4-24-1996

Layout Synthesis for Datapath Designs

Naveen Buddi
Portland State University

THESIS APPROVAL

The abstract and thesis of Naveen Buddi for the Master of Science in Electrical and Computer Engineering were presented April 24, 1996, and accepted by the thesis committee and the department.

COMMITTEE APPROVALS:

[redacted]

Malgorzata Chrzanowska-Jeske, Chair

[redacted]

Michael A Driscoll

[redacted]

Maria Balogh
Representative of the Office of Graduate Studies

DEPARTMENT APPROVAL:

[redacted]

Rolf Schaumann, Chair
Department of Electrical Engineering

****

ACCEPTED FOR PORTLAND STATE UNIVERSITY BY THE LIBRARY

by [redacted] on 3 October 1996

ABSTRACT

An abstract of the thesis of Naveen Buddi for the Master of Science in Electrical and Computer Engineering presented April 24, 1996.

Title: Layout Synthesis for Datapath Designs

As datapath chips such as microprocessors and digital signal processors become more complex, efficient CAD tools that preserve the regularity of datapath designs and result in small layout area are required. The standard-cell placement techniques ignore the regularity of datapath designs and hence give inefficient layouts. This has necessitated the development of new techniques for datapath module placement.

We developed a layout synthesis tool DataPathLAYOUT, for the bit-slice datapath logic designed using standard-cell libraries. We developed fast and area efficient heuristics for placing the cells in a bit-slice such that the regularity of datapath circuits is preserved and the number of channels in which a control signal is routed is minimized. The placement heuristics proposed here are general and also applicable to regular logic like *systolic arrays*. In addition, we propose a novel window-based heuristic, applicable to datapath and non-datapath circuits, for global routing of multi-terminal nets. We compared the area and run-time efficiency of the DPLAYOUT with an existing standard-cell placement and routing tool. We achieved 98-99% improvement in placement time, 28-33% improvement in area and 8-80% in total time. We conducted some experiments and demonstrated that for

standard-cell based datapath designs, bit-slice-based layout generation approach is superior to non-bit-slice-based layout generation approach both in terms of area and run-time. Finally, by providing interface to Verilog hardware description language, we developed a general tool which can be easily integrated with any high-level synthesis system. This tool is critical in any Datapath Silicon Compiler, to generate mask geometries from the behavioral level input specifications written in a hardware description language.

LAYOUT SYNTHESIS FOR DATAPATH DESIGNS

by

NAVEEN BUDDI

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

ELECTRICAL AND COMPUTER ENGINEERING

Portland State University

1996

ACKNOWLEDGEMENTS

I would like to express my deep appreciation and gratitude to Dr.Malgorzata Chrzanowska-Jeske for her encouragement and guidance in this work. I wish to thank Dr.Michael A Driscoll and Dr.Maria Balogh for serving on my thesis committee and for carefully reviewing my work. Thanks are due to all the committee members for their critical comments about my work.

I would like to thank Charles L.Saxe, Vince Ast of Tektronix Inc.for their cooperation and an interesting technical discussion throughout this work. This work was sponsored by a research grant from Tektronix Inc. and I would like to thank Tektronix Inc. for the support.

It is a pleasure to acknowledge the constant help from the faculty and administrative staff of Department of Electrical Engineering, PSU.

I wish to express my sincere thanks to my ex-colleagues at CAD(VLSI) lab of Micro-electronics and Computers Division, Indian Telephone Industries Ltd., India for the technical discussions which helped me in learning the various aspects of VLSI CAD tool development.

Finally, I would like to thank my present employer Cypress Semiconductor Corporation for making the time available for me to complete the work.

Table of Contents

List of Figures

List of Tables

1.INTRODUCTION

The datapath logic of microprocessors and digital signal processing circuits contributes significantly to the overall chip area. This logic is regular and repetitive.In cell-based designs, usually *datapath compilers* (tools that generate layouts directly from netlists) are used to generate the layouts. To achieve better performance the datapath compilers implement the datapath logic using cells from specialized (datapath) libraries [2,3]. However, the development and maintenance of specialized libraries requires an extra effort and causes cost overhead. In addition, not all vendors supply these specialized libraries. In such cases, the CAD tools use standard-cell library elements to implement the datapath logic. Traditional placement techniques [18] used in CAD tools ignore the regularity present in the datapath designs. Consequently, the CAD tools will not provide area efficient layouts. This has necessitated the development of new techniques for datapath module placement in standard-cell based designs.

In the past, researchers have proposed different techniques [5-8] to solve the datapath layout generation problem. More detailed description of these techniques is given in chapter 2. Some of these techniques [5] assume datapath libraries and some other [6-8] assume standard-cell libraries. The former techniques [5] use bit-slice based approach in which the layout of datapaths is generated by replicating the layout of one bit-slice. The standard-cell based techniques [6-8] do not effectively use the repetitive nature of bit-slice datapath designs. They follow non-bit-slice approach in which all the bit-slices are combined and treated as a single bit-slice. In this work, we solved the datapath layout problem using standard-cell

libraries by exploiting the bit-slice nature of datapath designs. The given circuit is divided into several bit-slices and the layout is generated for one bit-slice at a time. Then the layouts of all bit-slices are combined to obtain the layout of the given circuit. Our approach is faster than the non-bit-slice based approach because the size of the problem is reduced by considering a single bit-slice at a time. This is more significant when the number of leaf cells in a bit-slice is large which is typically the case in designs where large number of datapath functional blocks are stacked together. Our method is area efficient because the regularity of datapath circuits is preserved. We implemented this approach in a program called DPLAYOUT.

After experimenting with several designs, we demonstrate that for standard cell based datapath designs, efficient layouts can be generated by using the bit-slice based approach. Present day Datapath Silicon Compilers are capable of generating mask geometries automatically, from the behavioral or architectural level input specification described using a hardware description language. By providing interface to Verilog hardware description language [31], we made a general tool which can be integrated with any high-level synthesis system.

The remaining part of the thesis is organized as follows. Chapter 2 discusses datapath layout in general and the issues to be considered while designing a datapath layout synthesis system. The evolution of datapath layout tools and the previous work are also described in this chapter. In chapter 3, the layout model used in our work, the DPLAYOUT tool architecture and the importance of our work are described. The *Placement* and *Global Routing* heuristics developed are described in Chapters 4 and 5 respectively. Finally, the comparison between the DPLAYOUT and a standard-cell place and route tool, for a set of datapath designs, is given in chapter 6. Appendix 1 gives the details of the package and Appendix 2 shows

sample Verilog netlists and the layouts generated by DPLAYOUT.

2.DATAPATH LAYOUT GENERATION

Fig.2-1 shows typical floorplan of a microprocessor or a digital signal processor chip. It consists of one or more stacks of datapath logic, a random logic and memory. The datapath logic is regular and repetitive in nature when compared to the random logic.

Definition 2-1: A logic circuit is regular if it can be partitioned into slices (portion of a circuit) such that the functionality and logic structure in each of the slice is exactly the same or differ only slightly. And the connectivity between the slices is the same. *Systolic array* architectures, datapath logic circuits are examples of regular logic.

Each datapath stack is made up of many custom datapath macros (group of cells implementing a logic function), such as adders, shifters, registers, ALU etc. which form the data flow of the datapath functional units such as the fixed-point and floating-point execution units. Different approaches, namely, *bit-slice, macro-cell-based* and *array-based* datapaths are used for datapath design, based on different design styles.

In *bit-slice* approach, a datapath generator constructs the datapath by connecting several bit-slices in parallel, as shown in Fig.2-2. Each of the bit-slice performs the required function on a single bit of the data flow. The number of bit-slices in each of the datapath macro depends on the width of the datapath. In Fig.2-2, the datapath is 8-bit wide. These datapath macros are typically stacked up vertically (or horizontally) and wired almost exclusively with vertical (or horizontal) wires.

Macro-cell-based datapaths are typical of Digital Signal Processing (DSP) circuits. In these datapaths, module generators are used to synthesize the resources

(logic blocks like adders, multipliers, registers etc.). After synthesizing the resources, the resources have to be placed and wired. This method has been used by the CATHEDRAL-II compiler [24]. From the architectural level synthesis stand point this approach is more flexible (in terms of choice of resources) than one using *bit-slice* datapath synthesis approach. This is especially true when application-specific resources are needed (e.g., arithmetic operators with non-standard word lengths).

