SLAM: An Automated Structure to Layout Synthesis System

by

Allen C. H. Wu
Daniel D. Gajski

Technical Report 89-40

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

Abstract

SLAM is a structure to layout synthesis system. It incorporates parameterisable bit-sliced and glue-logic generators to produce high density layout. In this paper, we describe a sliced layout architecture and **SLAM** system. In addition, we present partitioning algorithms for generating the floorplan for such an architecture. The algorithms partition the netlist into component sets best suited for different layout styles such as bit-sliced or strip-oriented logic. Each group is partitioned further into clusters to achieve better area utilization. Several experiments demonstrate that highly dense layouts can be achieved by using these algorithms with the sliced layout architecture.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. Introduction

Surveys of VLSI products reveal that most of the 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 preferred layout strategy for such designs is the use of standard cells. Standard cell methodology does not take into account the regular nature (bit-slice property) of register-transfer components, since they are decomposed into basic gates, latches, and flip-flops before layout. Standard cells have two major disadvantages: (i) They require excessive routing, and (ii) They do not group the bits of register-transfer units into a bit-sliced layout. This lowers the performance of standard cell designs.

Other approaches[Joha79, JaJe85, PeWh86, VaCo86, RoWa87, ScWe87, ThKo87, HsGr87, LuDe89] have been reported that use datapaths with standard cells or macrocells. There are two common layout styles for datapaths: bit-sliced stacks and standard cells. Using a bit-sliced layout style, the datapath generator abuts the bits horizontally and register-transfer units vertically with no routing

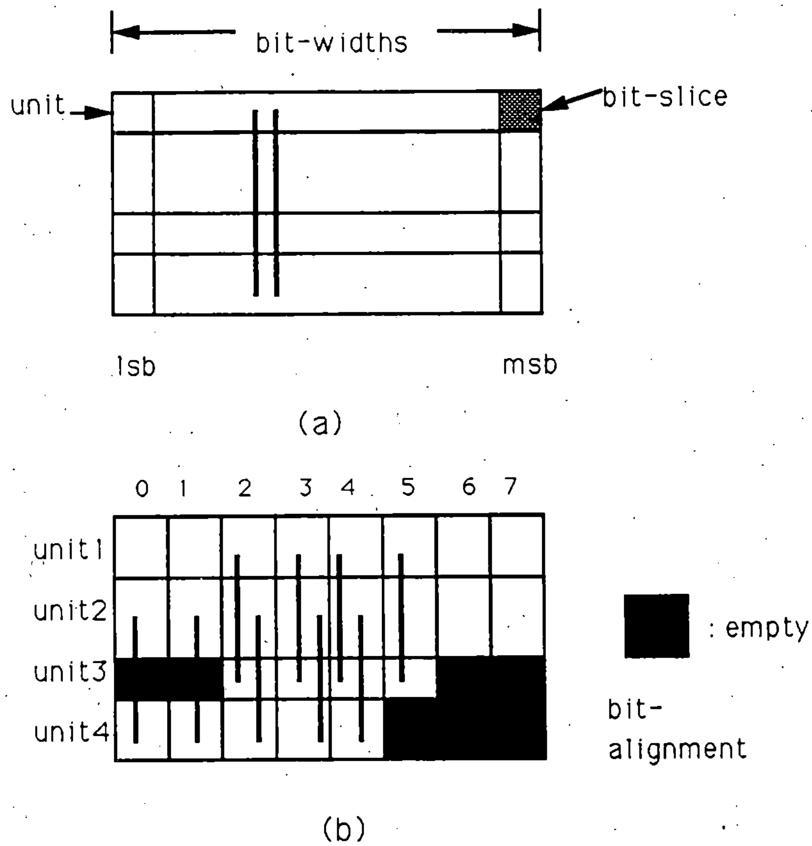


Figure 1. Bit-sliced layout architecture

channels between the units (Figure 1(a)). Data connections run over the bit-slices in the vertical direction. This approach can produce a high density layout if units are of the same bit-width and the interconnections between units are within the same bit-slice. This style, however, wastes area if units with different bit-widths are in the same datapath or if bits in different bit-slices must be connected. As shown in figure 1(b), the datapath has 4 units with bit-widths 8, 8, 5, and 4. The bit-slices(0-3) of unit 3 connect to the bit-slices(2-5) of unit 1 and the bit-slices(0-4) of unit 4 connect to the bit_slices(1-5) of unit 2. After placing and aligning the units 3 and 4, several bit-slices are left empty.

The second layout style uses standard cells or bit-sliced cells with routing channels (Figure 2(a)), producing a flexible datapath layout even with units of

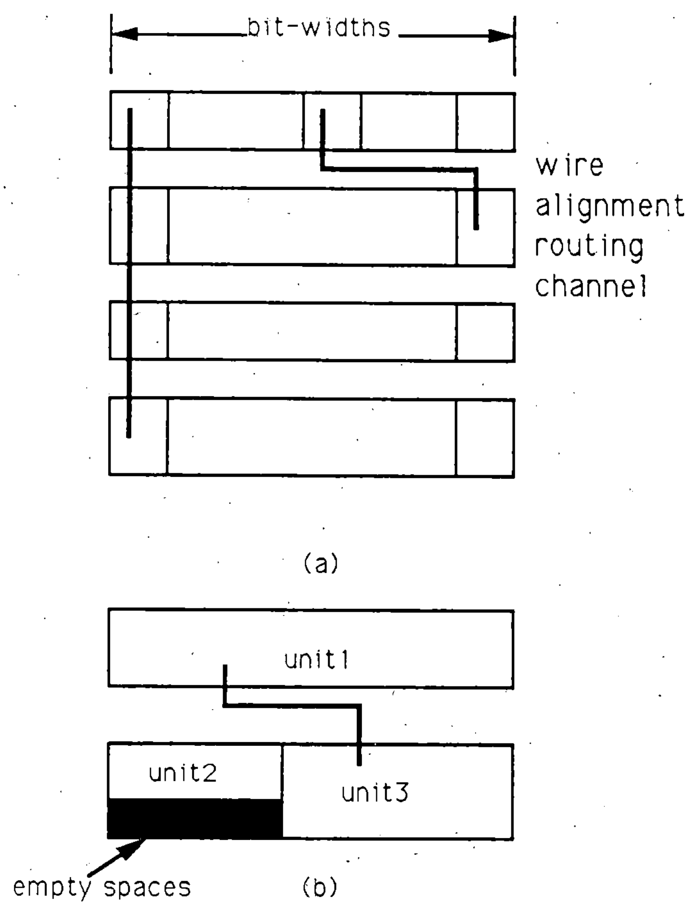


Figure 2. Standard cells and bit-sliced cells with routing channels layout architecture

different bit-widths. In this layout style, the bit-slices are abutted horizontally and bit-sliced units are placed vertically with routing channels between units. Several units with smaller bit-widths can be placed in the same row in order to reduce empty space (Figure 2(b)). This method still generates some wasted area because of mismatching of the height of adjacent units as shown in Figure 2(b). Furthermore, this approach needs to use a routing channel for wire connections between the units contributing to low area utilization. Recently, LASSIE[TrDi89] has used an approach that selects different layout styles (bit-slices or standard cells) for different designs.

In this paper, we first describe a "sliced" layout architecture which combines over-the-cell routing, switch box alignment, and layout folding to alleviate the problems that previous approaches encounter with the layout of register-transfer schematics. Furthermore, we describe an automated structure to layout synthesis system **SLAM** that uses parameterizable bit-sliced cell generator and a flexible custom layout system to produce high density layouts. **SLAM** tries to fully utilize high density bit-sliced cells and determines which layout style, glue-logic or bit-slices, is best suited for each component to achieve better area utilization. In addition, we describe partitioning algorithms used in the layout synthesis system, **SLAM**, that uses the new layout architecture. Two algorithms are used to obtain the final floorplan. The first algorithm partitions components into two groups for possible layout using a strip or bit-sliced layout architecture based on the connectivities, type of components, and possible overall area utilization. The other algorithm partitions those two groups further to achieve better area utilization. Bit-sliced components are partitioned by a folding algorithm into several folded "stacks" while striped components are partitioned into several striped modules with flexible aspect ratios to fit the overall floorplan.

In the remainder of this paper we describe the layout architecture (section 2), give the system overview (section 3), present the partitioning algorithms (section 4), describe the sliced-stack generation (section 5), and describe the floorplan and layout generation (section 6). Finally, we present the results of running several examples through the system (section 7) and conclusions (section 8).

2. Layout architecture

Sliced layout architecture combines over-the-cell routing, switch box alignment, and folding methods to produce high density layout. The sliced layout is a stack of register-transfer units. Each bit-slice has same width, but unit heights vary with the unit functionality. The stack grows horizontally when the bit-width increases, and grows vertically when the number of units increases. The sliced stack uses an over-the-cell routing strategy with data signals running vertically in 2nd metal over the bit-slices. Power, ground, carry, and control lines are routed horizontally in the 1st metal or poly between the bit-slices. When connections cross over several bit-slices are needed, a routing channel called "switch box" is inserted in the stack. This allows folding of the stack when several units of different bit-width are presented in the netlist.

