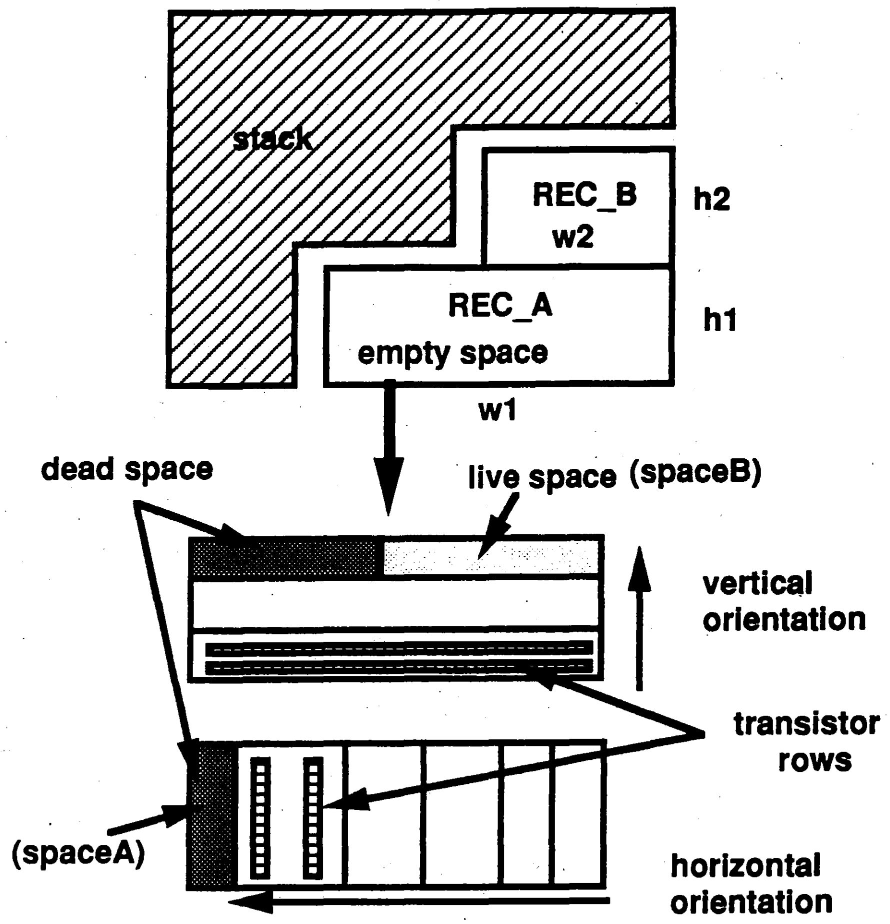

Figure 10. Two orientations of transistor placement

routing area between transistors rows, the estimator provides area utilization and block size information according to the rectangular size. Both orientations are tried on each rectangle. The algorithm selects the one with the highest area utilization. This process continues until the glue-logic components fill the entire ladder-shaped area.

If more space is needed, the algorithm then uses a constructive method to define the blocks around the bounding box according to the given aspect ratio and size constraints. The overall aspect ratio and size constraints are:

$W_{constraint} \geq W_{module}$

$\mathbf{H}_{constraint} \geq \mathbf{H}_{module}$

$\mathbf{Aspect\_Ratio}_{constraint} = \mathbf{W}_{module} / \mathbf{H}_{module}$

An example with a given 1:1 aspect ratio is shown in Figure 11. After filling block1 and block2, the algorithm first places glue-logic components into block3. If more space is needed, the algorithm then places components into block4. The process continues until the glue-logic components are all placed into blocks. Since block3 can be also placed on the left of the bounding box while block4 can be placed on the top of the bounding box, a connectivity evaluation is performed to determine the block locations.

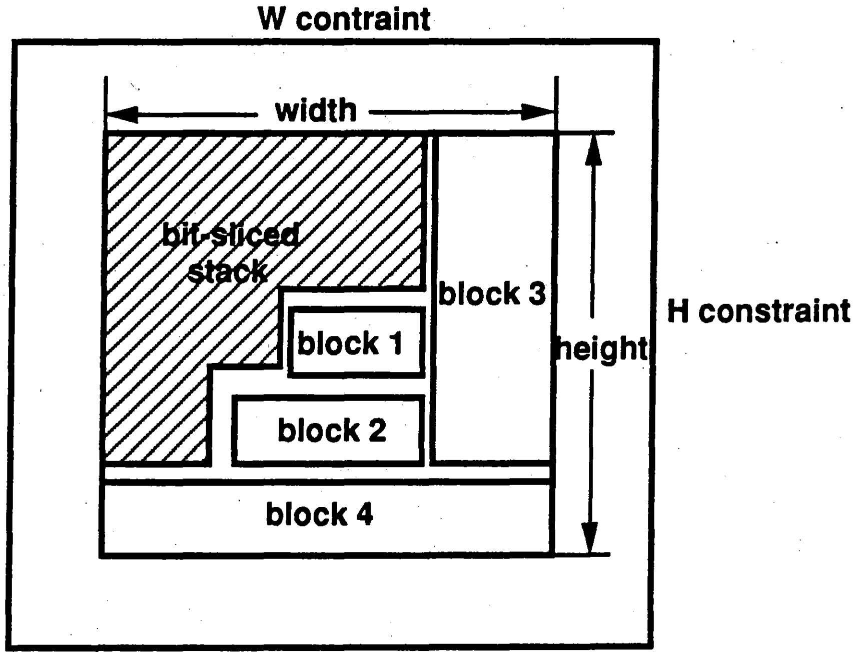

Figure 11. Glue-logic block partitions according to given size constraints

For instance, if there are 10 bottom external I/O ports and 2 top external I/O ports in the glue-logic module, then block4 is more suitable to be placed on the bottom of the bounding box.

Let M_i be the glue-logic block **i** which is denoted by a 4-tuple $<s_i, w_i, h_i, o_i>$ where s_i is the number of transistors in block **i**, w_i is the block width, h_i is the block height, and o_i is the orientation. Let $M=\{M_i | M_i=<s_i, w_i, h_i, o_i>\}$ be the set of glue-logic blocks.