In the case of *array-based* datapaths, logic and physical synthesis techniques [18,30] are applied to the datapath. Thus, the datapath is treated no differently than other portions of the design. In general, *bit-slice* datapaths consume less area and perform better than datapaths designed in a *macro-cell-based* or *array-based* style [30].

Datapath layout generation can be done in two different ways. In the first method the datapath macros are treated as blocks and block placement and floorplanning techniques [17] are used to generate the layout. In the second method, bit-slice nature of datapath designs is used for generating the layouts. First the layout of a single bit-slice is optimized, then this bit-slice is replicated and finally all the bit-slices are combined to perform the routing. In cell-based designs, *datapath compilers* follow this method.

To realize the datapath logic, *datapath libraries* or *standard-cell libraries* are used. Fig.2-3 shows a typical cell structure in a datapath library and in a standard-cell library. In a datapath library, the cell pins (I/Os) are available only on one side but at multiple horizontal locations. Thus this pin structure allows abutment of the cells in a bit-slice as shown in Fig.2-4. In this figure, the cells (C_1, C_2, ... C_{12}) are placed in rows. The dotted vertical lines show connectivity among cell pins. When

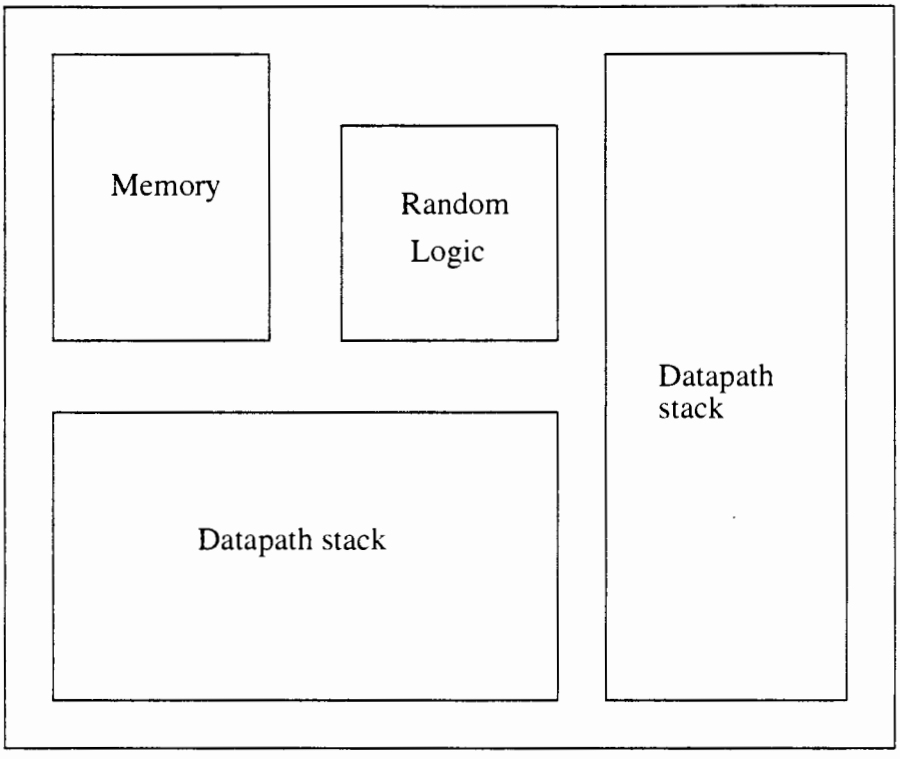


Fig.2-1 Floorplan of a datapath chip

the cells are abutted as shown in the figure, no space between rows is required for routing. However, some times this abutment is not possible and space is required to route the nets. For example, in Fig.2-4, space is needed between rows 3 and 4 to route the net connecting pins of cell C_8 and C_{12}.

Where as in a standard-cell library, the cell pins are available on both sides of the cell but at only one horizontal location (refer Fig.2-3). R.Leveugle et.al. [13] made an analysis of datapath library versus standard-cell library implementation of datapath designs and concluded that datapath library implementations are in most cases better from the area point of view than the standard-cell library implementations especially up to two level (layer) metal technologies. However, not all vendors offer these datapath libraries and they are not cost effective compared to

standard-cell libraries. Also, they are not efficient to realize the control logic. With recent advances in IC fabrication, many standard-cell library vendors are offering libraries with three or four levels (layers) of metallization. With more layers, routing can be done over the cells and hence the routing region between rows can be minimized. Using these standard-cell libraries the layout can be minimized if we

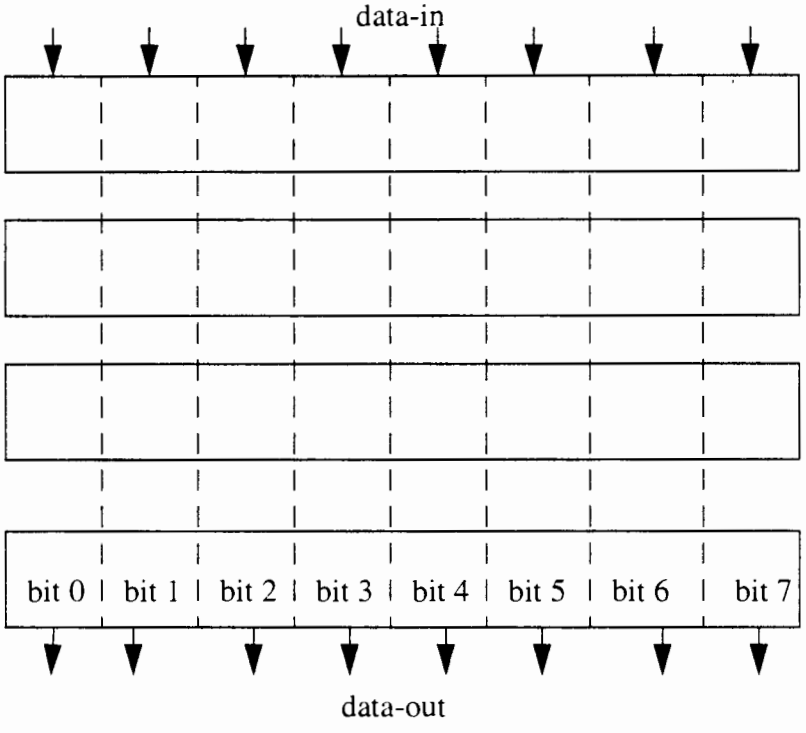
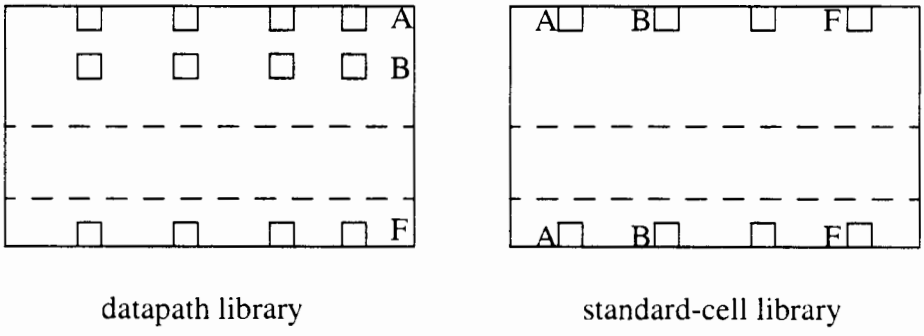


Fig.2-2 8-bit-slice datapath design



datapath library standard-cell library

Fig.2-3 Cell structure in a library

also take into account the regularity of datapath designs. Datapath libraries with more levels of metallization are expensive compared to the standard-cell libraries. Thus standard-cell libraries with efficient CAD tools and better technology give area comparable to datapath libraries, with less cost. At the same time, they offer the convenience of using the same library for the datapath logic and the control logic.