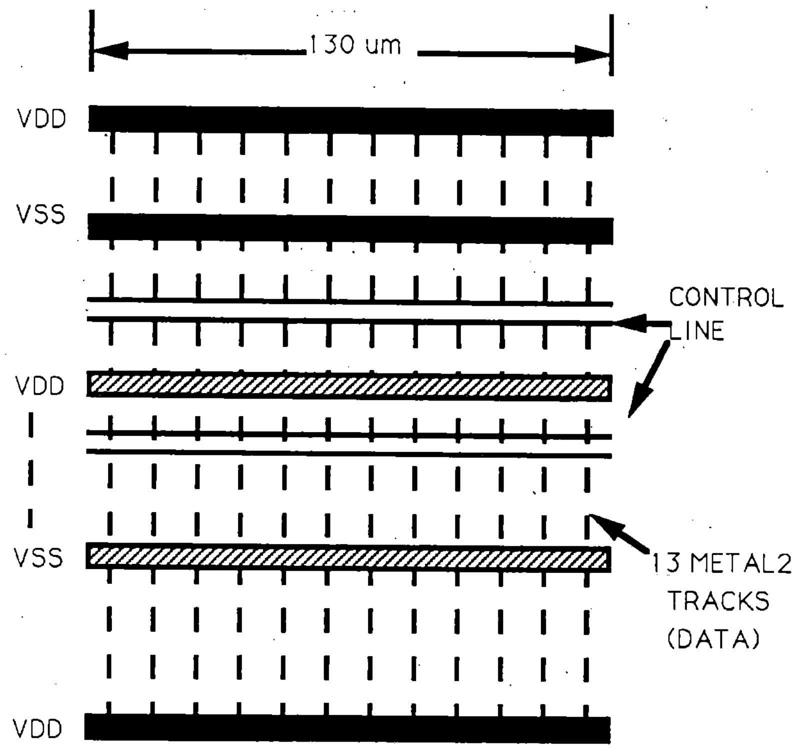


Figure 3. Sliced cell structure

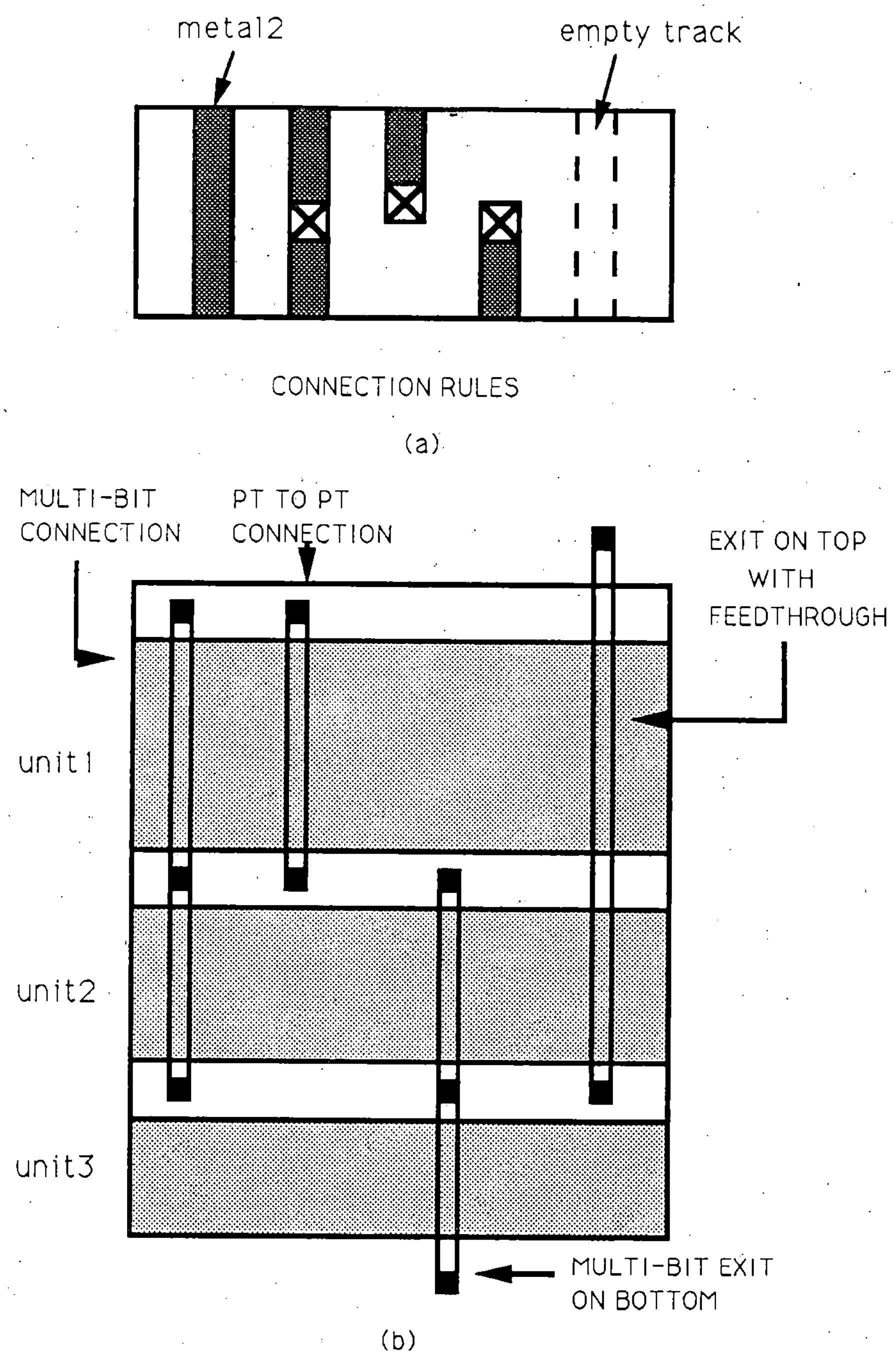


Figure 4. (a) Five connection modes, (b) Multi-bit connection, point to point connection, and input/output connection

The bit-sliced cell structure is shown in Figure 3. Each cell has a fixed horizontal pitch (130μm in our implementation), and a fixed number of metal2 routing tracks over the cell (13 in our implementation). There are five modes for connecting external wires: **(i)** Feedthrough, **(ii)** Feedthrough with connection, **(iii)** Up connection, **(iv)** Down connection, and **(v)** No connection(empty track) (Figure 4(a)). All the routing is accomplished by assigning tracks to the slices using these five rules. An example of multi-bit connection, point to point, and input/output connection is shown in Figure 4(b). Each unit, such as an ALU, multiplexer, register, adder, and register, is generated by a parameterizable generator. A layout for a 4-bit ALU is shown in Figure 5.

In the sliced layout, units are stacked vertically and aligned at the least significant bit. In order to connect different bit-slices of different units, a wire-alignment cell called a "switch box" is used for passing signals between bit-slices (Figure 6(a)). The switch box also rearranges the wire connections between two adjacent units as shown in Figure 6(b). Furthermore, the switch box can get external data from the right or the left (Figure 6(c)).

Units often have varying bit-widths. Because of these bit-width mismatches, there is a lot of empty space within the sliced stack's bounding box. A stack folding algorithm folds small units into this empty space. Units are fitted together as in a jigsaw puzzle. The folding is a two dimensional area filling

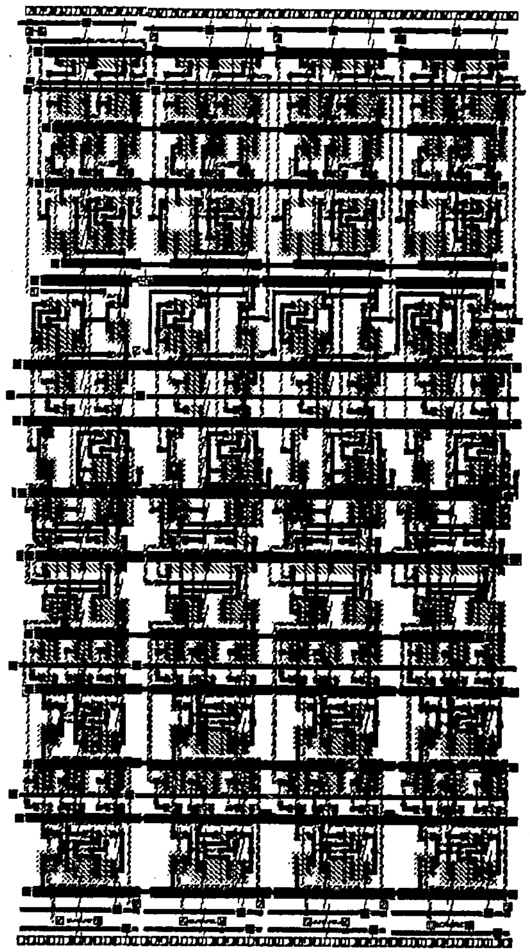


Figure 5. Layout of a 4-bit ALU

process that considers both the bit-widths and the heights of the units. Thus, it can alleviate the height mismatching problem that results from abutting two different units horizontally. The final stack structure is shown in Figure 7. The sliced stack is divided into two parts: folded and unfolded sections, which are connected by a switch box. The control signals exit on the left or the right. The

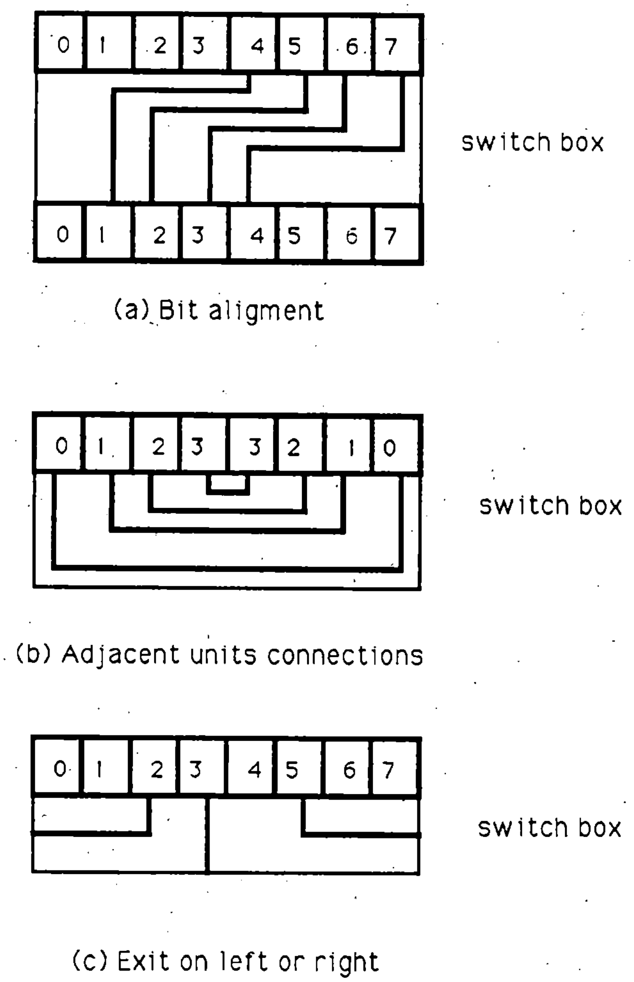


Figure 6. Three "switch box" modes for wire-alignment

input/output data signals exit on the top or bottom. After forming the sliced stacks, the rest of the components are placed around them under constraints such as input/output port positions, aspect ratio, and total area.