ALGORITHM III. Glue-Logic Block Partitioning

```
Glue_Logic_Block_Partitioning(){
  M = $\phi$ ;
  /*Define the blocks in the stack bounding box*/
  Sort rectangles in descending order according to width;
  for rectangle 1 to n do{
    if (total # of transistors in Glue-Logic > 0) then{
      Perform vertical and horizontal placement;
      M $\cup$ the block with the highest area utilization;
      Update the total # of transistors in Glue-Logic;
      if (vertical orientation is selected) then
        Pass live_space to the next rectangle;
    }
  }
  /*Define the blocks around the stack bounding box*/
  while (total # of transistors in Glue-Logic > 0) do{
    Define the empty space according to the given aspect ratio and module sizes;
    Place transistor rows into empty space;
    Update the total # of transistors in Glue-Logic;
  }
  M $\cup$ blocks around the stack bounding box;
}
```

5.2. Seed Implantation

Seed implantation uses a multi-stage clustering method [Hohn67] to cluster seeds for each glue-logic block. The glue-logic blocks can be divided into four sections: top, bottom, left, and right (Figure 12(a)). We define four clustering groups: **glue-logic-top**, **glue-logic-bottom**, **glue-logic-left**, and **glue-logic-right**. Each group contains the glue-logic components connected to the same side of the stack or the external I/O ports. For example, consider two glue-logic components **comp**A and **comp**B such that **comp**A connects to the left external I/O port and **comp**B connects to the left boundary of the stack (Figure 12(a)). Then both components should be placed in the left section. Therefore, **comp**A and **comp**B are both clustered into the group **glue–logic–left** (Figure 12(c)). In the first clustering stage, glue-logic clustering is based on five criterions: (i) **critical–path**, (ii) **glue–logic–top**, (iii) **glue–logic–bottom**, (iv) **glue–logic–left**, and (v) **glue–logic–right** (Figure 12(c)). If a component **GL** can be placed in more than one group, the following rules are applied:

(1) If **GL** is on the critical path then $\mathbf{GL} \subset C_{critical\_path}$, where $C_{critical\_path}$ is the critical path cluster.

(2) Otherwise, the component clustering depends on the number of wire connections between **GL** and the four different glue-logic sections (top, bottom, left, and right). **GL** will be assigned to the cluster with the maximum connections.

In the second stage, the algorithm continues to cluster the critical paths to determine the preferred location for the critical paths. There are two cases. In the first case, if $C_{critical\_path} \cap$ **glue–logic–section** $= \phi$ where **section** $\in$ {**top**, **bottom**, **left**, or **right**}, then $C_{critical\_path}$ is not strongly connected to any sections. This $C_{critical\_path}$ becomes a stand alone cluster and will not be implanted into blocks as a seed. In the second case, if $C_{critical\_path} \cap$ **glue–logic–section** $\neq \phi$ then the rule (2) in the first stage is used to determine the new clusters for $C_{critical\_path}$. For example, in Figure 12(a) there is a critical path that strongly connects to the left section. Therefore, the critical path cluster and the **glue–logic–left** cluster are clustered further (Figure 12(d)). In the third stage, the seed clusters are grouped into blocks. For example, there are two glue-logic blocks,