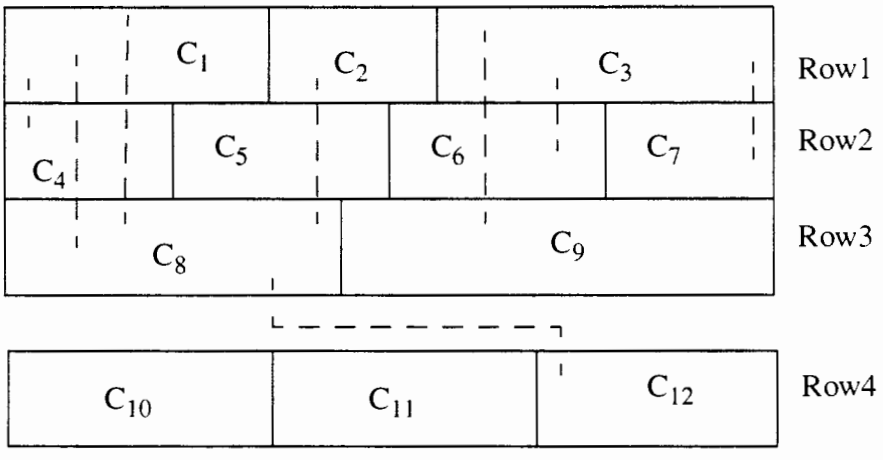


Fig.2-4 Abutment of cells

Several options need to be considered to automate the datapath layout generation. They are:

a. A number of datapath stacks: The number of datapath stacks (blocks) in the overall chip. In a single stack approach, entire datapath logic is implemented as one block. In multi-stack approach, datapath logic is partitioned into several blocks.
b. The composition of bit-slices: All the bit-slices are constructed from the same set of cells or different bit-slices are constructed from different sets of cells.
c. The library: The type of the library (datapath or standard-cell) used to imple-

ment the datapath logic.

d. The basic approach: Bit-slice approach or non-bit-slice approach.

Previous Work:

The early work on automatic datapath layout generation [25,26] has been focused on generating the layouts directly from the schematic description of a design. Later gate-arrays and cell-based designs gained popularity due to the need for fast design turn-around time. In cell-based designs, it is often more area efficient to build small datapaths using standard-cells because of floorplan flexibility. So the researchers addressed the problem of generating efficient netlist compilers [14]. These netlist compilers generate standard-cell and gate-array netlists from the same high-level specification. They do not synthesize any control logic, but focus on building the datapath logic within a design. Some efforts were also made to reduce the area of cell based datapaths by using bit-slice based module generators [5]. In these module generators, the chip designer can specify not only datapath parameter values (like number of bits) but also the resources (like number of multipliers, adders etc.) of the datapath.

Other class of tools [6,7] concentrated more on higher level optimization, that is, optimizing area of datapath chips by optimizing the number of datapath stacks and their placement. Luk and Dean [6] use a multi-stack approach in which the datapath logic cells are partitioned and assigned to several stacks with the objective of minimizing the number of nets crossing a stack. The placement within a stack is determined such that the misalignment of macros and a number of vertical wiring tracks used for routing, are minimized. However, they did not use the regularity present in the datapath designs. Wu and Gajski [7] used the regularity property of datapath designs, but concentrated only on partitioning the Register Transfer Level

(RTL) netlist to generate RTL-component layout. In [8], the authors extract similarity among several bit-slice components to form macros and solve the problem as the placement of the extracted macros. The authors did not effectively use the repetitive nature of bit-slice datapath designs because they consider several bit-slices at a time. Also their cell-matrix approach for placement leaves some empty slots in the matrix. The area wastage due to these empty slots is more significant when the macros have non-uniform width. The objective of the above tools is to ensure the datapath internal wirability, as well as external stack wirability to the other circuits and to minimize the wire lengths for routability and timing.

In custom-designs, like in microprocessors, large scale datapaths are conventionally hand-crafted to obtain a high density and high performance circuits. As the complexity of the circuits increases, the design effort for the datapaths is increased remarkably in the hand-craft methodology. To reduce the design effort, new hierarchical symbolic design methodology has been proposed [1]. In this, 1-bit field of the bit-slice structure is designed symbolically using gates as the primitives, then compacted and finally the entire datapath is generated. Later using a new cell structure called *stretchable cell with access free terminals*, efficient datapath layouts comparable to a hand-crafted one are generated [3].

As none of the previous work exploited the regularity of datapath logic for generating the layouts in standard-cell based designs, we addressed this, in our work. The subsequent chapters explain the methodology and the algorithms used in DPLAYOUT.

3.DPLAYOUT METHODOLOGY

As we can see from the previous work on the datapath layout generation, different tools have followed different approaches for generating the layouts of datapath circuits. No single tool gives efficient results for all the designs. In order to generate area efficient datapath designs, we need an integrated system which includes all the above approaches. An integrated datapath synthesis system with DPLAYOUT is shown in Fig.3-1. From the behavioral description, architectural level synthesis is done and Register Transfer Level (RTL) netlists are generated. These RTL netlists are partitioned into single or multiple stacks of datapaths. Each stack of a datapath logic is implemented using standard cell or datapath libraries. When standard-cell libraries are used, DPLAYOUT can be used to realize area efficient layouts. Finally, the layouts are combined to generate the complete layout of the datapath logic.

We assume that a datapath circuit is designed using standard-cell library components (cells). The layout model used is a single stack of rows with cells placed in rows. All data signals enter from the top of the stack and leave to the bottom of the stack. The control signals enter from the left side of the stack and leave to the right side of the stack. Fig.3-2 shows the layout model used in DPLAYOUT. A datapath is constructed by connecting several bit-slices in parallel. Slice0, slice1,.... slice(n-1) are n-bit slices. In each of the bit-slices, the cells are placed in rows. The cells are shown as filled boxes. Each of the bit-slices can consist of different cells.

The DPLAYOUT tool architecture is shown in Fig.3-3. The design is described in the form of a hierarchical Verilog netlist. A bit-slice is described as an entity in Verilog, and the datapath is described by instantiating these entities. Refer

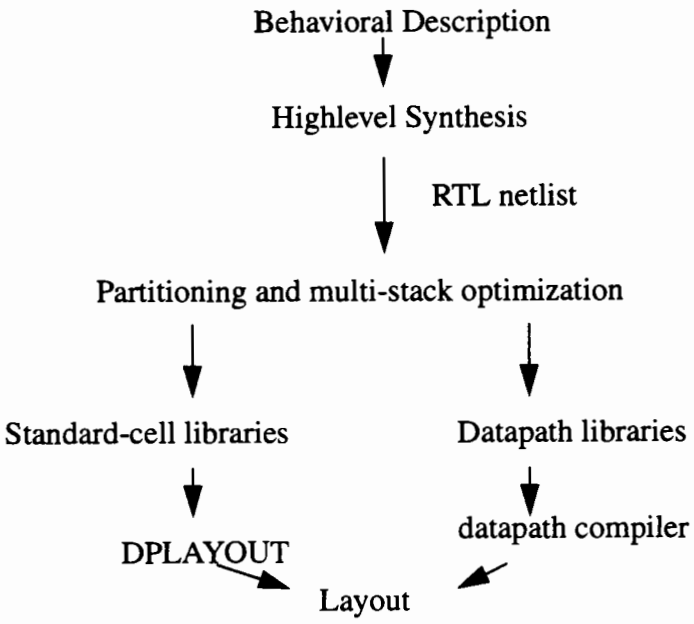


Fig.3-1 Datapath synthesis system

to Appendix 2 for the sample netlists. This interface to Verilog makes DPLAYOUT a general tool which can be integrated with any high-level synthesis system. The tool is modular and hence to handle hierarchical VHDL [32] netlists only the Verilog parser needs to be replaced with a VHDL parser.

Each of the bit-slices is constructed from a set of primitive cells present in the target standard-cell library. The input netlist is first analyzed and the bit-slices are classified into different types as follows. Each of the bit-slices is represented as a cyclic graph in which the nodes represent the cells and the edges represent the nets. Each node has a cell type attribute (represents the functionality of the cell) associated with it.

Definition 3-1: Two bit-slices *i,j* are treated as the same if the graphs representing the bit-slices are *isomorphic* and there is a one to one correspondence between the cell type attribute of the nodes of the two graphs.

The bit-slice order (the order in which the bit-slices are connected) is

extracted from the input netlist. After performing the placement and global routing of each bit-slice type, the bit-slices are abutted in the extracted order. Finally, the nets within a channel are routed using *greedy channel routing* technique [10]. The output of DPLAYOUT is a CIF (Caltech Intermediate Format, a standard format used to represent layout information) file containing the layout information.