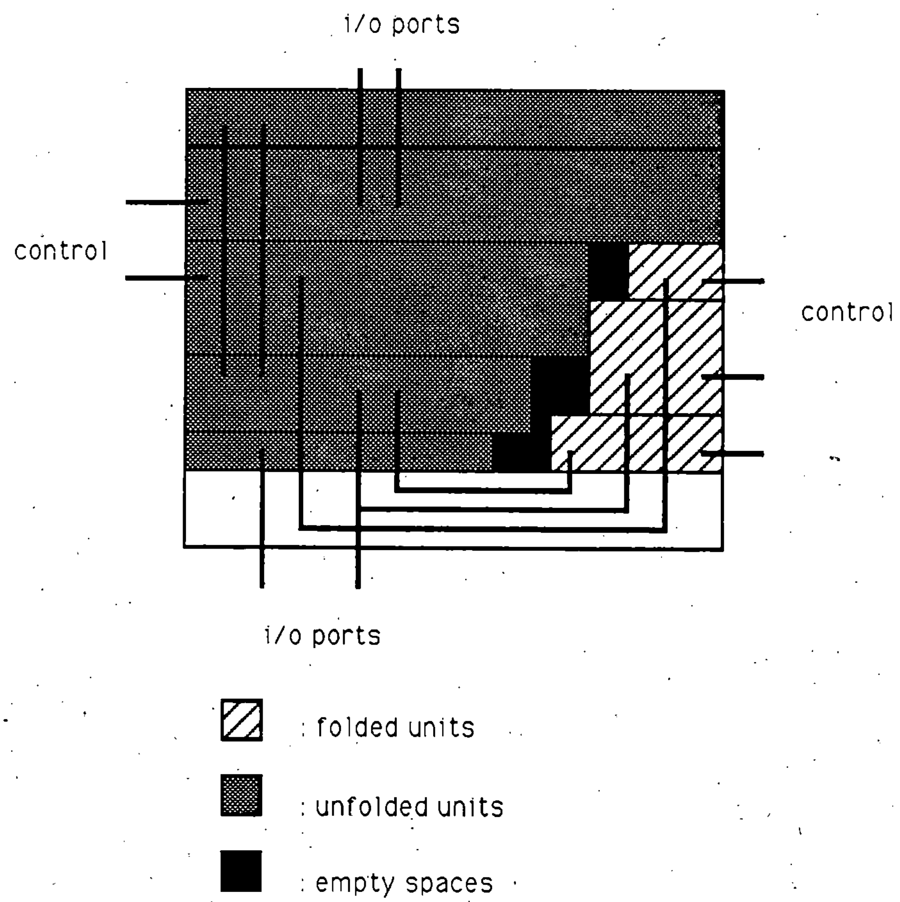


Figure 7. Folded sliced-stack structure

3. System overview

SLAM(Sliced Layout Methodology) is a register-transfer layout system that combines high density bit-sliced cells and flexible strip oriented modules to produce high density layouts. It uses the "sliced" stack and strip oriented layout architecture and combines partitioning and floorplanning techniques to transform a register-transfer netlist into a layout. The system block diagram is shown in Figure 8. The **SLAM** system consists of three main parts: (**i**) partitioning and component binding, (**ii**) floorplanning, and (**iii**) layout generation.

Input to **SLAM** is a register-transfer VHDL netlist which contains the description of a design[Lis89]. The connection binder first builds up a connected graph from the netlist. Then, the component partitioner separates component instances into sliceable or non-sliceable types based on the connectivities of components and their functionalities. The component binder obtains information for each component, such as type, area, and delay, by querying the Component Database[Chen89]. The component binder then assigns component types to the component instances based on the component partitioning algorithm. To achieve better area utilization, the stack partitioner folds the small units to fill empty space in the stack.

The unit placer permutes the bit-sliced units to minimize the routing track density, and the stack router assigns the routing tracks between the connected

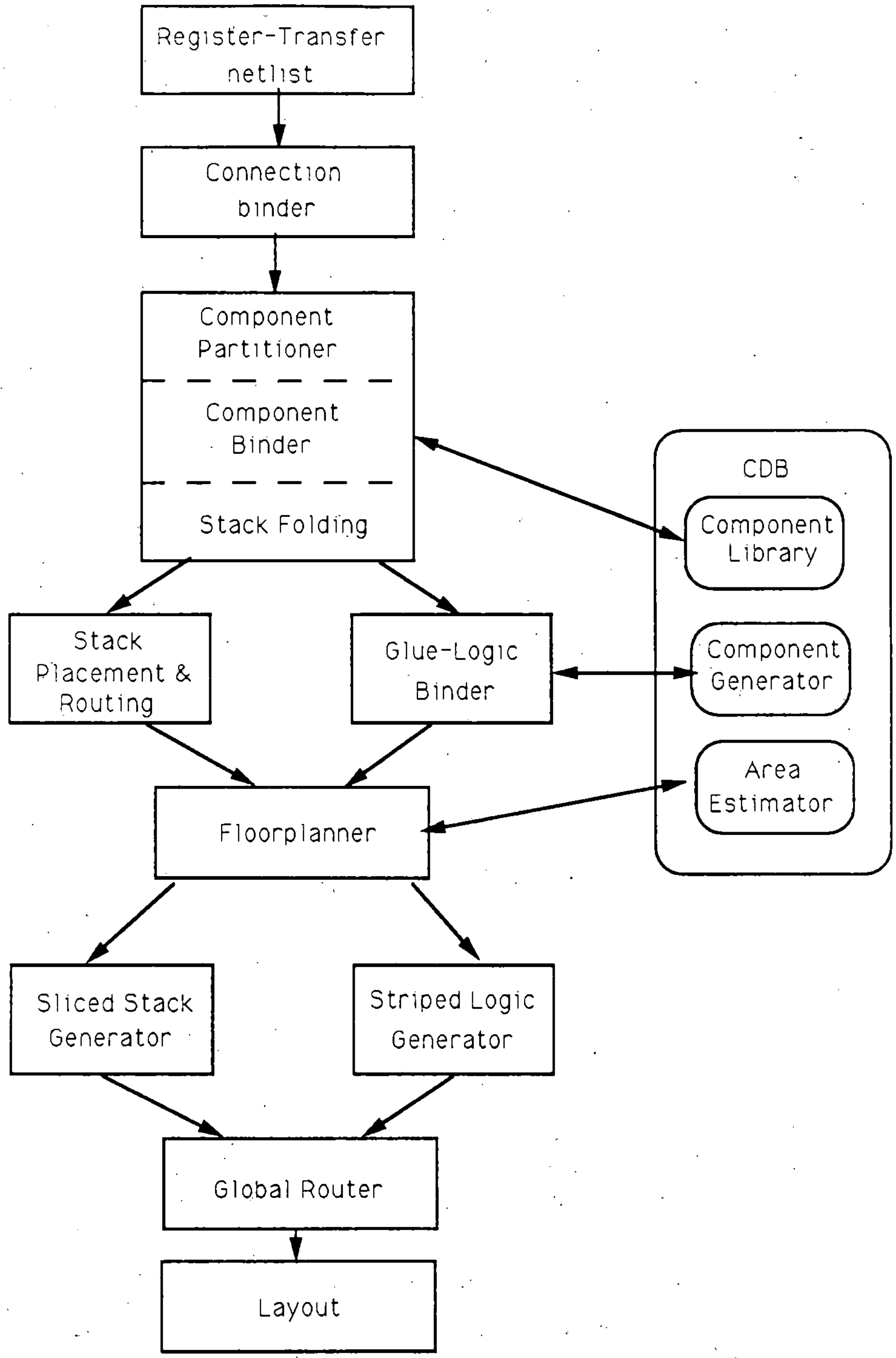


Figure 8. SLAM system block diagram

ports. After forming the sliced stack, the glue-logic component binder first estimates the loads for each wire giving to the sliced stack. The loads are calculated by summing the input capacitances of driven bit-sliced units and the routing wire capacitances. In the binding step, the binder forwards the glue-logic netlist, output loads, and delay constraints to the database, and retrieves a netlist of gates with pin information from the database. This netlist also contains the transistor sizes for each component. These transistor sizes are generated by a logic optimization phase[VaGa88] to meet a set of design's constraints. Finally, the binder maps the glue-logic unit into a gate level module by reconnecting the gate netlists of glue-logic components retrieved from the database.

The floorplanner uses a constructive method to place the glue-logic around the stack module. It also assigns the global routing channel and determines the aspect ratio of the glue-logic module. Furthermore, the flooplanner determines the ordering of input/output pins for the glue-logic that will minimize the wire crossing between stack and glue-logic modules.

In the final phase, the glue-logic module is generated by the striped layout generator[LiGa87], and the stack module is generated by generators using GDT[BuMa85]. A global router[SCS89] then finishes the detailed routing between modules to generate the final layout.

4. Partitioning

The primary purpose of partitioning is to define the layout style for each component in the design. By fully utilizing high density bit-sliced cells and using the best suited layout style for each component, better area utilization can be achieved. The overall objective is to minimize the total layout area and the interconnections between the sliced stack and glue-logic modules. Since routing among units use 1st and 2nd metal, the performance will stay the same or be slightly improved because of the smaller area.

The partitioning algorithm consists of three phases: **(i)** Component partitioning into bit-slices and glue-logic, **(ii)** Stack folding and partitioning, and **(iii)** Layout balancing. In phase one, the algorithm partitions the structural instances into sliceable and non-sliceable components based on components' characteristics and connectivities. In phase two, the algorithm partitions the sliceable components into several stacks using a folding method to reduce layout area. After folding, the layout balancing algorithm reassigns small components that do not fit in the stack module to the glue-logic to improve area utilization.

Throughout the paper, we will use **SS** for the bit-sliced units in the sliced stack, and **GL** for the glue-logic components.

4.1 Component partitioning

A weighted and labeled undirected graph **G** is formed by a set **U** of nodes, a set **V** of ports, and a set **E** of edges. There are **m** nodes in **U** where **m** is the number of components in the design, and there are **n** ports in each node where **n** is the number of ports in each component. The attribute type of a port **i**, ptype(**i**), indicates that port **i** is a control port or a data port. Let $e(\mathbf{i}_k, \mathbf{j}_l)$ be the edge between port **i** of $\mathbf{u}_k$ and port **j** of $\mathbf{u}_l$, where $\mathbf{u}_k, \mathbf{u}_l \in \mathbf{U}$ and $\mathbf{i}, \mathbf{j} \in \mathbf{V}$. The weight of an edge $\mathbf{e} \in \mathbf{E}$, $w[e(\mathbf{i}_k,\mathbf{j}_l)]$, is the number of wires between these two ports. The graph generated from the schematic in Figure 9(a) is shown in Figure 9(b). There are two components SS1 and SS2 with bit-widths 4 in the netlist. SS1 and SS2 form two nodes, U1 and U2, in the graph, with three ports each, one control port and two data ports. The edges correspond to connections between ports, while weights are equal to the multiplicity of connections. For example, $\mathbf{W}_3=4$ because there are 4 wires connect between port c and port **d**.