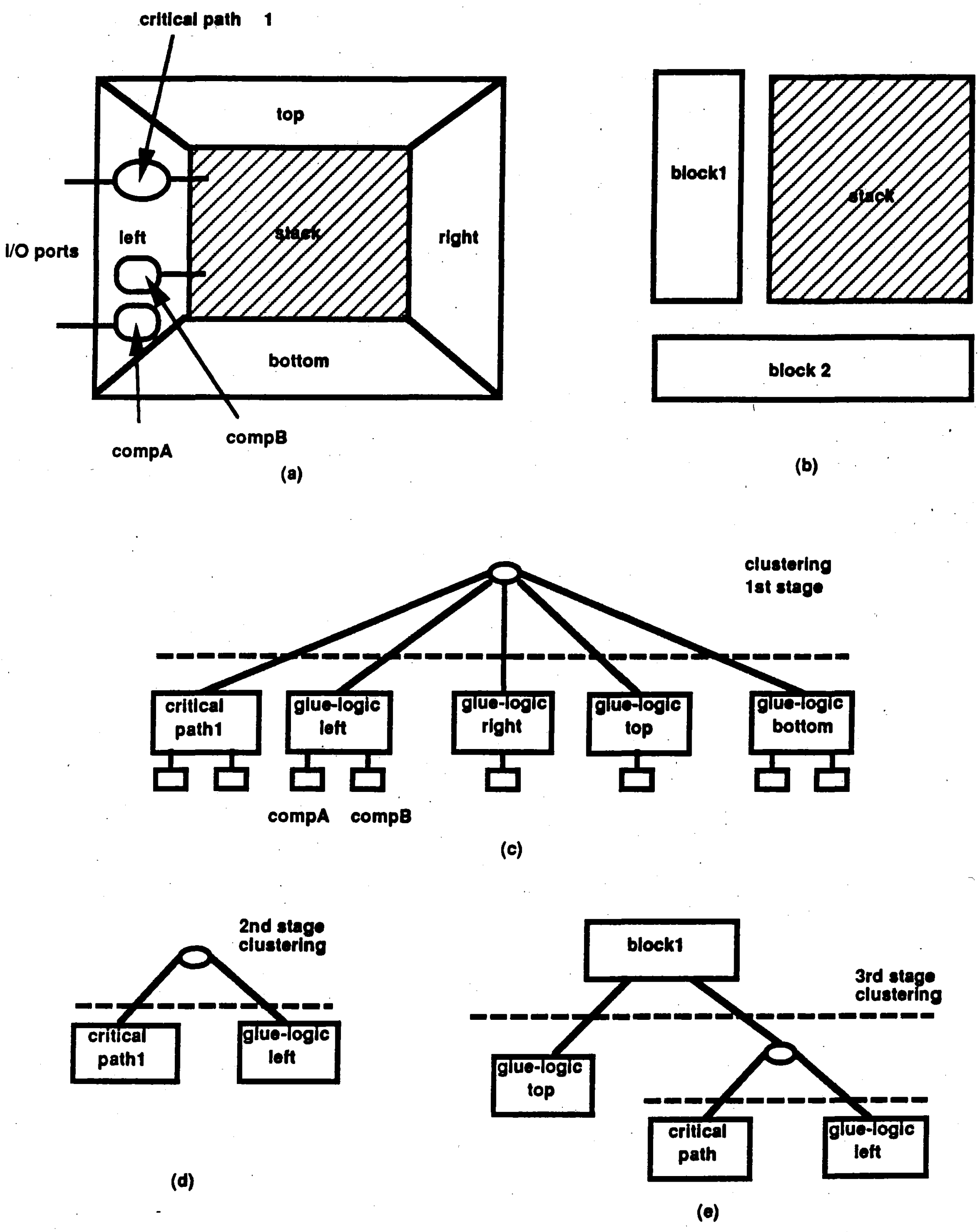

Figure 12. Seed implantation using a multi-stage clustering method

blockl and block2, in the layout module (Figure 12(b)). Obviously, the **glue–logic–left** cluster should be clustered into blockl (Figure 12(e)). Ideally, the **glue–logic–top** cluster should reside in the glue-logic block on the top section. However, if there is no glue-logic block on the top section then the **glue–logic–top** cluster will clustered with blockl that is close to the top section (Figure 12(e)).

5.3. Multiway Seed Partitioning

The multiway seed partitioning method is an extension of the min-cut partition algorithm [KeLi70]. The original two-way uniform partition is to find a minimal-cost

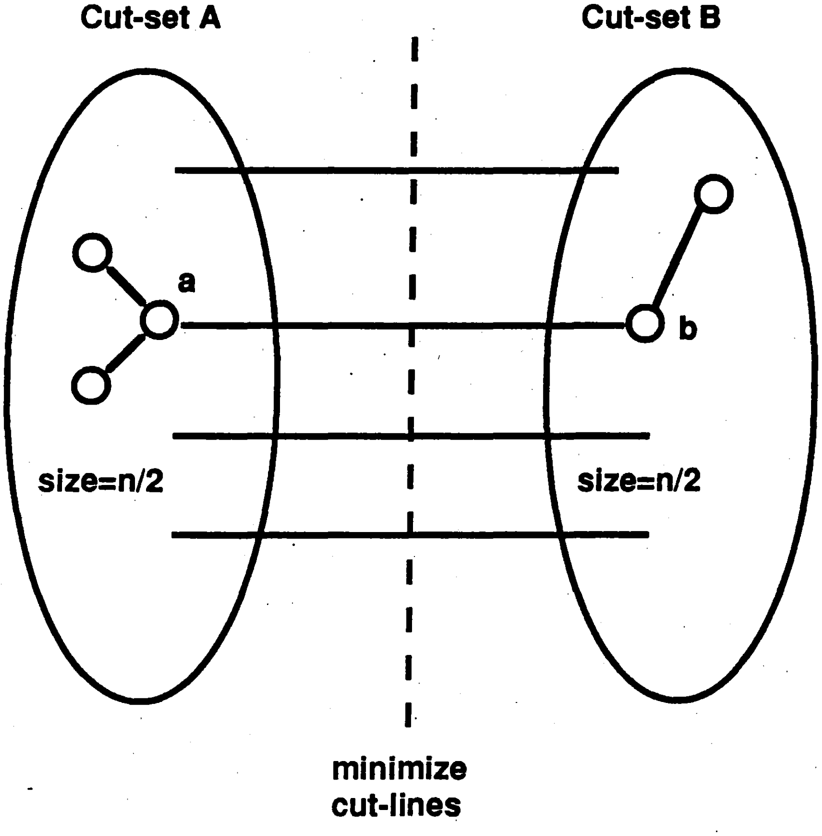

Figure 13. Bi-partition graph

partition of a given graph of **n** nodes connected by edges into two equal subsets of **n/2** nodes (Figure 13). The min-cut partition algorithm uses heuristics to find the minimal partition by exchanging the elements in cut-set A and cut-set B based on a partition cost function. Let's define $\mathbf{I}_i$ as the number of wire connections between element **i** and other elements in the same cut-set, and $\mathbf{E}_i$ as the number of wire connections between element i and other elements in the different cut-set. For example in Figure 13, $\mathbf{E}_a$ is 1 and $\mathbf{I}_a$ is 2. Then, $\mathbf{D}_i$ is the difference between external and internal costs of element **i**, that is, $\mathbf{D}_i = \mathbf{E}_i - \mathbf{I}_i$. $\mathbf{C}_{ab}$ is a double counting correction function if element A and element B are connected to the same net. The partitioning cost ($\mathbf{gain}_{ab}$) calculation for exchanging element A and element B is $\mathbf{D}_a + \mathbf{D}_b - 2\mathbf{C}_{ab}$. If $\mathbf{gain}_{ab}$ is positive then the total cost can be reduced by $\mathbf{gain}_{ab}$ by swapping element A and element B. In each partition pass, it obtains a sequence of gains $\mathbf{gain}_1,\ldots,\mathbf{gain}_n$ with corresponding swapped pairs. **K** swapped pairs are selected to maximize the total gain **G(k)** where $\mathbf{G(k)} = \sum \mathbf{gain}_i$ for $1 \leq \mathbf{i} \leq \mathbf{k}$. The partitioning process continues until $\mathbf{G(k)} \leq 0$.

In our case, the goal of multiway seed partitioning is to partition glue-logic components into blocks with minimal cut-lines between blocks subject to block size constraints. To achieve the minimal wire lengths of global nets, it is not adequate to partition glue-logic components into blocks without considering the external connections [DuKe85, LaDi86]. For instance, by swapping element X and element Y (Figure 14), the partitioning cost increases by 1 if the external connections between the elements in the block A and the element X in block B are not taken into account. However, the actual partitioning cost decrease by 1 when the external connections are taken into account.