Status: In our work we concentrated on developing area efficient placement and global routing algorithms, instead of spending our time on implementing a trivial task like bit-slice abutment. Currently DPLAYOUT reads Verilog netlists and generates layouts of each of the bit-slice types. Since bit-slice abutment is not implemented, we had to perform channel routing for each of the bit-slice type. In the subsequent chapters, placement and global routing heuristics used to create the layout of a single bit-slice are described.

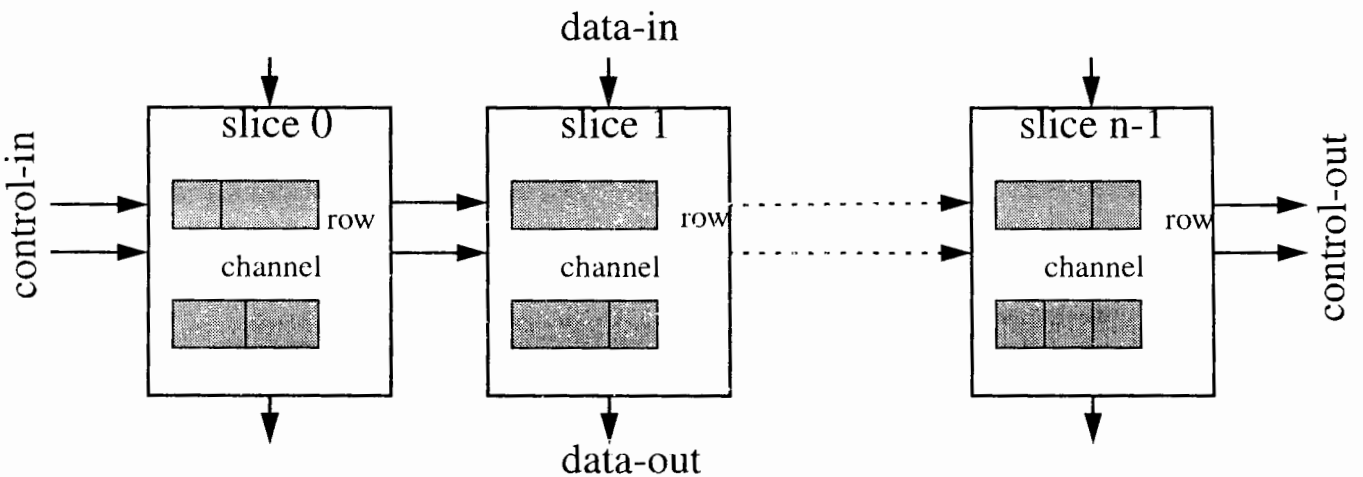


Fig.3-2 Layout Model of DPLAYOUT

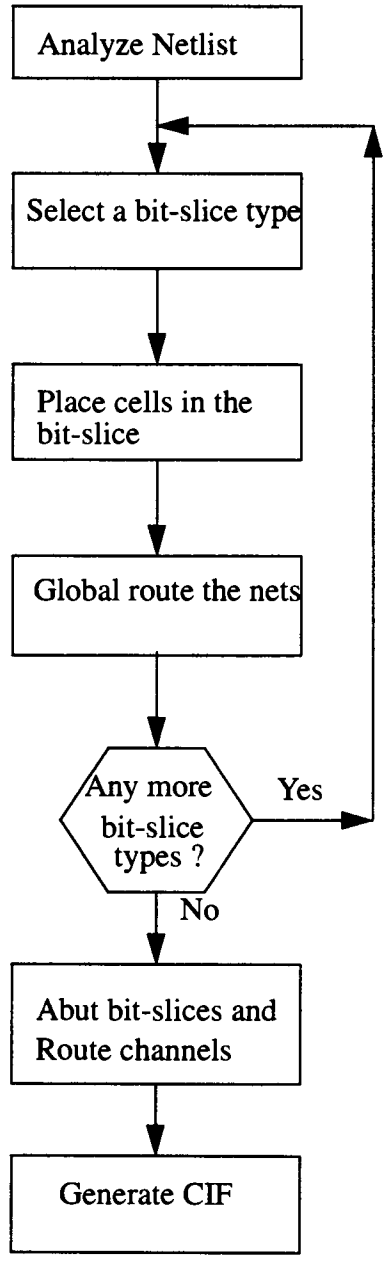


Fig.3-3 DPLAYOUT tool architecture

4.PLACEMENT

Placement is a key step in physical design cycle. During placement, the location of the cells on the chip is determined. The placement problem can be stated as follows. Given a circuit description (netlist), construct a layout indicating the position of each cell in the design such that all the nets can be routed with minimum crossovers and the total layout area is minimized. As mentioned in the previous chapter, in our approach, placement is determined for one bit-slice at a time. A more formal statement of the single bit-slice datapath placement is as follows.

Problem formulation: Let C_1, C_2, ... C_m be a set of cells in a bit-slice. Each cell has width $w = w_i$ and all the cells have the same height h. Let N = {N_1, N_2, ..., N_n} be the set of nets within the bit-slice. Let R_1, R_2, ..., R_r be the set of standard cell rows and B_1, B_2, ... , B_m be the set of bounding rectangles enclosing each of the cells C_1, C_2, ... C_m respectively. Then the placement problem is defined as assigning each cell C_i , a row R_j such that,

i) No two cells overlap. That is $B_i \cap B_j = 0$ for $1 \leq i, j \leq m$

ii) The total area of the rectangle bounding all the cells is minimized.

iii) The nets can be routed with less crossovers.

Based on the methodology used, traditional placement algorithms can be divided into two main classes, simulation based [4,27,28] and partition-based [15,29]. Simulation-based algorithms are iterative improvement algorithms. There are three major algorithms in this class: simulated annealing, simulated evolution and force directed placement. In simulated annealing, given a placement configuration, a change to that configuration is made by moving a component or interchanging locations of two components. Then the cost function is reevaluated with this

new configuration. All new configurations that result in a decrease in cost are accepted. New configurations that result in an increase in cost are accepted with a probability that decreases over the iterations.

Simulated evolution is analogous to the natural process of mutation of species as they evolve, to better adapt to their environment. The algorithm starts with an initial set of placement configurations, which is called the *population.* The individuals of the population are evaluated on the basis of certain fitness tests which can determine the quality of each placement. Two individuals among the population are selected as *parents* with probabilities based on their fitness. The better fitness an individual has, the higher the probability that it will be chosen. Then a set of operators called crossover, mutation and inversion, which are analogous to the counterparts in the evolution process, are applied on the parents to combine 'genes' from each parent to generate a new individual called the *offspring.* The offspring are then evaluated and a new generation is then formed by including some of the parents and the offspring on the basis of their fitness in a manner that the size of the population remains the same. This process is repeated until there is improvement in the overall placement quality. Refer to [18] for more details.

Force directed placement algorithm [16] is another example of simulation based placement techniques. It explores the similarity between placement problem and classical mechanics problem of a system of bodies attached to springs. In this method, the cells connected to each other by nets, exert attractive forces on each other. The magnitude of these force is directly proportional to the distance between the cells. According to Hooke's law, the force exerted due to stretching of the springs is proportional to the distance between the bodies connected to the spring. If the cells were allowed to move freely, they would move in the direction of the

force until the system achieved equilibrium. The same idea is used for placing the cells. The final configuration of the placement of cells is the one in which the system achieves equilibrium. Even though simulation based algorithms produce good quality placement, they are computationally expensive and can lead to longer run times.

Partitioning-based placement algorithms are an important class of placement algorithms. In these algorithms, the given circuit is repeatedly partitioned into two sub-circuits. At the same time, at each level of partitioning, the available layout area is partitioned into horizontal and vertical sub-sections alternatively. Each of the sub-circuits so partitioned is assigned to a sub-section. This process is carried out till each sub-circuit consists of a single cell and has a unique place on the layout area. During partitioning, the number of nets that are cut by the partition is minimized. There are other types of placement algorithms: cluster growth, quadratic assignment, resistive network optimization and branch-and-bound algorithms. Refer to [18] for details of these algorithms.