The component partitioning algorithm initially determines the component types of each node by querying the database. The component type can be **GL** only, such as single gates or DECODERs, SS only if is specified by user, or both **GL** and SS, such as a MUX or ALU. The algorithm assigns the component type to the node if it is a SS only or **GL** only component. If a component can be used as SS or **GL** and the bit-widths are larger than a user specified threshold(i.e. the

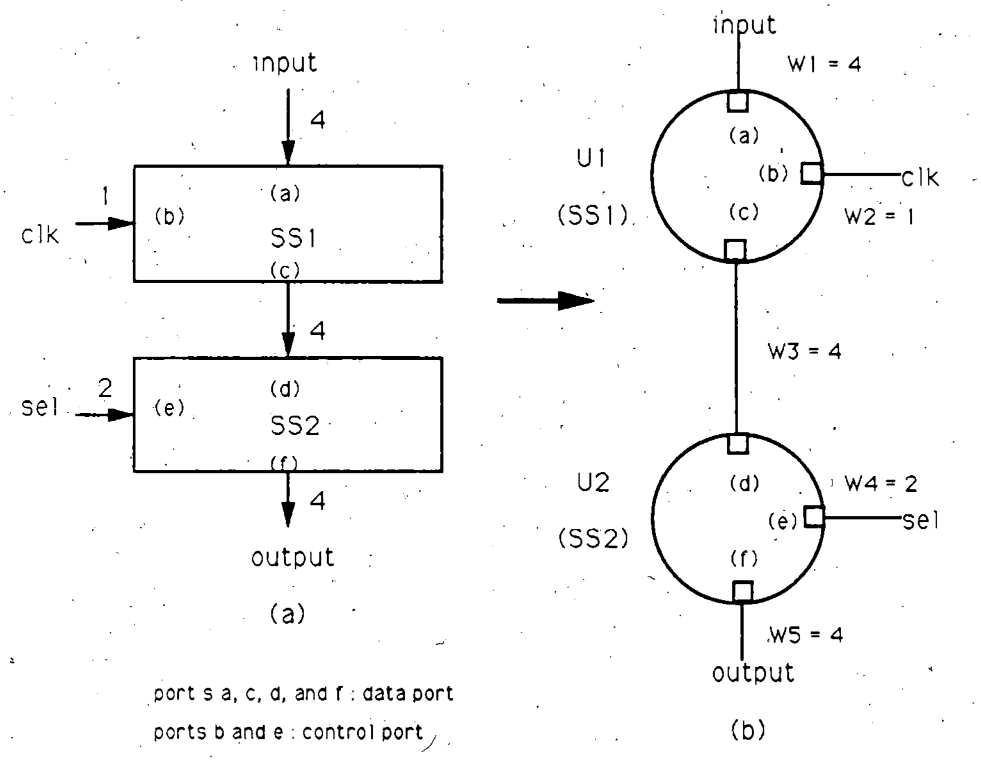


Figure 9. Graph representation of the structural netlist

minimum bit-width requirement of components that can be laid out by bit-sliced units), the SS is initially assigned to the component; otherwise, an undecided component type(UN) will be assigned to the component.

After the initial component type assignments, the algorithm evaluates and assigns the component types of the nodes based on a linking cost function. The linking cost functions are

$$\mathbf{W}_{control}(\mathbf{l}) = \sum \mathbf{w}(\mathbf{e}_{\mathrm{ln}}) \text{ if } \mathbf{u}_n \text{ is a } \mathbf{GL}$$

$$\mathbf{W}_{data}(\mathbf{l}) = \sum \mathbf{w}(\mathbf{e}_{\mathrm{ln}}) \text{ if } \mathbf{u}_n \text{ is a } \mathbf{SS}$$

where $\mathbf{W}_{control}(\mathbf{l})$ is the number of wires connected to $\mathbf{u}_l$ from other **GL** nodes, and $\mathbf{W}_{data}(\mathbf{l})$ is the number of wires connected to $\mathbf{u}_l$ from other **SS** nodes. If $\mathbf{W}_{data}(\mathbf{l}) > \mathbf{W}_{control}(\mathbf{l})$ then $\mathbf{u}_l$ is a **SS**; otherwise, $\mathbf{u}_l$ is a **GL**.

Assuming there is an edge $\mathbf{e}(\mathbf{i}_k, \mathbf{j}_l)$, the component type of $\mathbf{u}_l$ can be determined as follows:

(1) If the node $\mathbf{u}_l$ is a component of undecided type, the algorithm simply assigns a **SS** component type to the node $\mathbf{u}_l$ if $\mathbf{ptype}(\mathbf{i}_k)$ is a data port; otherwise, a **GL** component type will be assigned to $\mathbf{u}_l$.

(2) If the node $\mathbf{u}_l$ has an initial **SS** or **GL** component type, there are two possible cases: **(i)** If $\mathbf{ptype}(\mathbf{i}_k)$ is a data port and $\mathbf{u}_l$ is a **SS**, or $\mathbf{ptype}(\mathbf{i}_k)$ is a control port and $\mathbf{u}_l$ is a **GL**, the $\mathbf{u}_l$'s component type is unchanged. **(ii)** If $\mathbf{ptype}(\mathbf{i}_k)$ is a data port and $\mathbf{u}_l$ is a **GL**, or $\mathbf{ptype}(\mathbf{i}_k)$ is a control port and $\mathbf{u}_l$ is a **SS**, the linking cost function, $\mathbf{W}_{control}(\mathbf{l})$ and $\mathbf{W}_{data}(\mathbf{l})$, are used to determine the $\mathbf{u}_l$'s component type.

ALGORITHM 1 Component Partitioning

```
PROCEDURE Component_partitioning()
begin
/**Let Ct(u) be the component type of u.**/
  G = build_graph();
  /**initial component type assignments by querying database**/
  for all u ∈ U
    Ct(u) = init_type_assignment(u);
  /**Let ptype(i_k) ∈ {data or control} where i ∈ V and k ∈ u. Let Ψ = U and assigns ψ ∈
  Ψ into four groups Ct(ψ) ∈ {IO, SS, GL, or UN} where Ψ is in the sorting order according
  to the bit-widths and group orders**/
  while Ψ ≠ φ
  begin
    ψ_k = head of Ψ;
      for i ∈ V and i ⊂ ψ_k
      begin
        for e(i_k, j_l) ∈ E
        begin
          if (Ct(ψ_l) == UN and ptype(i_k) == data) then
            Ct(ψ_l) = SS;
          else if (Ct(ψ_l) == UN and ptype(i_k) == control) then
            Ct(ψ_l) = GL;
          else
          begin
            if (((ptype(i_k)==data and Ct(ψ_l)==GL)
              or (ptype(i_k)==control and Ct(ψ_l)==SS))
              and Ct(ψ_l) ≠ {SSonly, GLonly, or IO}) then
              begin
                calculate W_control(l) and W_data(l);
                if W_control(l) > W_data(l) then
                  Ct(ψ_l) = GL;
                else
                  Ct(ψ_l) = SS;
              end;
          end;
          Ψ = Ψ - ψ;
        end;
      end;
    end;
  end;
```

4.2 Stack partitioning by folding

Using the sliced architecture, the bit-sliced units need to be aligned so that signals can pass through all of the units. Often units have different bit-widths. Because of this bit-width mismatch, there is some empty space within the stack bounding box. The folding algorithm tries to fold small units to fill these empty space. The stack will also be partitioned into multiple stacks if a smaller layout area can be achieved. The main objective of folding is to minimize the total layout area.

Definition 1

The bounding box of the 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 $\mathbf{u}_i$. $\mathbf{h}_i$ and $\mathbf{w}_i$ are the height and width of unit $\mathbf{u}_i$.

Definition 2

The sliced-stack area, $\mathbf{A}_{ss}$,is determined by the minimum bounding box enclosing all units, where $\mathbf{H}_{ss}$ and $\mathbf{W}_{ss}$ are the height and the width of this bounding box.

Definition 3

Let $\mathbf{fold}_{ss}$ be the stack containing all of the folded units, and $\mathbf{H}_{ss\_fold}$ and $\mathbf{W}_{ss\_fold}$ be the height and width of $\mathbf{fold}_{ss}$'s bounding box. Let $\mathbf{unfold}_{ss}$ be the datapath containing all of the unfolded units, and $\mathbf{H}_{ss\_unfold}$ and $\mathbf{W}_{ss\_unfold}$ be the height and width of **unfold**$_{ss}$'s bounding box. Let $\mathbf{H}_{cutline}$ be the cutline that separates the unfolded units with maximum bit-widths and the rest of unfolded units.