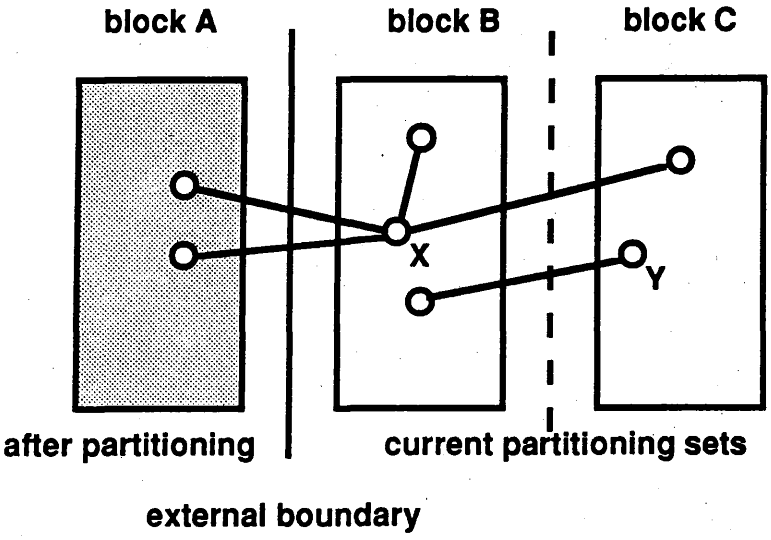

Figure 14. Partition cost including the external connections

To take the external connections between blocks into account, a penalty external cost $\mathbf{PE}_{cost}$ is added to calculate the partitioning cost. $\mathbf{PE}_{cost}$ can be (i) zero, (ii) negative, or (iii) positive. For example in case (i) (Figure 15(a)), block B and block C are adjacent to the block A. Thus, $\mathbf{PE}_{cost}$ can be set to zero by swapping element X and element Y. In case (ii) (Figure 15(b)), if element X connects to the block A and the block D is not adjacent to the block A, $\mathbf{PE}_{cost}$ will be made negative by swapping element X and element Y (it needs one more extra vertical routing track). In case (iii) (Figure 15(c)), if element X connects to the block B and block D is adjacent to the block A, $\mathbf{PE}_{cost}$ will be made positive by swapping element X and element Y (it reduces

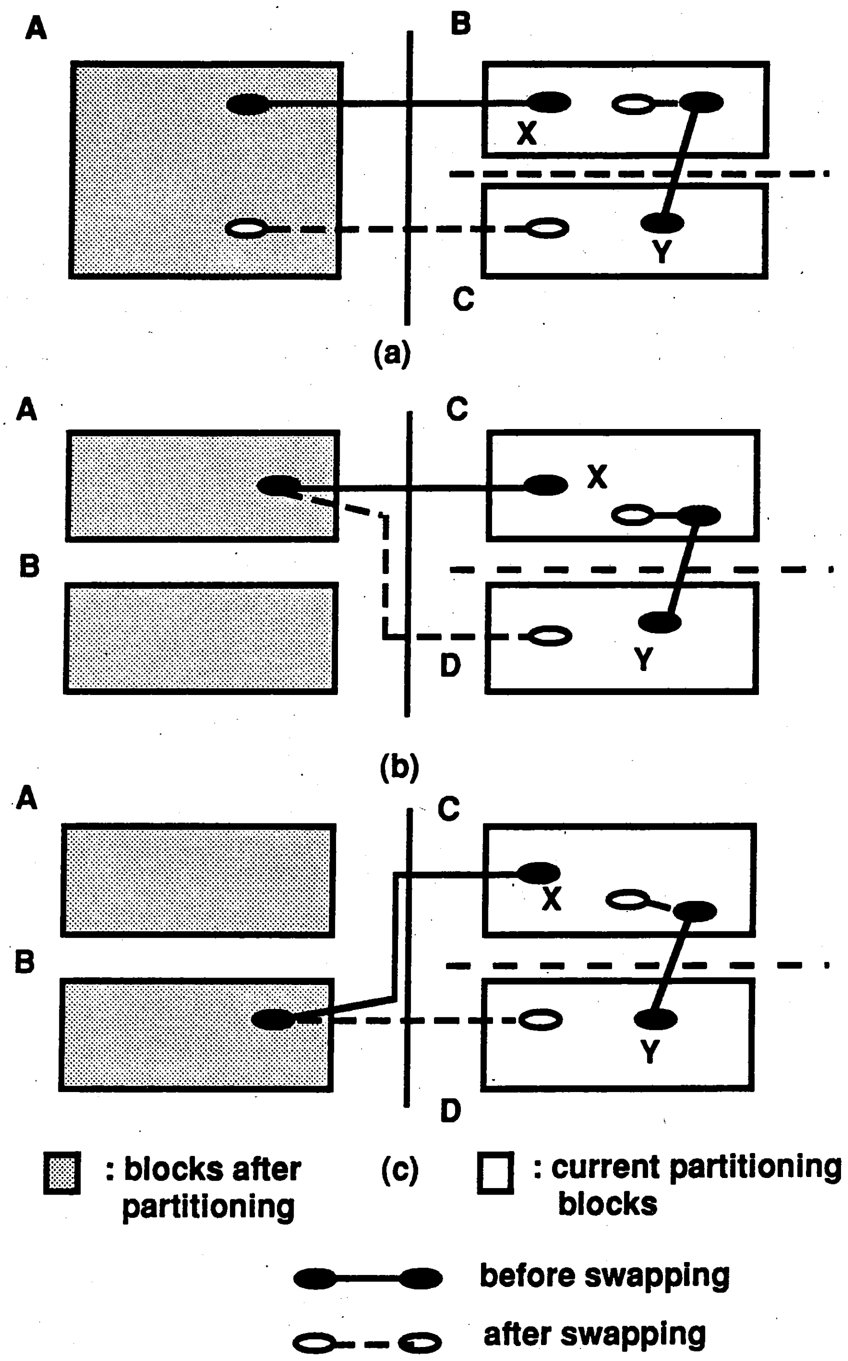

Figure 15. Partition cost calculation based on external connections

one vertical routing track).

As a result, the total partitioning cost $\mathbf{gain}_{ab}$ for swapping element A and element B is $\mathbf{D}_a + \mathbf{D}_b - 2\mathbf{C}_{ab} + \mathbf{PE}_{cost}$.

To consider the I/O port locations, the critical paths, and the connections between blocks, a seed partitioning scheme is used to ensure the proper partitioning decisions for reducing the wire lengths of global nets. The idea of seed partitioning is to force certain components to be placed in the particular blocks as seeds. During the partitioning process, the seeds are treated as non-swapable cells. Using the seeding strategy, the critical paths will be tightly clustered and placed in the same block close to the connecting ports. The wire lengths of the critical paths is thus reduced.