The above placement techniques ignore the regularity present in the datapath logic. So they result in inefficient layouts. After analyzing several datapath designs, we noticed that the area of the datapath layouts can be minimized by preserving the data-flow during the placement and by minimizing the number of channels in which a control signal is routed. With the placement shown in Fig.4-1(a), the control signal C is routed in two channels. To route this control signal one feed-through (a vacant position in a cell row that allows connection between two segments in two adjacent channels) is required. The same control signal can be routed in one channel, by changing the placement as shown in Fig.4-1(b). In this case, no feed-through is needed to route the signal C. Thus we can minimize the demand on the

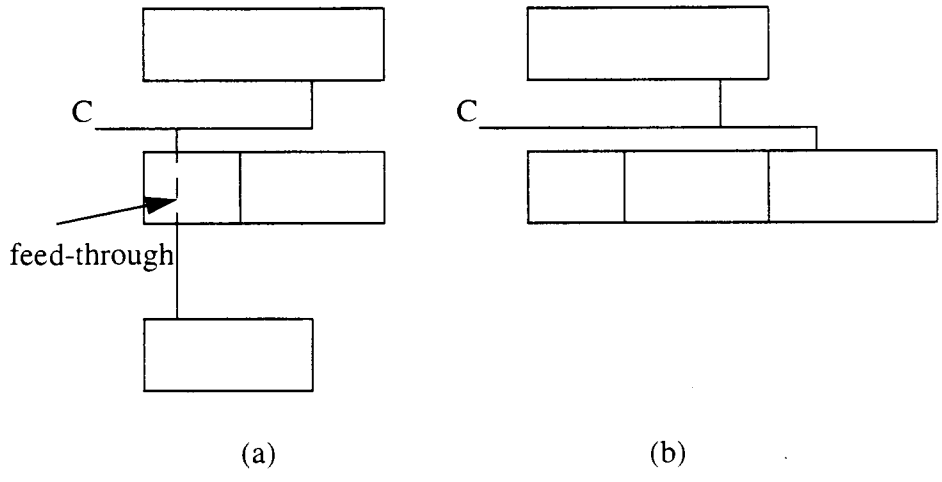


Fig.4-1 Control signal routing (a) in two channels (b) in one channel

feed-throughs by changing the placement.

Usually, in datapath circuits the same control signal is connected to several cells. Therefore, to maintain the data-flow a control signal has to be assigned to more than one routing channel (multiplication of control signals, as in Fig.4-1(a). The routing of a control signal can be limited to minimum number of channels (ideally to one, as in Fig.4-1(b)) by placing all the cells connected to that control signal in the same row or in two adjacent rows. However, this may violate the data-flow. Hence, while generating the layout, we can meet only one of the two objectives: preserving the data-flow or minimizing the multiplication of control signals. The placement heuristics proposed here minimizes the multiplication of control signals. The steps involved in the placement heuristic are described below.

4.1 Regularity preserving classification of control signals:

In the case of the bit-slice structure, control signals propagate through several bit-slices. This propagation can be of two types, *direct* and *indirect.*

Definition 4-1: Direct propagation occurs when a control signal is connected to cells belonging to different bit-slices.

Definition 4-2: Indirect propagation occurs when an output control signal of one bit-slice is connected as an input control signal to the next bit-slice.

Fig.4-2(a) shows both kinds of propagation. *Direct* propagation is shown as solid line and *indirect* propagation is shown as dotted line. The control signal connected to pins C, D, E, F is directly propagating from bit-slice *i* to bit-slice *(i+1).* Clock signal is a typical example of *direct* propagation. The net B in bit-slice *i* is connected as the net A in bit-slice *(i+1).* Therefore, nets A and B of the bit-slice *i* are involved in *indirect* propagation. This kind of indirect propagation is typical in adder circuits (the carry-out of one bit-slice is connected as carry-in of next bit-slice).

We classify the control signals into two categories, *related* and *unrelated.*

Definition 4-3: Related control signals are control signals involved with *indirect* propagation.

Definition 4-4: Unrelated control signals are control signals other than *related* control signals.

In Fig.4-2 A and B are related signals.

The first step in the placement is identifying the *related* control signals within a bit-slice type. These *related* control signals are treated as one signal during initial stages of the placement. DPLAYOUT also allows the designer to specify a set of control signals to be treated as *related.* This control signal classification preserves

the regularity of datapath designs, as we see from the following sub-sections.

4.2 Classification of the cells

The cells in the bit-slice are classified into groups, based on the following criteria. All cells that are connected to the same control signal or to *related* control signals are assigned to the same group. Each of the remaining cells is assigned to a separate group. The purpose of this classification is to place cells using the same control signal in the same row or in two adjacent rows. Fig.4-2(a) shows two bit-slices (*i, i+1*) in which the cells M_1, M_2 connected to the related signals A and B are assigned to the same channel. In Fig.4-2(b) the signals A and B are assigned to two different channels. In Fig.4-2(b) the path connecting nets A and B is longer than that in Fig.4-2(a). The classification of cells allows us to assign the *related* control signals to one channel as shown in Fig.4-2(a), thus minimizing the routing area. When a cell is connected to more than one control signal, then it is assigned to more than one group. The subsequent sub-sections describe how such cells are placed.

4.3 Row assignment

The cells classified into groups are assigned to rows using the following heuristic. The bit-slice netlist is represented as a directed cyclic graph G = (V, E) in which the vertices set V = {v_1, v_2, ...} represents cells and the edge set E = {e_1, e_2, ...} represents the nets (signals). The edge set represents both the data signals and the control signals. The direction of an edge represents the direction of the signal flow. Fig.4-3(a) shows a sample netlist and Fig.4-3(b) its graph representation (d1_in, d2_in are data input signals and d1_out, d2_out are data output signals.

c1_in, c2_in are control input signals and c1_out is control output signal). The edges representing the data signals are shown as solid lines and the edges representing the control signals are shown as dotted lines. Assuming that c1_in, c1_out are related control signals, then I1, I4, I5 belong to one group, say G_1, and I4, I6 belong to another group, say G_2. In this case, cell I4 is assigned to more than one group because it is connected to more than one control signals. The placement of such cells is discussed in the subsequent paragraphs. In the following description of the proposed heuristic, we use node and cell as synonyms.

The row assignment heuristic involves three phases.

Phase 1: The graph is traversed in a breadth-first manner, starting from the cells connected to data input signals. We assume that the order of data input signals is specified by the user. The edges representing the control signals are not considered during the traversal. For each of the nodes (cells) visited, the current level represents the row in which the cell has to be placed. In other words, cells connected to data-input signals are assigned to row one (top row). Cells within a group have to be assigned to the same row or to two adjacent rows. Whenever the first cell from a group is encountered during the traversal, then the channel below the row corresponding to the current level is assigned as the channel number of the group. Thus group G_1 is assigned to channel 1. Then all control signals connected to the cells in that group are assigned to that group's channel. Thus, c2_in and c1_out will be assigned to channel 1. If the group the visiting node belongs to, has already been assigned to a channel, then the visiting node is placed in a row below that channel. Otherwise, it is placed in a row corresponding to the current level. When a cell appears in more than one group, the row assigned to that cell is the channel number of the first group encountered during the graph traversal. Fig.4-3(c) shows place-

ment after this phase.

Phase 2: In this phase, we place all the unplaced cells, which are connected to the control signals. We repeat the following procedure for each of the control signals which have been assigned to a channel. For all the unplaced cells connected to the control signal, assign the row directly below the control signal's channel (for example, for the netlist in Fig.4-3(a), I6 is assigned to row 2). Fig.4-3(d) shows placement after this phase.

Phase 3: In this phase, we place all the remaining unplaced cells. The graph is traversed in depth-first manner starting from the data-output signals. Unplaced cells are assigned to rows using the same row assignment technique as in the first phase. The depth-first traversal backtracks when we encounter any of the input signals (control/data) or placed cells and terminates when all the cells are placed. This phase is required only if there are feedback signals in the given circuit or there are cells not in the data-flow (for example cell I7 in Fig.4-3). Fig.4-3(e) shows placement after this phase.

Analysis of the row assignment heuristics shows that in the first phase all the cells which constitute the data-flow are placed. In the second phase any glue logic associated with the control signals is placed and in the last phase cells in the feedback loop and cells not in the data-flow are placed.

This graph based technique preserves the data-flow and the grouping of cells minimizes the number of channels used by each control signal, thus the circuit layout area is minimized. The relative position of cells in a row is determined by the order in which the cells are assigned to that row.