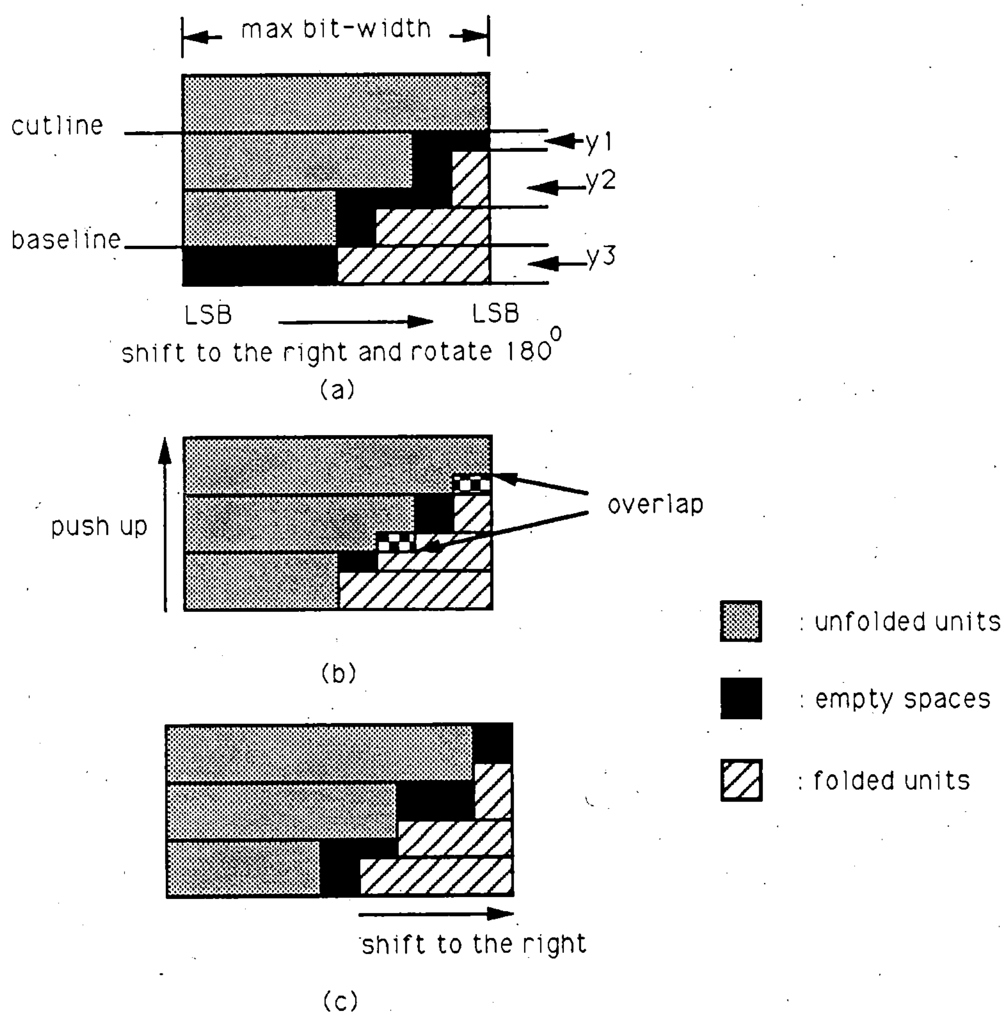


Figure 10. Stack folding process

The folding algorithm includes three steps: unit folding, overlap checking, and area cost function evaluation. There are two main constraints for the stack folding: (i) the units must be aligned with the least significant bit and (ii) the units must not overlap. The algorithm first sorts the units based on the units' bit-widths. One unit is folded at a time. The folding process has 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 10(a)) and (ii) push all of the folded units up based on a step function, $\mathbf{y}_{step}$, until reaching the base-line (Figure 10(b)). The step function $\mathbf{y}_{step}$ is defined as follows:

$\mathbf{y}_{step} = \text{Min}\{\ \mathbf{y}_1,\ \mathbf{y}_2, \text{ and } \mathbf{y}_3\}$ **and** $\mathbf{y}_{step} > 0$

where
$\mathbf{y}_1$ is the height between the $\mathbf{H}'_{cutline}$ and the top of $\mathbf{fold}_{ss}$.

$\mathbf{y}_2$ is the height of first folded unit below the $\mathbf{H}_{cutline}$.

$\mathbf{y}_3$ is the height between the base-line and the bottom of $\mathbf{fold}_{ss}$.

After unit folding, an overlap checking procedure is implemented to check whether the units in the folded part and the unfolded part overlap. 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 $\mathbf{u}_i$. The overlapping conditions are

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

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

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

If an overlap occurred, the algorithm will shift the folded units to the right by $\mathbf{X}_{shift}$ to avoid the overlap (Figure 10(c)). $\mathbf{X}_{shift}$ is defined as follows:

$$\mathbf{X}_{shift} = \max\{\mathbf{x}_{lr.i} - \mathbf{x}_{ul.j}\}$$

iff $\mathbf{x}_{ul.j} < \mathbf{x}_{lr.i}$ and $\mathbf{y}_{ul.j} < \mathbf{y}_{lr.i}$

where
$\mathbf{u}_i \in \mathbf{unfold}_{ss}$ and $\mathbf{u}_j \in \mathbf{fold}_{ss}$

After folding a unit $\mathbf{u}_i$, we have

$$\mathbf{W}_{ss} = \mathbf{W}_{ss} + \mathbf{X}_{shift} \textbf{ if overlap}$$

$$\mathbf{H}_{ss} = \max\{\mathbf{H}_{ss\_fold}, \mathbf{H}_{ss\_unfold}\}$$

where
$$\mathbf{H}_{ss\_fold} = \mathbf{H}_{ss\_fold} + \mathbf{h}_i$$

$$\mathbf{H}_{ss\_unfold} = \mathbf{H}_{ss\_unfold} - \mathbf{h}_i$$

The algorithm then evaluates a cost function to select the best stack partition. The area cost function $\mathbf{A}_{ss}$ evaluation has two conditions as follows:

(1) If $\mathbf{H}_{cutline} > \mathbf{H}_{ss\_fold}$, the cost function of $\mathbf{A}_{ss}$ is

$$\mathbf{A}_{ss} = \mathbf{W}_{ss} * \mathbf{H}_{ss\_unfold} + \mathbf{A}_{routing\_fold\_unfold}$$

where

$\mathbf{A}_{routing\_fold\_unfold}$ is the routing channel area for connecting $\mathbf{unfold}_{ss}$ and $\mathbf{fold}_{ss}$

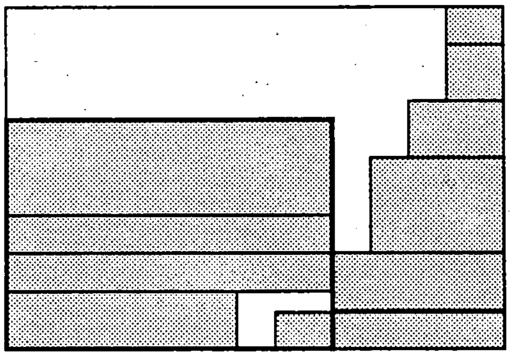
(2) If $\mathbf{H}_{ss\_fold} > \mathbf{H}_{cutline}$, some units in $\mathbf{fold}_{ss}$ overshoot the cutline of $\mathbf{unfold}_{ss}$ and overlap with some units in $\mathbf{unfold}_{ss}$. There are two area cost functions $\mathbf{A}_{ss}$ and $\mathbf{A}_{new}$. They are defined as follows:

(i) The $\mathbf{A}_{ss}$ cost function is the minimum bounding box enclosing all of the units and the routing area for connecting the unfolded units and the folded units Figure 11(a).

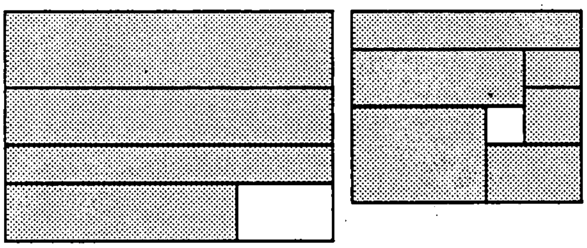
(ii) Using the second cost function, $\mathbf{A}_{new}$, the algorithm moves the unfitted units(overshoot units) from the first stack module to form a new stack module Figure 11(b). The cost function is

$$\mathbf{A}_{new} = \mathbf{A}_{ss\_old} + \mathbf{A}_{ss\_new} + \mathbf{A}_{routing\_old\_new}$$

$\mathbf{A}_{ss\_old}$ is the first stack module area without the unfitted units, and $\mathbf{A}_{ss\_new}$ is the new stack module area that contains the units that do not fit in the first stack module. $\mathbf{A}_{routing\_old\_new}$ is the routing area between two stack modules. If $\mathbf{A}_{new} < \mathbf{A}_{ss}$ then the algorithm moves the unfitted units to the new stack group for further partitioning. The stack folding algorithm is shown as follows:



(a)



(b)

: bit-sliced unit

Figure 11. Two area cost functions for area evaluation

ALGORITHM 2 Stack Partitioning

```
Let D be a set of SS components.
PROCEDURE Stack_partitioning(D)
begin
  D_unfold = sorts d ∈ D according to bit_widths in descending order;
  D_fold = φ ;
  D_new = φ ;
  A_ss_minimum = H_ss_unfold * W_ss_unfold;
  D_min = D_unfold;
  d = head of D_unfold;
  {fold d}
  while (bit_width(d) < max_bit_width)
  begin
    D_unfold = D_unfold - d;
    D_fold = D_fold + d;
    if (D_unfold overlaps D_fold) then
      shift D_fold;
    if (H_cutline > H_ss_fold) then
      calculate A_ss;
    else
    begin
      calculate A_ss and A_new;
      if (A_ss > A_new) then
      begin
        D_new = D_new + d_unfit;
        D_fold = D_fold - d_unfit;
        A_ss = A_new;
      end;
    end;
    if (A_ss_minimum > A_ss) then
    begin
      A_ss_minimum = A_ss;
      D_min = D_unfold + D_fold;
    end;
    d = head of D_unfold;
  end;
  if (D_new ≠ φ ) then
    Stack_partitioning(D_new);
end;
```

The stack partitioning algorithm is executed recursively until no more stacks can be formed. The layout balancing algorithm first moves the small units that do not fit in the stack to the glue-logic. If there is more than one stack module, the algorithm estimates two area costs: (i) The layout of the small stack module using striped logic and (ii) The layout of the small stack module using sliced units. If the total area of (i) is less than that of (ii), the algorithm moves all of the components in the small stack to the glue-logic.

5. Sliced-stack generation

Stack generation maps the bit-sliced components into a sliced stack module. Stack generation includes three steps: (i) placement, (ii) folding, and (iii) routing.

The connection binding step maps the register-transfer structural netlist into connected graph as described in section 4.1. In our implementation, there are three wiring modules, SELECTOR, CONCAT, and PORT[Lis89], that offer the wiring information among units. The binder will delete the wiring modules after specifying all the wire connections. The component partitioning step partitions the modules into bit-slice and glue-logic as described in section 4.1.

The main goal of placement is to determine the optimal placement of units that minimizes the total number of routing tracks and wire length. The placement algorithm includes three steps: (i) initial placement, (ii) routing track alignment, and (iii) routing track minimization. The algorithm first sorts the units based on bit-widths, and places units in a descending order aligned with the least significant bit. The algorithm then determines the misaligned wire connections between units by traversing the connected graph. If there are wire connections between the different bit-slices in the units, a switch box will be inserted to align the routing tracks. An example of wire connection misalignment and switch box insertion is shown in Figure 12. There are three **SS** units with bit-widths 8,5, and 4. Because the units are abutted and aligned with the least significant bit (Figure 12(b)), the wire connections between **Reg1** (bits 4-7) and **Reg2** (bits 0-3) are not routeable. Therefore, a routing box is inserted to connect **Reg1** and **Reg3** (Figure 12(c)).