In the glue-logic block partitioning step, the glue-logic module is divided into blocks. Let **n** be the number of pre-defined glue-logic blocks and $\mathbf{S}_i$ be the size of block **i.** Thus, the size of glue-logic module $\mathbf{S}_{Glue-Logic} = \sum \mathbf{S}_i$ for $1 \leq \mathbf{i} \leq \mathbf{n}$.

The multiway seed partitioning algorithm performs min-cut partitioning repeatly based on the cut-set size of $(\mathbf{C}_A, \mathbf{C}_B)$ where $\mathbf{C}_A = \mathbf{S}_i$ and $\mathbf{C}_B = \sum \mathbf{S}_j$ for $1 \leq \mathbf{i} \leq \mathbf{n}$ and $\mathbf{i+1} \leq \mathbf{j} \leq \mathbf{n}$. Since there are pre-implanted seeds in each block, the seeds in the cut-sets need to be rearranged before the partitioning process takes place. Let $\mathbf{SEED}_i$ be the seeds in block **i.** In every partitioning iteration, the seeds are rearranged as $\mathbf{SEED}_i \subset \mathbf{C}_A$ and $(\mathbf{SEED}_{i+1} \cup \mathbf{SEED}_{i+2} \cup \ldots \cup \mathbf{SEED}_n) \subset \mathbf{C}_B$ for $1 \leq \mathbf{i} \leq \mathbf{n}$. Frequently, $\mathbf{C}_A$ and $\mathbf{C}_B$ are not equal; therefore, a set of dummy elements are added to the original set to allow unbalanced partitioning.

ALGORITHM IV. Glue-Logic Partitioning

```
Glue_Logic_Partitioning(){
  /*Block partitioning*/
  Glue_Logic_Block_Partitioning();
  /*Seed implantation*/
  Cluster components as seeds based on the critical path and the external I/O pin positions;
  Build multi-stage clustering tree to place seeds into blocks;
  /*Multiway seeding partitioning*/
  for block i=1 to n do{
    /*Arrange seeds for cut-sets*/
    $\mathbf{SEED}_i \subset \mathbf{Cut\text{-}Set}_A$;
    $\mathbf{SEED}_{i+1} \cup \mathbf{SEED}_{i+2} \cup \ldots \cup \mathbf{SEED}_n \subset \text{Cut-Set}_B$;
    /*Calculate cut-set sizes*/
    $\mathbf{Size(Cut\text{-}Set}_A) = \mathbf{Size(Block}_i)$;
    $\mathbf{Size(Cut\text{-}Set}_B) = \sum \mathbf{S}_k$ for $\mathbf{i+1 \leq k \leq n}$;
    /*Min-cut partitioning*/
    Partition Glue-Logic components into $\mathbf{Cut\text{-}Set}_A$ and $\mathbf{Cut\text{-}Set}_B$;
  }
  for block i=1 to n do{
    I/O pin order assignments for Block i;
  }
}
```

6. Experimental Results

The previously described algorithms are embedded in SLAM [WuCG90] which currently runs on SUN3/SUN4 workstations under the UNIX operating system. Several examples have been tested. The layouts were generated using a 3-micron CMOS technology.

The first example is a controlled counter [Arms89] that consists of approximately 50% sliceable components and 50% non-sliceable components. The register-transfer schematic (**Figure** 16) was generated by **VSS** [LiGa88]. SLAM partitions the LIM register and the up/down counter into bit-sliced units and partitions the glue-logic module into two blocks to satisfy a given 1:1 aspect ratio requirement. The final layout is shown in Figure 17. The second example is the digital section of a DSP chip supplied by local industry that consists of an ALU, registers, flip-flops, a shifter, counters, latches, and simple gates. The final layout which consists of one bit-sliced stack and three glue-logic blocks is shown in Figure 18. The third example is the MARK1 simple computer [SiBN82] which includes 32, 16, 13, and 3 bits register-transfer components and simple gates. The final layout is shown in Figure 19. It consists of a unfolded stack and a folded stack with a glue-logic block.

Using the same layout system, we compare our method with a manual floorplanning and automatic placement and routing method for layout generations of the previous examples. Using the second method, the register-transfer schematic was first partitioned into blocks. For example, the original DSP design was partitioned into three parts: program counter, control unit, and arithmetic unit. Each block was generated individually. Then we used the GDT's interactive floorplanner to find the best floorplan and to determine the shape and I/O pin locations of the glue-logic block. The results in Table 1 show that the layouts generated using our layout method are 10% better than those using the manual floorplanning and automatic placement and routing method. Moreover, the wire lengths on the critical path produced using our layout architecture are 30%-50% shorter than those produced using the manual floorplanning and automatic placement and routing method.

7. Conclusions

In this paper, we have presented a new layout style for netlists with sliceable components. The empty space created by abutting the bit-slices of different bit-widths can be fully utilized by filling it with glue-logic components or folding the bit-sliced stack. As a result, better area utilization can be achieved using this sliced layout architecture.

We also presented partitioning algorithms for layout generation of register-transfer netlists. The partitioning algorithms not only select the best suited layout style for each component, but also consider critical paths, I/O pin locations, and connections between blocks. This improves the overall area utilization and minimizes the total wire lengths.

Future work includes developing different stack folding algorithm such as interleaved folding method and investigating timing issue based on our layout architecture.

8. Acknowledgements

This work was supported by NSF grant #MIP-8711025, California MICRO grant #88-134, and contributions from Rockwell International, Western Digital, Silicon Systems, and S-MOS Inc. We are grateful for their support.