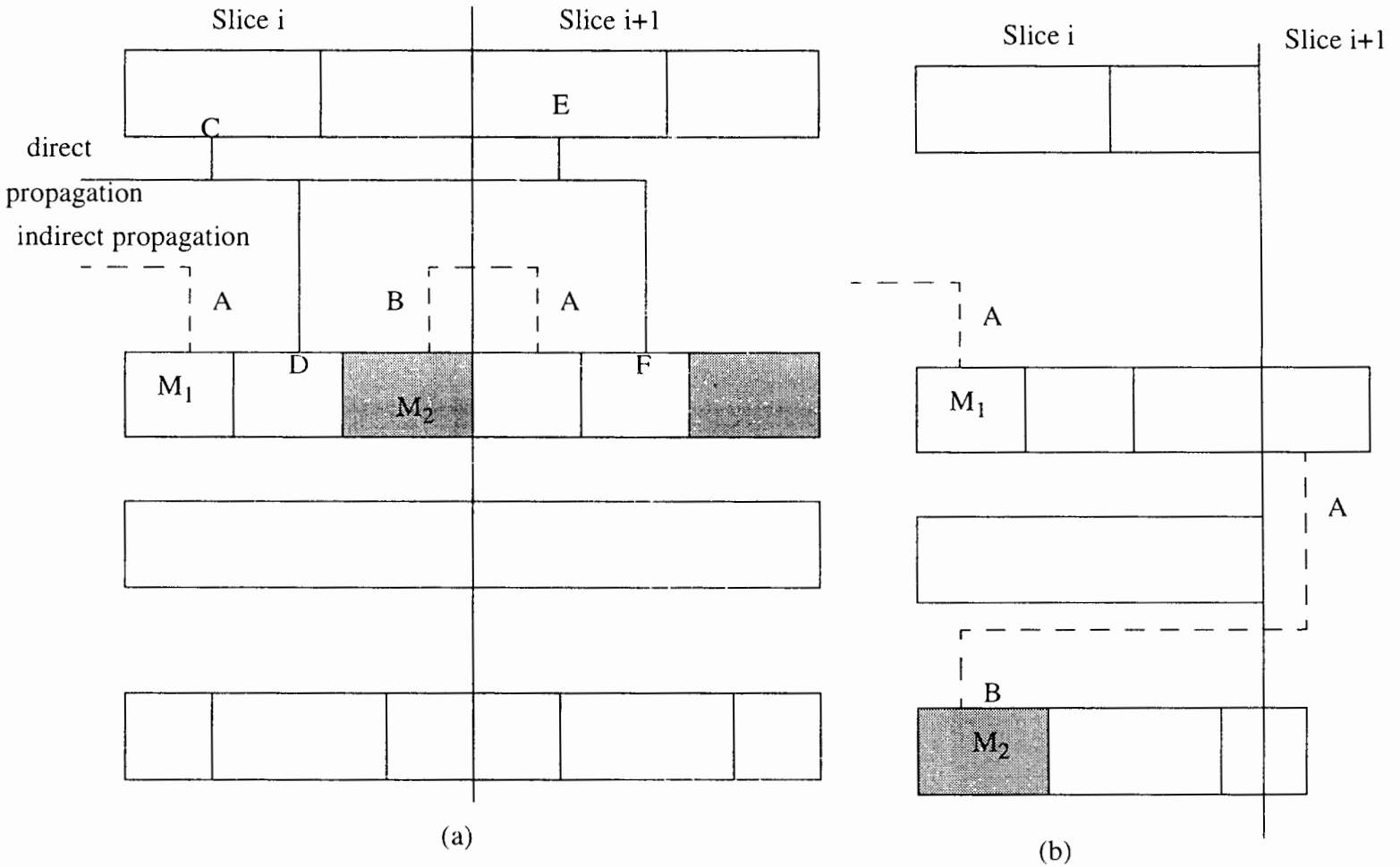


Fig.4-2 Classification of control signals (a) Control signal propagation (b) Related signals A,B assigned to different channels

d1_in, d2_in, c1_in, c2_in : IN ;
c1_out, d1_out, d2_out : OUT ;

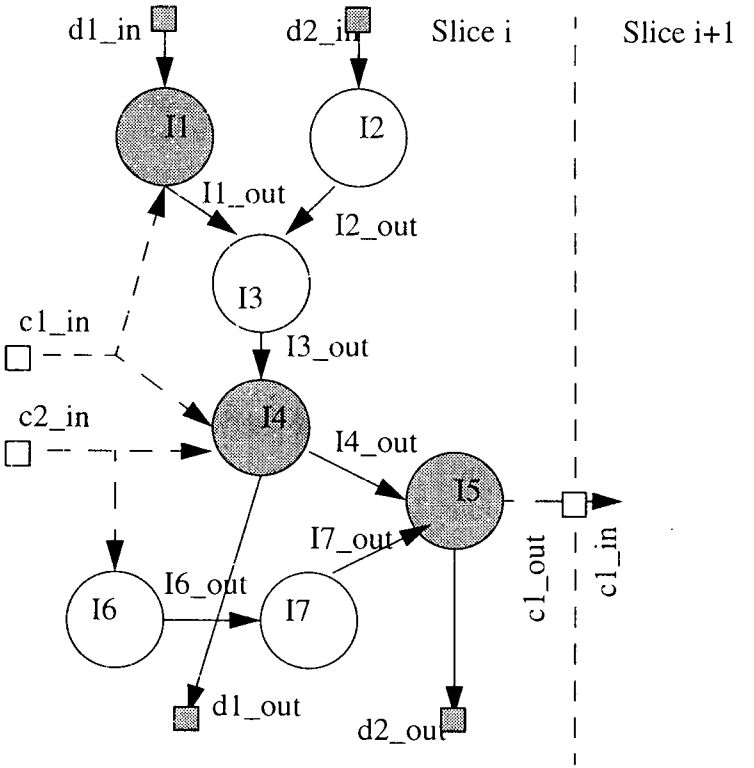
I1(d1_in, c1_in, I1_out) ;
I2(d2_in, I2_out) ;
I3(I1_out, I2_out, I3_out) ;
I4(I3_out, c1_in, c2_in, d1_out, I4_out) ;

I5(I4_out, I7_out, c1_out, d2_out) ;

I6(c2_in, I6_out) ;

I7(I6_out, I7_out) ;

(a)

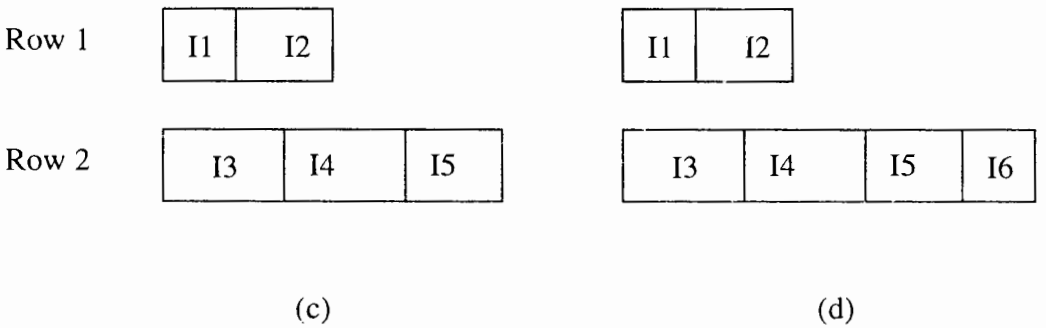


(b)

control signals
data signals
data input/output
control input/output

I1, I4, I5 assigned to same group

Fig.4-3 Row Assignment (a) Input netlist (b) Graph representation



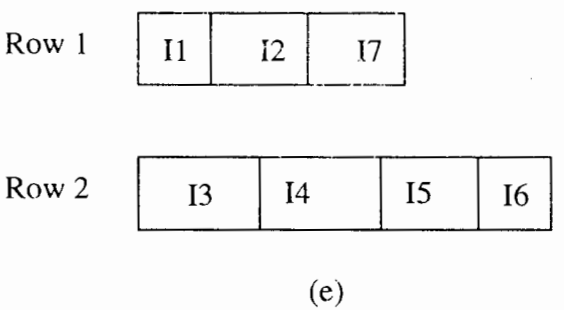


Fig.4-3 (cont.) Row assignment (c) after phase 1 (d) after phase 2 (e) after phase 3

4.4 Row merging

In this step, some of the rows are merged in order to maintain user specified aspect ratio. However, this step is not trivial because merging of rows may violate the above described minimization of MCS objective. So we only allow merging of complete rows so that the control signals in the channels adjacent to the merged rows need not be multiplied. The row merging algorithm is shown in Fig.4-4

```
for( all rows from top to bottom)
{
    if( there are two consecutive rows whose
        sum <= (factor) of maximum row size)
    {
        merge the two rows.
        update number of cells in the new row.
    }
}
```

Fig.4-4. Row merging algorithm

4.5 Complexity analysis

Let N be the total number of nets, C be the total number of cells and S be the total number of bit-slices in the design. Let I_c be the number of control input nets and O_c be the number of control output nets. Let C_p be the average number of pins in a cell.