The routing track minimization step permutes the order of the units of the same bit-widths to minimize the track density. The complexity of the exhaustive ordering search algorithm is **O(n!)** where **n** is the maximum number of units in the same bit_width group. Therefore, the exhaustive search method is suitable for small unit sizes, but is impractical to implement for large problems. The routing track minimization algorithm implements a heuristic by combining min-cut and exhaustive search methods. Because units are placed in a sorted order, the algorithm only needs to permute the units in the same bit-width group. The algorithm first partitions the units into groups based on bit-width. If the number of units in a group is less than a threshold, then it permutes the units exhaustively to find the minimal track density. Otherwise, a min-cut

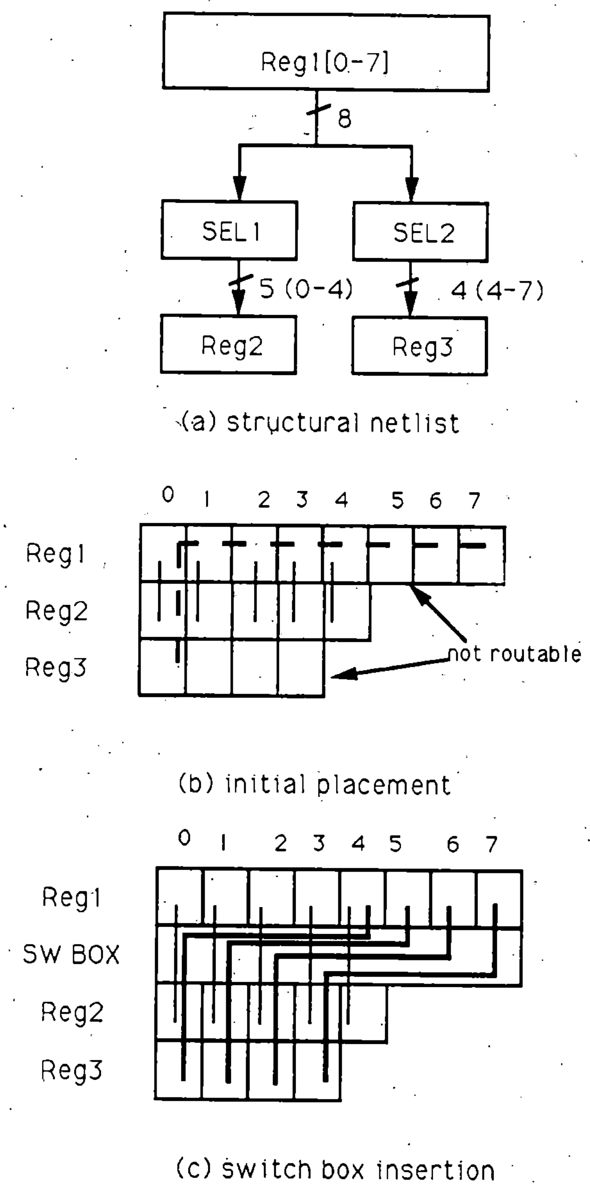


Figure 12. Switch box insertion for wire-alignment

algorithm[KeLi70] is implemented recursively until the sub-group sizes are less than a threshold, at which time it applies exhaustive permutation on each sub-group.

After the placement step, the stack folding algorithm as described previously is applied to reduce the layout area. Finally, the routing tracks are assigned to the units using a left-edge algorithm[HaSt71].

6 Floorplanning and layout generation

The glue-logic is placed around the stacks after forming the stack modules. The floorplanner determines the aspect ratio and location of the **GL** module to achieve minimum total layout area or aspect ratio. To obtain minimal layout area, the system examines different floorplan styles and selects the one with the minimum area for the final floorplan. For instance, consider a floorplan style that incorporates one stack and one glue-logic unit. There are two sub-styles: (i) The glue-logic module can be placed on the left of the stack module or (ii) The glue-logic module can be placed on the bottom of the stack module (Figure 13). The final area is calculated as follows:

$$\mathbf{A}_{total} = \mathbf{A}_{gl} + \mathbf{A}_{ss} + \mathbf{A}_{routing}$$

$$\mathbf{A}_{routing} = \mathbf{A}_{ss-gl} + \mathbf{A}_{fold-control} + \mathbf{A}_{unfold-control}$$

The total area is the summation of the stack area, the glue-logic area, and the total routing area. The routing area consists of three parts: (i) The routing channel between the glue-logic module and the control ports of the folded stack,

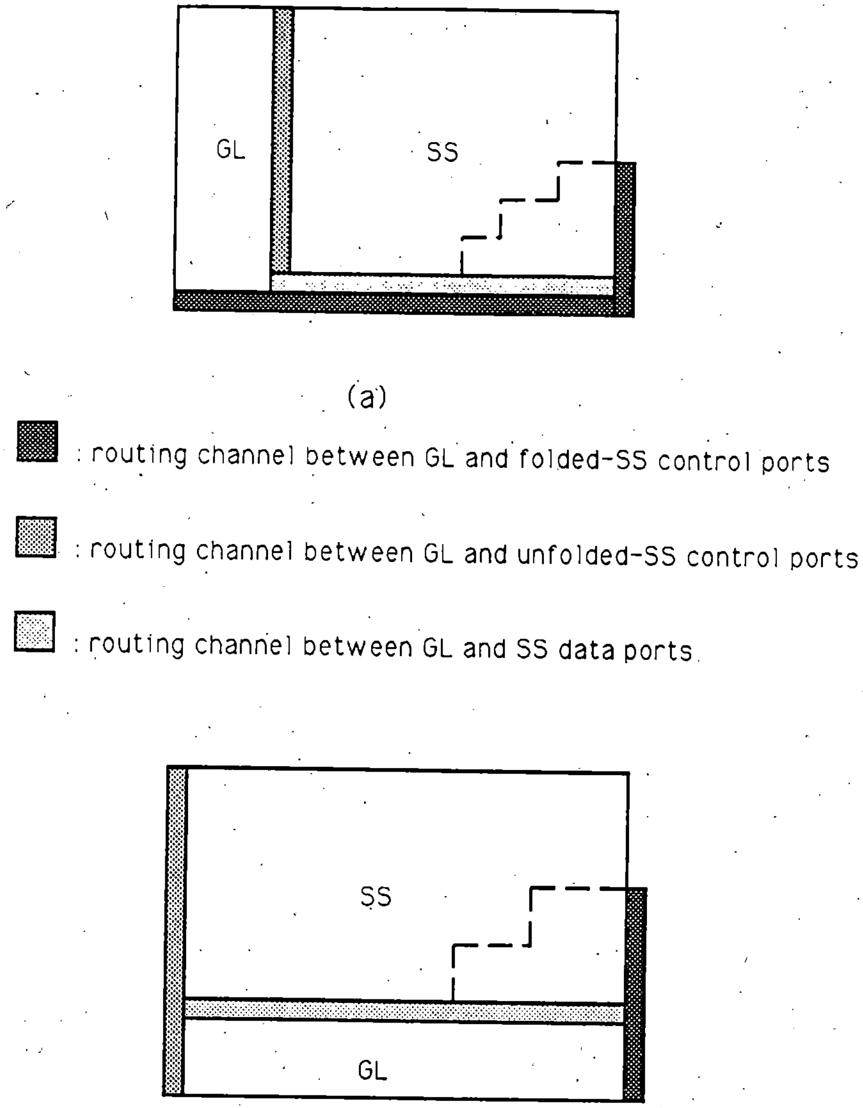


Figure 13. Two floorplan styles for one sliced stack and one glue-logic unit

(ii) The routing channel between the glue-logic module and the control ports of the unfolded stack, and (iii) The routing channel between the glue-logic module and the data ports of the stack. By querying the database, the aspect ratio, height, and width of the glue-logic module can be determined. The placer then calculates the total area, and the style with minimal total area is selected as the final floorplan.

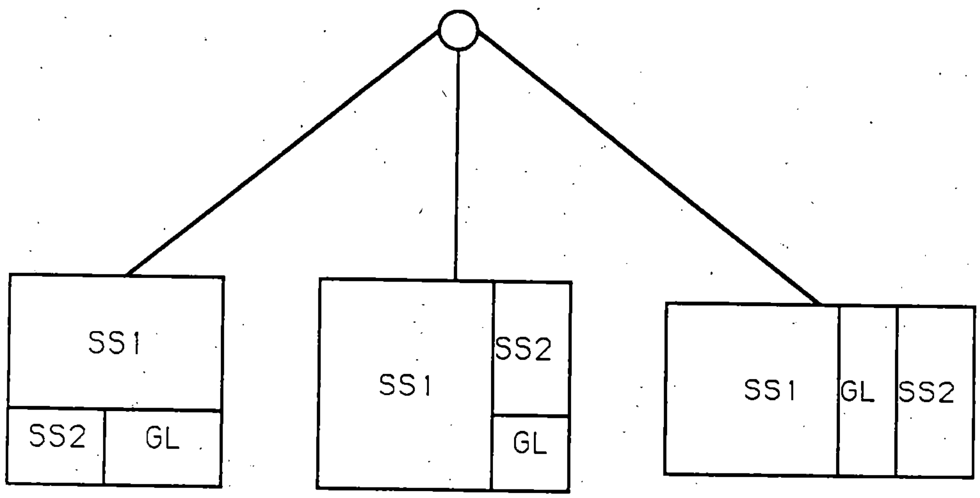


Figure 14. Three floorplan styles for two sliced stacks and one glue-logic unit

The second floorplan style incorporates two stacks and one glue-logic module. There are eight possible configurations of the layout that can be divided into three styles(figure 14). The placer treats the glue-logic module as a soft block which is flexible for changing the aspect ratio. The placer first determines the minimum bounding box that contains both stacks. After placing the stack modules, the placer determines the aspect ratio of the soft glue-logic block by querying the database. The style with minimal total area is selected as the final floorplan.

After selecting the floorplan style, the placer determines the ordering of input/output ports for the glut-logic module. Since the input/output ports of the stack are in the fixed positions, the placer simply assigns the port positions of glue-logic module corresponding to the port positions of the stacks based on connection configurations. There are two basic connection configurations. If the glue-logic module and the stack have adjacent connection boundary then the placer assigns the same ordering of ports to the glue-logic module as stack's(figure 15(a)). Otherwise, the placer assigns the reverse ordering of ports to the glue-logic module(figure 15(b-c)). Finally, the layout is generated using the striped layout generator and the bit-sliced generators with a global router.