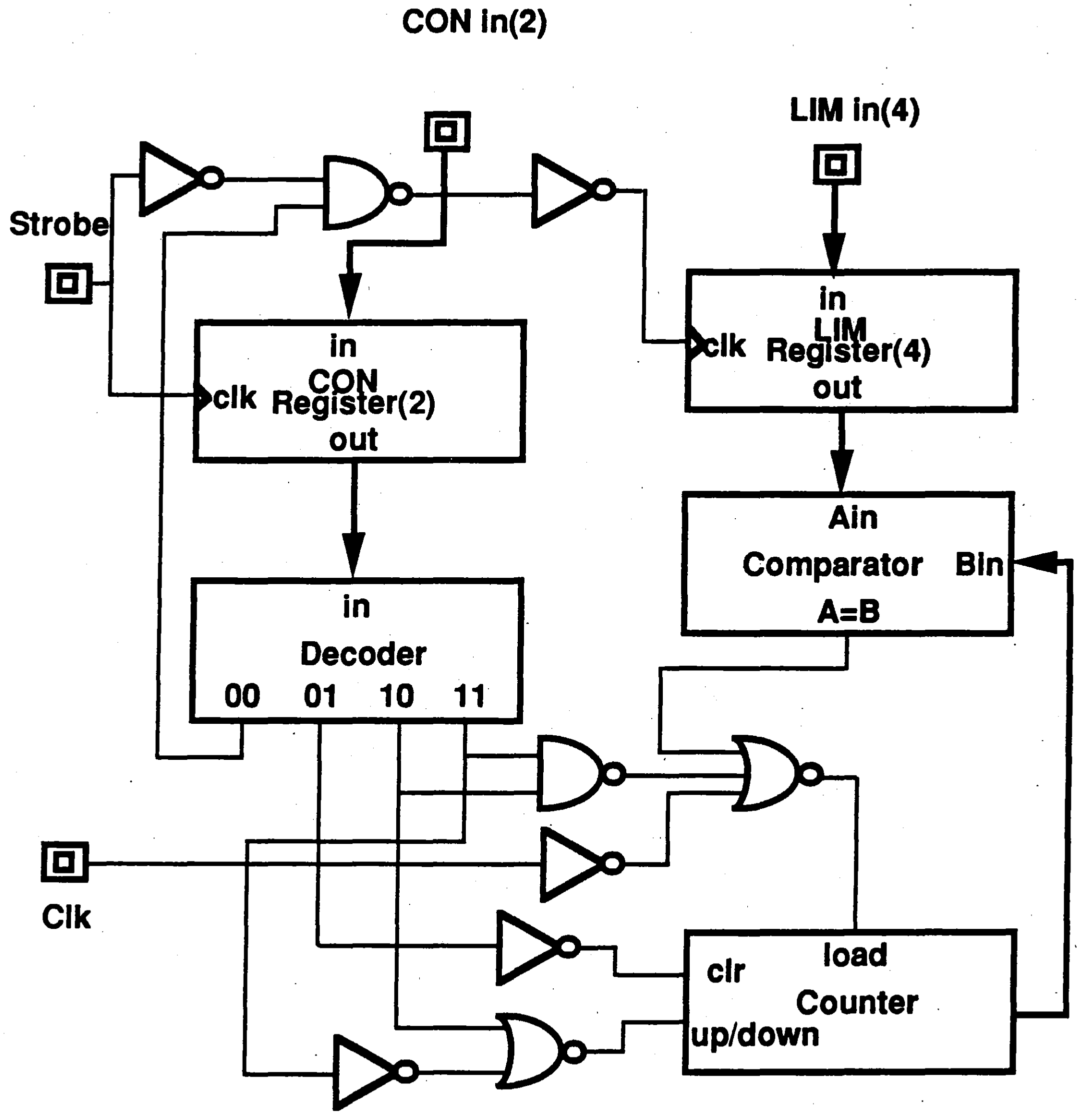

Figure 16. The controlled counter schematic

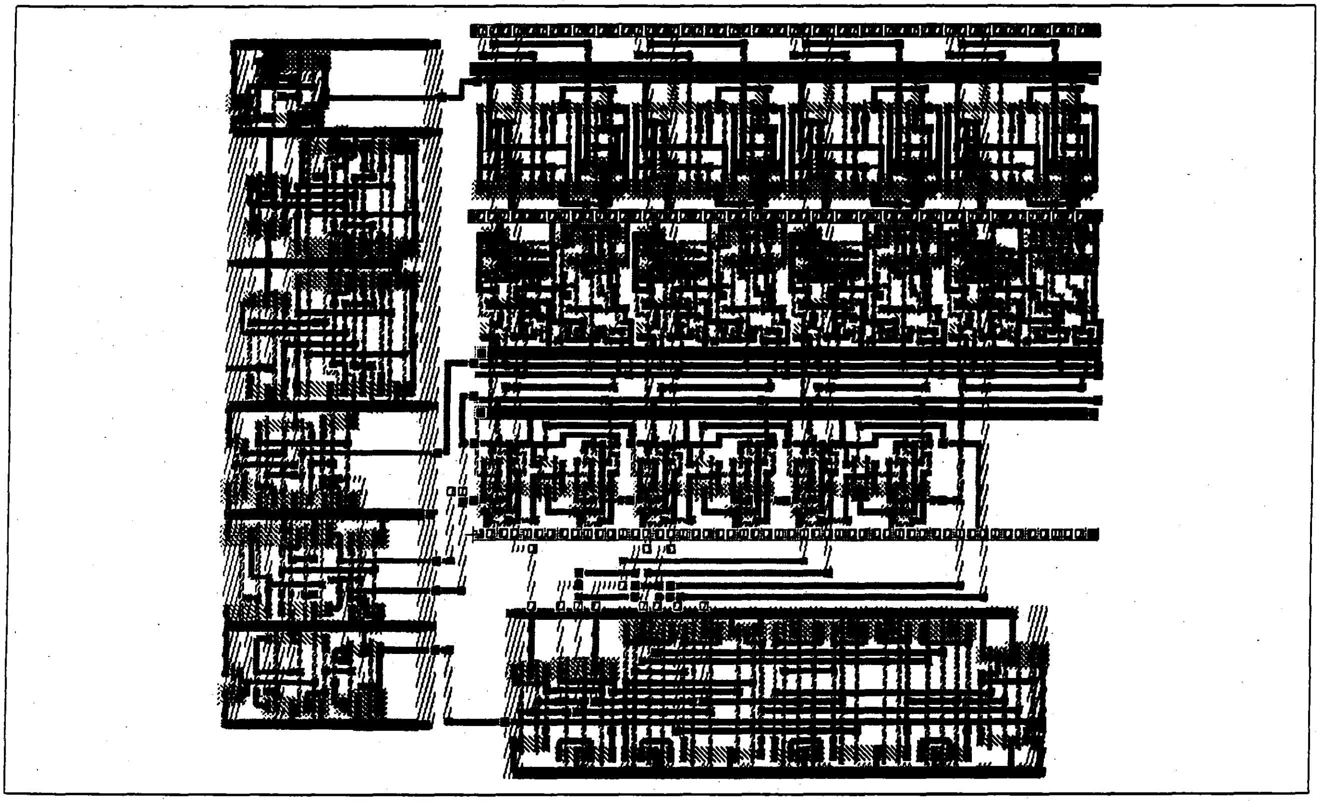

Figure 17. The layout of the controlled counter

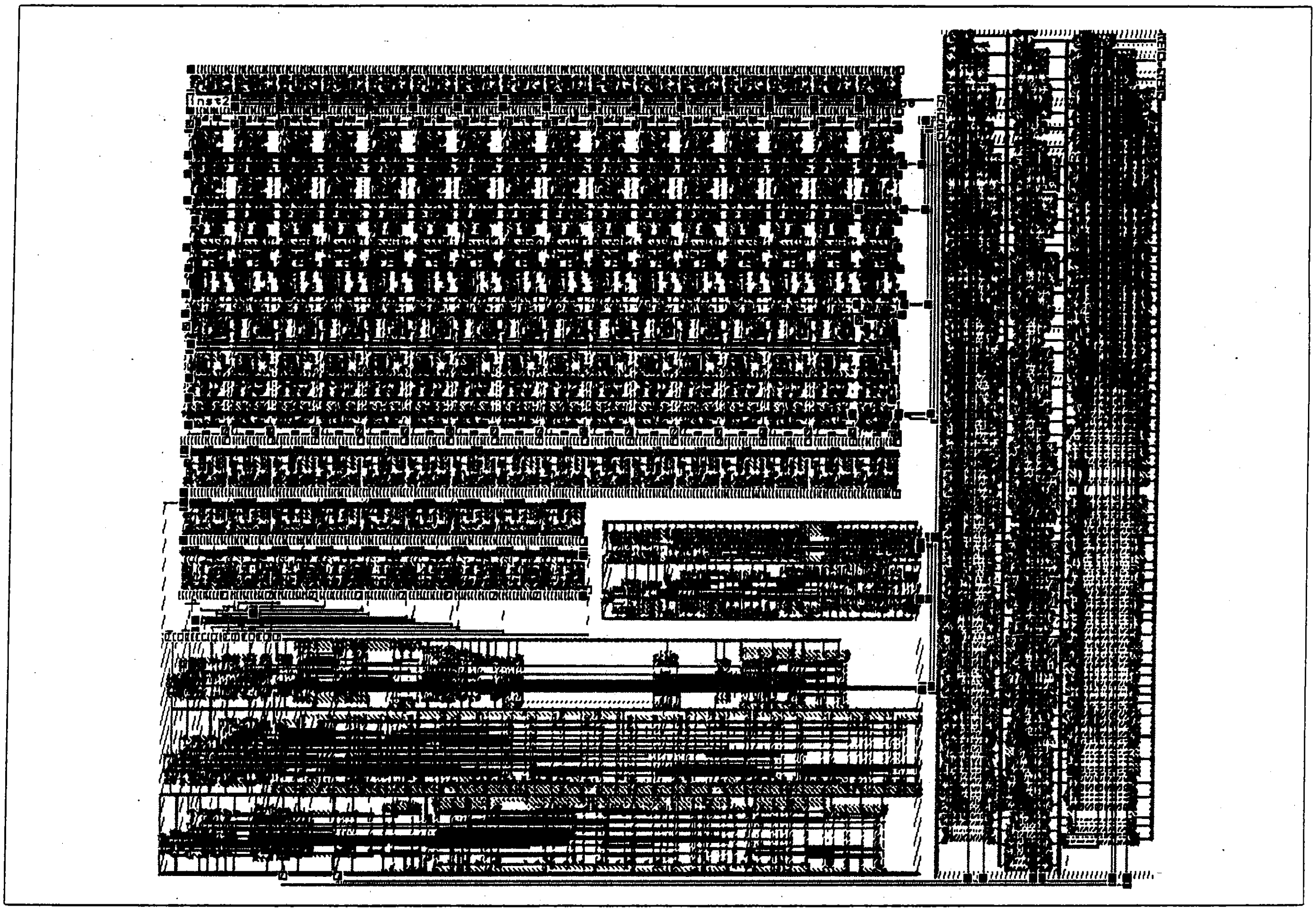

Figure 18. The layout of a DSP example

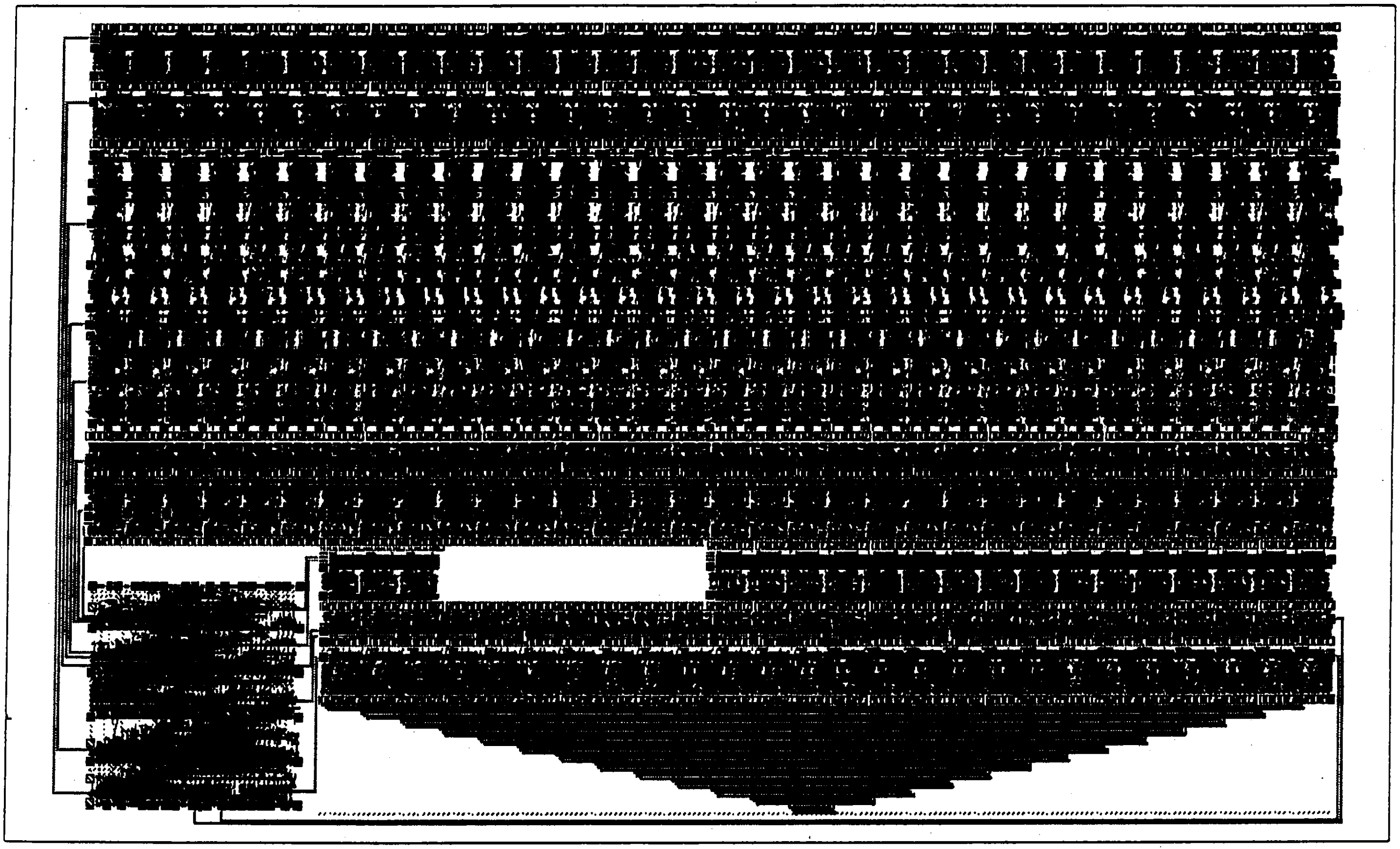

Figure 19. The layout of the MARK1 simple computer

example	our layout method	manual layout method	%
Controlled Counter	450,328 um^2	498,883 um^2	-10.8
DSP	7,056,000 um^2	7,896,042 um^2	-11.9
MARK1	11,220,000 um^2	12,701,040 um^2	-13.2

(a)

example	our layout method	manual layout method	%
Controlled Counter	495 um	765 um	-54.5
DSP	2,665 um	3,650 um	-36.9
MARK1	3,885 um	4,950 um	-27.4

(b)

Table 1. The comparisons of our partitioning and floorplanning with a manual partitioning and floorplanning: (a) total area and (b) critical-path wire lengths

9. References

[Arms89] Armstrong, J., Chip Level Modeling with VHDL, Prentice-Hall, 1989.

[Ayre90] Ayres, R. F. "Completely Automatic Completion of VLSI Designs," IEEE Trans.Computer-Aided Design Vol. 9, No. 2, pp.194-202. 1990.