Then the average number of nets in a bit-slice $N_i = O(N/S)$, the average number of cells in a bit-slice $C_i = O(C/S)$. The number of control input nets in a bit-slice $I_{ci} = O(I_c)$ and the number of control output nets in a bit-slice $O_{ci} = O(O_c)$. Assum-

ing that each pin of a cell is connected to a different net (implies each cell is connected to C_p nets), the average number of pins connected to a net $N_p = C_i C_p / N_i$.

Control signal classification: In order to classify the control signals, we need to compare each of the pins connected to control output signals with each of the pins connected to control input signals. Since the number of pins connected to control input signals = $I_{ci} N_p$ and the number of pins connected to control output signals = $O_{ci} N_p$, the worst case time complexity of this step is $O(I_{ci} O_{ci} N_p^2)$.

Cell classification: In order to classify the cells into groups, we need to visit each of the cells. The time complexity of this step is thus $O(C_i)$.

Row assignment: The row assignment algorithm involves visiting each of the cells and nets once. The time complexity of this step is $O(C_i N_i)$.

Thus the placement algorithm has a time complexity of $O(I_c O_c C^2 C_p^2 / N^2) + O(C/S) + O(CN/S^2)$.

5.GLOBAL ROUTING

During the placement step, the exact locations (row and position within a row) of cells are determined. The region between rows (channels) is used for routing the nets. In standard-cell based designs, the height of the channels is not fixed. The channel height can be varied by varying the distance between adjacent cell rows to accommodate the nets. In other words, the channels do not have predetermined capacity. The routing problem can not be solved in polynomial time [33]. Therefore, routing has traditionally been divided into two phases, *global routing* and *detailed (channel) routing*. During the *global routing* phase, the nets are assigned to various channels. And in the *detailed routing* phase the exact path of a net in a channel is determined. Fig.5-1(a) shows the pin positions of a sample netlist after placement. Pins with the same number belong to the same net. Fig.5-1(b,c) shows the two stages of routing. There is also a single phase routing approach, namely *area routing*. This technique is computationally expensive and is used in full-custom designs. In DPLAYOUT, we use two phase routing approach. In this chapter, the *global routing* algorithms used in DPLAYOUT are described. For *detailed routing*, we used the *greedy channel router* [10] implementation provided by Tektronix Inc.

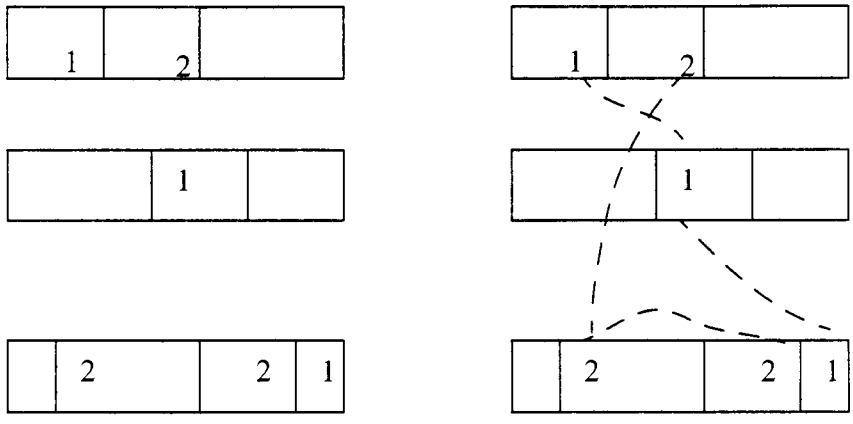
There are two approaches to solve the global routing problem.

1. *Sequential approach*: In this approach, nets are routed one after another. So whenever a net is routed, it may block other nets which are yet to be routed. As a result, this approach is very sensitive to the order in which the nets are considered for routing. Usually, the nets are ordered according to their criticality, perimeter of the bounding rectangle and the number of terminals. Typically, clock nets and nets

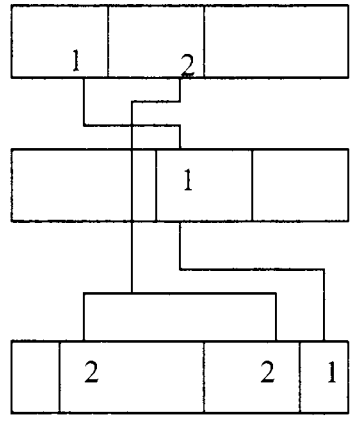
on the critical paths are assigned high criticality numbers since they play a key role in determining the performance of the circuit. This criticality based sequencing technique do not solve the net ordering problem completely, because it is the disposition of the cells and nets that plays role in determining the net routing order. So in addition to a net ordering scheme, often an improvement phase is used to remove blockages when further routing of nets is not possible. The blockages are removed by unrouting the interfering nets and rerouting them, to accommodate the routing of affected nets. This kind of improvement phase is known as *Rip-up and reroute* [19]. However, there is no guarantee that the *rip-up* and *reroute* gives optimal routing because unrouting a net means, loosing its optimum path [20].

2. *Concurrent approach*: In this approach, all the nets are routed simultaneously, thus avoiding the ordering problem present in the sequential approach. This approach is computationally hard and no efficient polynomial algorithms are known even for two-terminal nets. As a result, linear and integer programming techniques are suggested. Linear programming techniques [21,22] routes nets simultaneously using a randomized routing technique. This approach does not route multi-terminal nets optimally. Another approach [23] was a hierarchical method in which the problem is partitioned into a hierarchy of global routing sub-problems and each sub-problem is solved by integer programming. The solutions are then combined to obtain the solution of original global routing problem. However, the resulting global routing solution highly depends on the quality of the partitions and often is sub-optimal.

Usually, the sequential approach is used to route two-terminal nets. Multi-terminal nets are routed using either sequential approach or concurrent approach. Several methods were proposed to extend the two-terminal algorithms [18] to solve the



(a) Pin locations after placement (b) Global routing



(c) Detailed routing

Fig.5-1. Placement and routing of a sample netlist

multi-terminal net *global routing* problem. In these methods, the multi-terminal nets are decomposed into several two-terminal nets and the resultant two-terminal nets are routed by using two-terminal algorithms. This approach produces sub-optimal results. So multi-terminal nets are routed using Minimum Spanning tree (MST) approach. A Minimum Spanning tree connects all the nodes of a graph such that the total path length is minimum. Fig.5-2 shows paths connecting a three terminal net.

The path length in Fig.5-2(b) obtained by constructing MST is less than the path length in Fig.5-2(a). A better approach for routing of multi-terminal nets is Rectilinear Steiner tree (RST) approach. Rectilinear Steiner tree is obtained by adding intermediate points (Steiner points) to the MST such that all the net pins can be connected with minimum net length. Fig.5-2(c) shows a RST for the net in Fig.5-2(a) and the Steiner point is shown as dark circle.

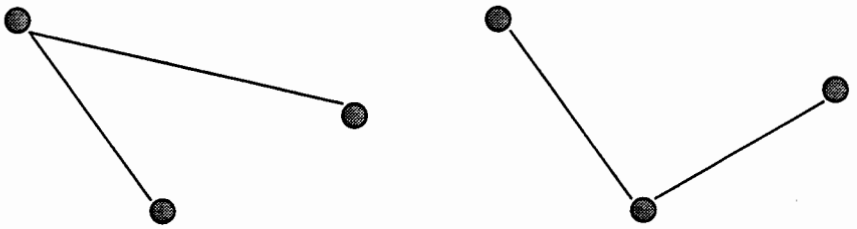
In our work, we propose a novel window based heuristic which is a combination of sequential and concurrent approaches. To my knowledge, window based technique is not used before, to solve the *global routing* problem. In DPLAYOUT, this technique has been integrated with a Minimum Spanning Tree based global routing algorithm [18].