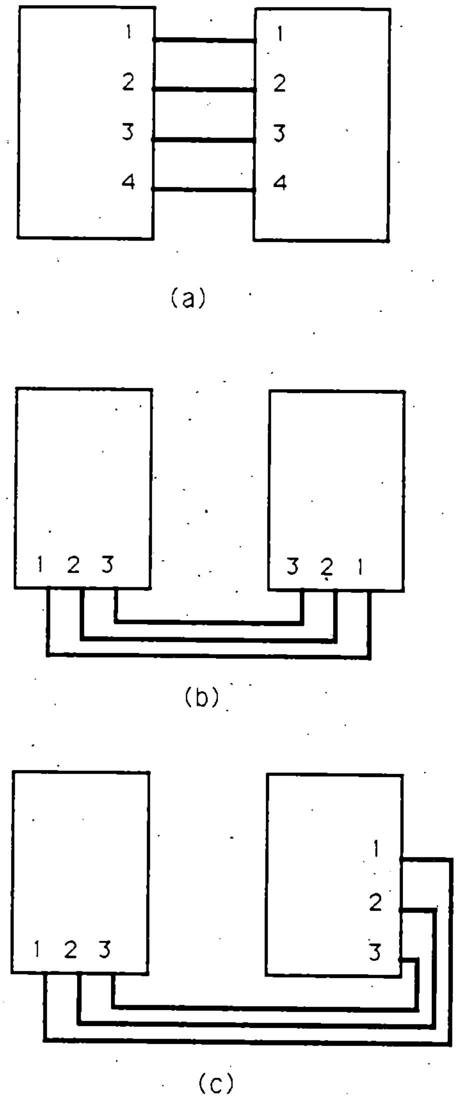


Figure 15. Wire ordering for sliced stack and glue-logic unit

7. Results

The **SLAM** system is implemented in the C programming language and is currently running on SUN 3/SUN 4 workstations under the UNIX operating system. A number of examples have been tested. The register-transfer structural netlists were generated from a VHDL synthesis system **VSS**[LiGa89] or mapped from register-transfer schematics. 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. Three different layouts were generated using (i) **SLAM** without partitioning, (ii) **SLAM** with partitioning, and (iii) standard cells. The results in Table 1 show that the layout generated by **SLAM** with partitioning is 12% smaller than that without partitioning, and 20% smaller than that of standard cells. The final layout that is generated by **SLAM** with partitioning is shown in Figure 16. Example 2 consists of seven units with different bit-widths. Figures 17(a) and 17(b) show the layouts generated using the **SLAM** system without and with implementing stack folding. Figure 17(c) shows the layout generated using the same units with a global router. To generate the layout of Figure 17(c), we used GDT interactive floorplanner to place the units. The results in Table 2 show that the total area using stack folding is 40% less than that of the other two approaches. Example 3 is the Mark1 simple computer[SiGo82] which consists of 20 components with bit-widths 32, 16, 13, 3, and 1. The partitioner partitions the design into two sliced stack modules and one glue-logic module, and the final layout is shown in Figure 18. The results in Table 3 show that the layout generated by **SLAM** with partitioning is 20% smaller than that without partitioning.

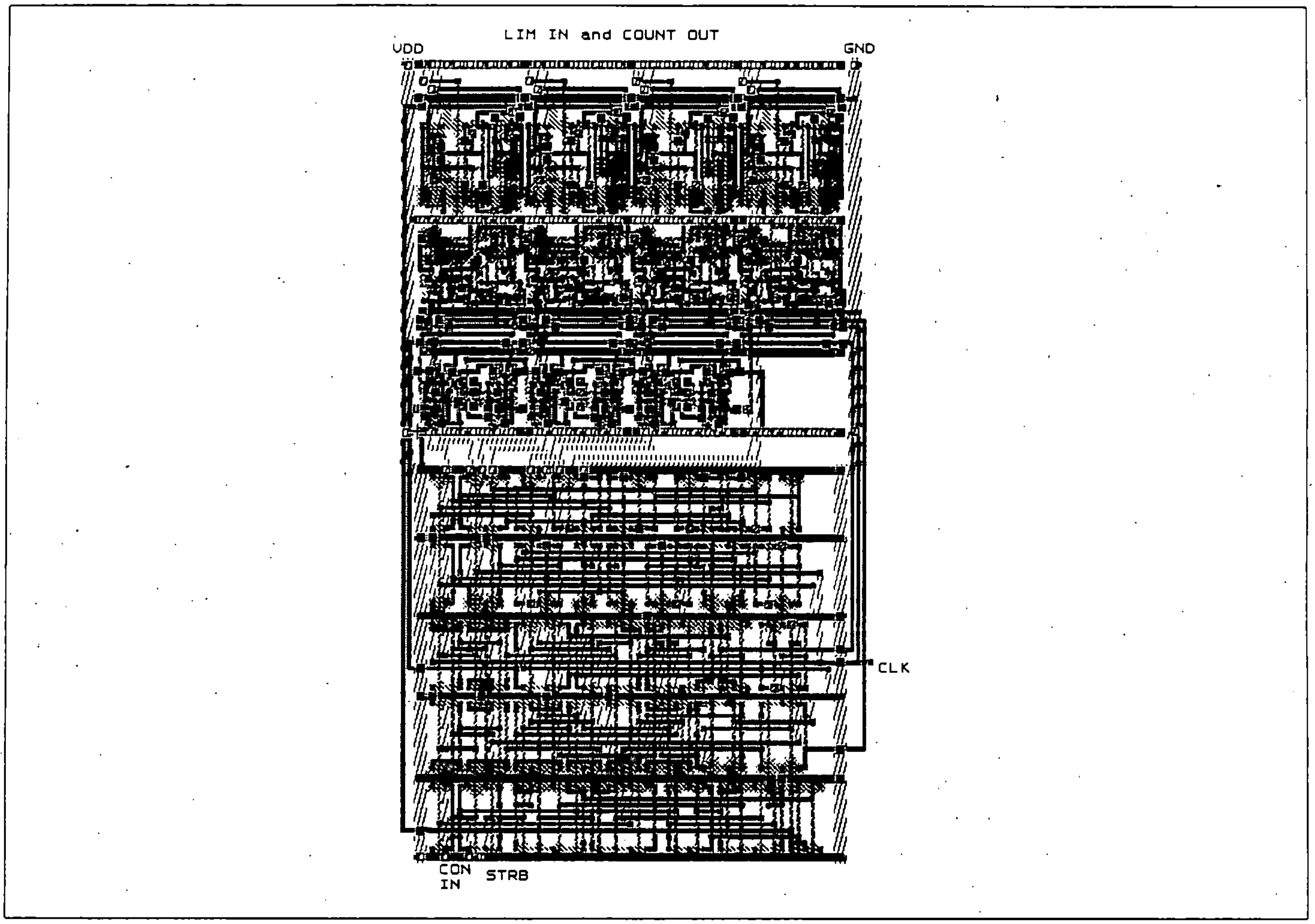


Figure 16. Layout of a controlled counter

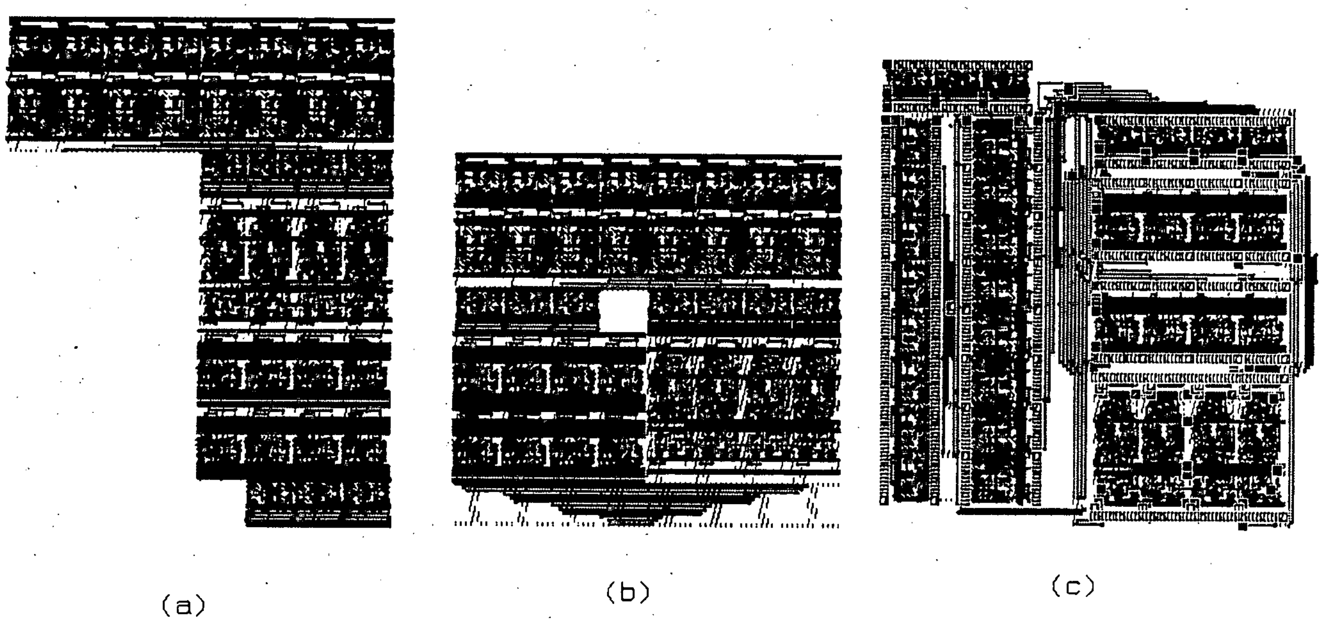


Figure 17. (a) SLAM without stack folding, (b) SLAM with stack folding, and (c) macrocells placement/routing