[ChGa90] Chen, G. D. and Gajski, D., " An Intelligent Component Database System for Behavioral Synthesis," Proc. 27th DAC, 1990.

[DaEs89] Dai, W. M., Eschermann, B., Kuh, E. S., and Pedram, M., "Hierarchical Placement and Floorplanning in BEAR," IEEE Trans.Computer-Aided Design Vol. 8, No. 12, pp.1335-1349, 1989.

[DuKe85] Dunlop, A. E. and Kernighan, B. W., "A Procedure for Placement of Standard-Cell VLSI Circuits," IEEE Trans.Computer-Aided Design Vol. CAD-4, No. 1, pp.92-98, 1985.

[Hohn67] Hohnson, S. C., "Hierarchical Clustering Schemes," Psychometrika, pp.241-254, 1967.

[HsGr87] Hsu, D., Grate, L., Ng, C., Hartoog, M., and Bohm, D., "The ChipCompiler, An Automated Standard Cell/Macrocell Physical Design Tool," Proc. CICC, 1987.

[JaJe85] Jamier, R. and Jeraya, A., "APOLLON: A Datapath Compiler," Proc. ICCD, 1985.

[Joha79] Johannsen, D. L., "Bristle Blocks: A Silicon Compiler," Proc. 16th DAC, 1979.

[KeLi70] Kernighan, B. W. and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs," Bell System Technical Journal, Vol. 49, pp.291-308, (2), 1970.

[LaDi86] LaPotin, D. P. and Director, S. W., "Mason: A Global Floorplanning Approach for VLSI Design," IEEE Trans.Computer-Aided Design Vol. CAD-5, No. 4, pp.477-489, 1986.

[LiGa87] Lin, Y.L. and Gajski, D., "LES: A Layout Expert System," Proc. 24th DAC, 1987.

[LiGa88] Lis, J. S. and Gajski, D., "Synthesis from VHDL," Proc. ICCD, 1988.

[LuDe89] Luk, W. K. and Dean, A. A., "Multi-Stack Optimization for Data-Path Chip (Microprocessor) Layout," Proc. 26th Design Automation Conf., pp.110-115, 1989.

[RoWa87] Rowson, J., Walker, B., and Dholakia, S., "A Datapath Compiler for Standard Cells and Gate Arrays," Proc. CICC, 1987.

[SiBN82] Siewiorek, D. P., Bell, C. G., and Newell, A., Computer Structures: Principles and Examples, McGraw-Hill, 1982.

[ThKo87] Thonemann, H. G., Kolonko, M., Severloh, H., "VENUS-An Advanced VLSI Design Environment for Custom Integrated Circuits with Macros Cells, Standard Cells and Gate Arrays," Proc. CICC, 1987.

[UeKi85] Ueda, K., Kitazawa, H. and Adachi, T., "A Highly Automated Top-Down Layout Design System for Hierarchical Custom VLSIs," Proc. Custom IC Conf., pp.452-455, 1985.

[WuCG90] Wu, A. C. H., Chen, G. D. and Gajski, D., "Silicon Compilation from Register-Transfer Schematics," Proc. ISCAS, 1990.