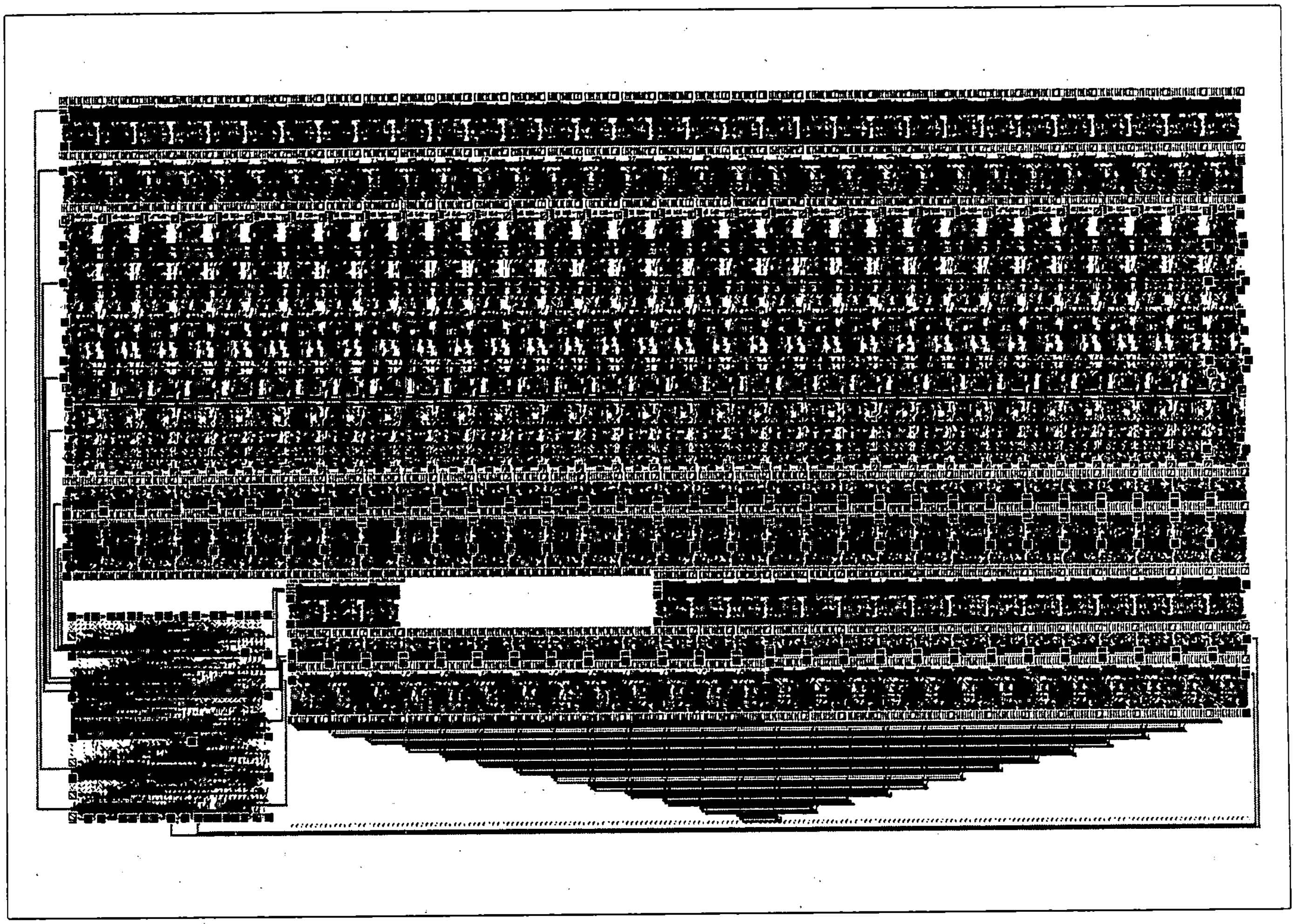


Figure 18. Layout of MARK1 simple computer

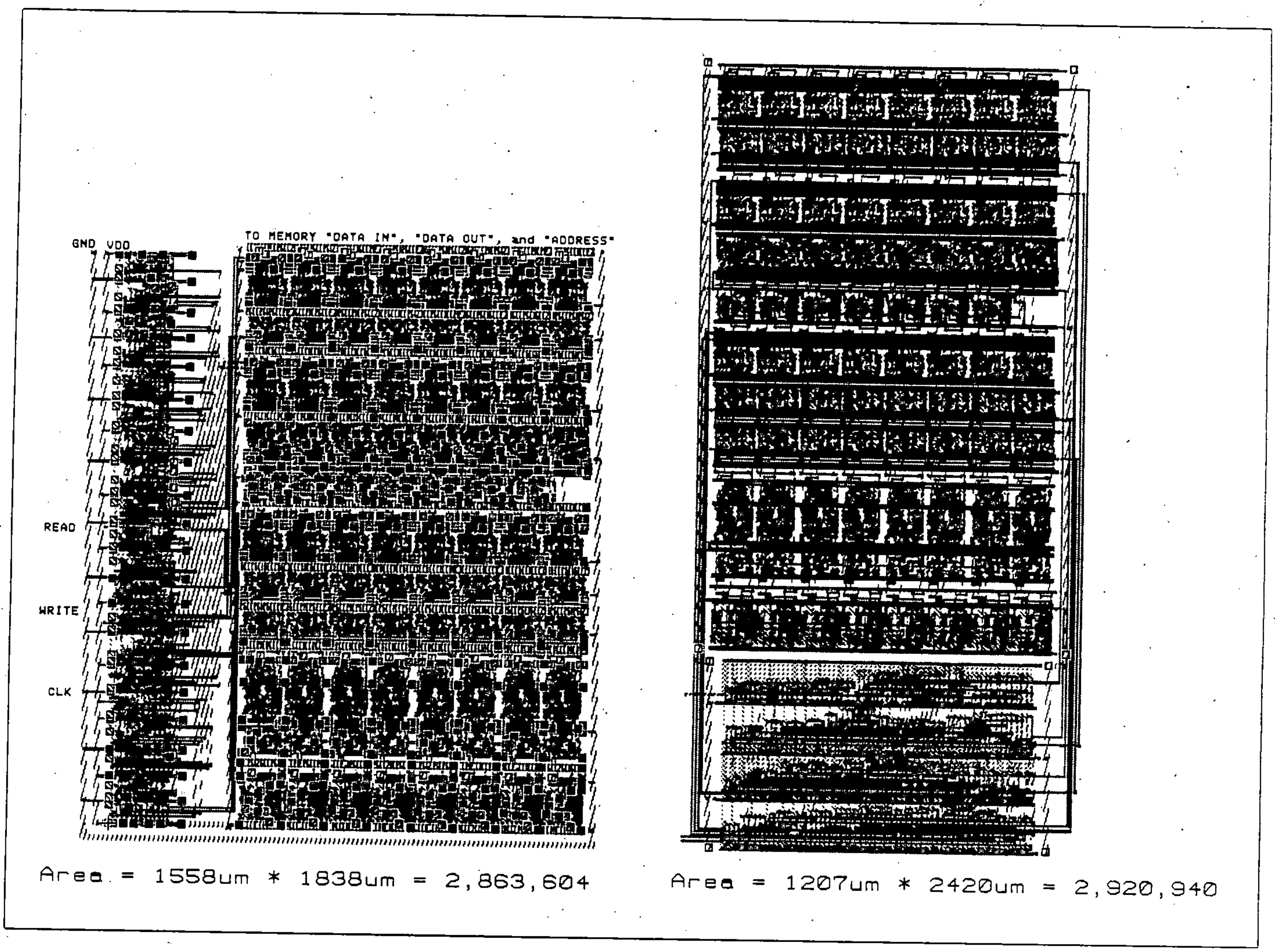


Figure 19. The layouts of a simple computer with 1:1 and 2:1 aspect ratios

unit micron	Slam with partitioning	Slam without partitioning	standard cells
H/W Area	620 * 780 483,600	934 * 582 543,588	740 * 782 578,680
%	0	+12.4	+19.6

Table 1. Layout comparison of the controlled counter

Unit micron	folded	unfolded	macrocell placement and routing
Total (W/H) Area	1046 * 997 1,042,862	1046 * 1380 1,443,480	1264 * 1175 1,485,200
%	0	+38.4	+42.4

Table 2. Layout comparison for stack folding implementation

unit micron	Slam with partitioning	Slam without partitioning
H/W Area	4250 * 2640 11,220,000	4250 * 3150 13,387,500
%	0	+19.3

Table 3. Layout comparison of the MARK1 computer

Finally, example 4 is a simple computer[Mano88]. The layouts for example 4 with 1:1 and 2:1 aspect ratios are shown in figure 19.

8. Conclusions

In this paper, we described a sliced layout architecture and presented two methods for performing partitioning, component partitioning and stack partitioning. The component partitioning algorithm decides which layout style, glue-logic or bit-slices, is best suited for each component based on the component's types and connectivities. The stack folding algorithm partitions glue-logic and bit-slices into clusters, and implements layout tradeoffs between bit-slices and glue-logic components to achieve better area utilization. The experimental results in Table 1 show that using the partitioning techniques and the sliced layout architecture, denser layouts are achieved in comparison with standard cells. The experimental results in Table 2 and Table 3 demonstrate that better area utilization can be achieved using the stack folding technique.

9. Acknowledgements

This work was supported by NSF grant #MIP-8711025. We are grateful for their support.

10. References

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

[BuMa85] Buric, M. R., and Matheson, T.G., "Silicon Compilation Environments," Proc. CICC, 1985.

[ChGa89] Chen, G. D. and Gajski, D., " An Intelligent Component Database System for Behavioral Synthesis," Tech. Report 89-39, ICS Dept., U.C. Irvine, 1989.

[HaSt71] Hashimoto, A. and J. Steven, "Wire Routing by Optimizing Channel Assignment within Large Apertures," Proc. of 8th DAC, 1971.

[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 Heuristics for Partitioning Graphs," Bell System Technical Journal, 49, (2), 1970.

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

[Lis89] Lis, J. S., "VHDL Structure Netlist Specification," CADLAB Internal Document, ICS Dept., U.C. Irvine, 1989.

[LiGa89] 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 DAC, 1989.

[Mano88] Mano, M. M., Computer Engineering Hardware Design, pp. 291, Prentice-Hall, 1988.

[PeWh86] Peterson, B. R., White, B. A., Salomon, D. J., and Elmasary, M. I., "SPIL:A Silicon Compiler with Performance Evaluation," Proc. ICCAD, 1986.

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

[SCS89] "GDT Database and Language Tools" Silicon Compiler System, Sec. 7, V.4.0, 1989.

[ScWe87] Schuck, J., Wehn, N., Glesner, M., and Kamp, G., "The ALGIC Silicon Compiler System: Implementation, Design Experience and Results," Proc. 24th DAC, 1987.

[SiGo82] 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.

[TrDi89] Trick, M. T., Director, S. W., "LASSIE:Structure to Layout for Behavioral Synthesis Tools," Proc. 26th DAC, 1989.

[VaCo86] Varinot P., J. A., Courtois B., J. R., "Principles of The SYCO Compiler," Proc. 23rd DAC, 1986.

[VaGa88] Vanzen Zanden, N., and Gajski, D., "MILO: A Microarchitecture and Logic Optimizer," Proc. 25th DAC, 1988.