5.1 Net ordering

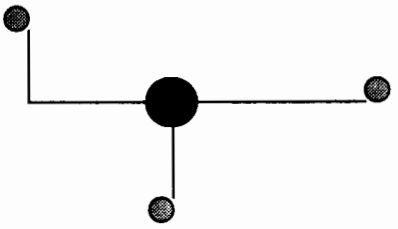
The Net ordering heuristic used in DPLAYOUT is described below.

a. The nets are ordered in the ascending order of the location of their right most pins (maximum columns). When a group of nets have the same maximum column they are sorted in the ascending order of the location of their left most pins (minimum columns)
b. If two or more nets have the same minimum and maximum columns, they are sorted in the descending order of their vertical spans.

Fig.5-3 shows a set of nets, their pin locations and the net order obtained from the above ordering heuristic. Fig.5-3(a) shows the pin positions and the pins of the same net have same number. Fig.5-3(b) shows net order obtained by selecting only the right most column of the nets. Fig.5-3(c) shows net order obtained by considering both the right most column and the left most column of the nets. Fig.5-3(c) shows the net order obtained by considering the left column, right column and ver-



(a) a path connecting all the net pins (b) MST



(c) RST

Fig.5-2 Paths connecting a 3-terminal net

tical span of the nets. For this example, net 2 has to be routed before net 3 and so on. In addition to the above technique, DPLAYOUT also allows the user to specify an order of his choice. For example, the designer can give highest priority to the critical nets in the design.

5.2 Window-based routing

A segment of a net is a path connecting two terminals of a net. Instead of global routing a net completely, we route only a subset of net segments and defer the routing of the remaining segments. To determine which subset of net segments to route first, we define a parameter called *window.* A *window* is a rectangular region with constant height and variable width. At the beginning of the global routing, the window width is set to a range 0 - W_s , where W_s is the window width specified by the user. The window height is always fixed and includes all rows of the bit-slice

layout. Fig.5-4(a) shows a sample *window,* which includes the pins A_1, B_1 of net 1 and pins E_2 and F_2 of net 2. For all the nets which originate in the current window, MST is constructed. For the net 1, the MST consists of the segments A_1-B_1 and B_1-C_1. For the net 2, the MST consists of the segments E_2-F_2 and E_2-G_2. Then for each net, only those edges in its MST that terminate in the current window are routed. For example, when the window size is as shown in Fig.5-4(a), only the net segments A_1-B_1 and E_2-F_2 are routed. After completing the routing of net segments in the current window, the window is moved to the right by W_s. This is repeated until all the nets are routed. Thus the net segments B_1-C_1 and E_2-G_2 are routed when the window is moved as shown in Fig.5-4(b).

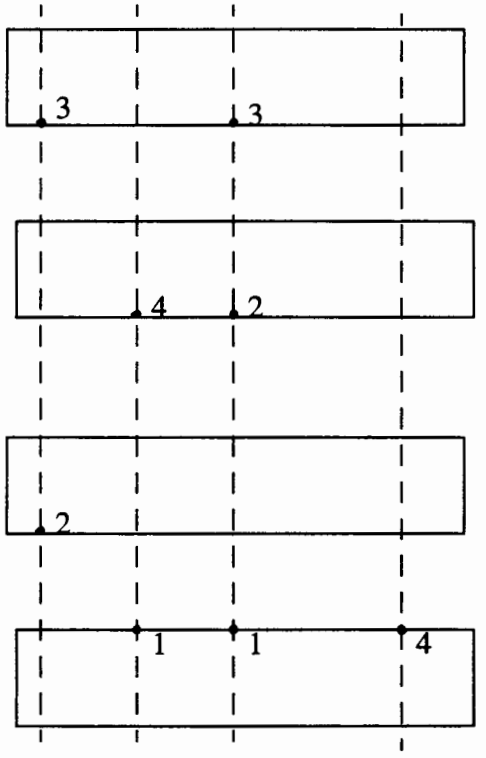
The advantage of the window-based routing technique is evident from the following example. Fig.5-5(a) shows two terminal nets N_1, N_2. When the window size is as shown in Fig.5-5(b), the net order for this example is N_1, N_2. When the feedthrough assignment for the net N_2, routed after N_1, results in the cell movement in row 2, then the path of the net N_1 is as shown in Fig.5-5(b). Since we are not rerouting the nets disturbed by the cell movement, the overall global routing solution will not be efficient. When the window size is small, as shown in Fig.5-5(c), the net N_1 will not be routed until its end point is visible in the window. That is net N_1 will be routed after the cell movement caused by net N_2 routing is completed. This example shows that the *window-based* technique results in better quality routing than the one obtained without this technique. The window size has an effect on the global routing quality only if the cells are moved during global routing.

The window-based global routing technique proposed here is general and is applicable to both datapath circuits and non-datapath circuits. Also it can be integrated with any global routing algorithms.

5.3 Complexity analysis

Net ordering: The net ordering algorithm involves sorting the nets based on their positions. *Quick sort* algorithm [34] is used to sort the nets. Each of the steps *a* and *b* in section 5.1 have a time complexity of $O(N_i \log N_i)$ [34], where N_i is the number of nets in a bit-slice.

Minimum Spanning Tree: Let N_p be the average number of pins of a net (refer section 4.5). Then the MST of each of the net can be constructed in a time complexity of $O(N_p^2)$ [11]. Thus global routing algorithm has a time complexity of $O((N/S) \log(N/S)) + O(C^2 C_p^2 / N^2)$ (refer section 4.5).



(a) Pin Positions

1, 3, 2, 4

(b) Net order based on location of maximum column

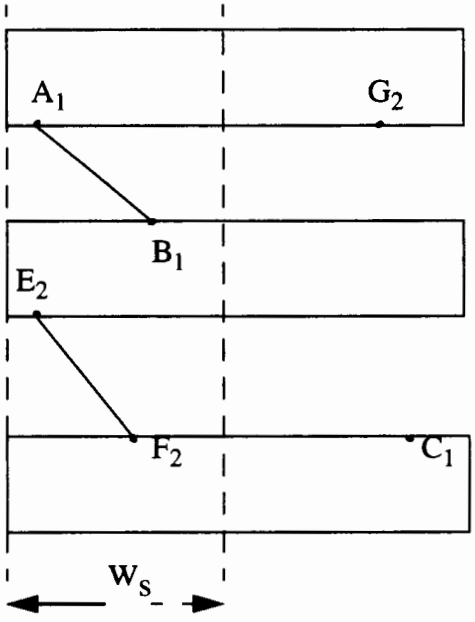
3, 2, 1, 4

(c) Net order based on location of maximum column and minimum column

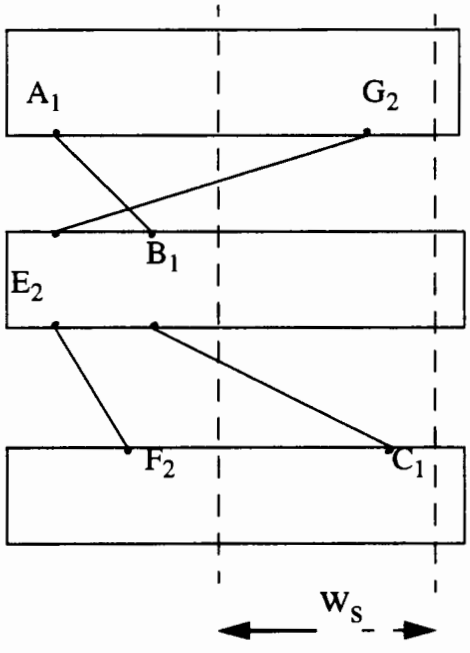
2, 3, 1, 4

(d) Final net order based on minimum column, maximum column and vertical span

Fig.5-3 Net ordering scheme



(a) Initial window position. Segments A_1-B_1 and E_2-F_2 are routed



(b) New window position after it is moved. Segments B_1-C_1 and E_2-G_2 are routed

Fig.5-4 Window based global routing

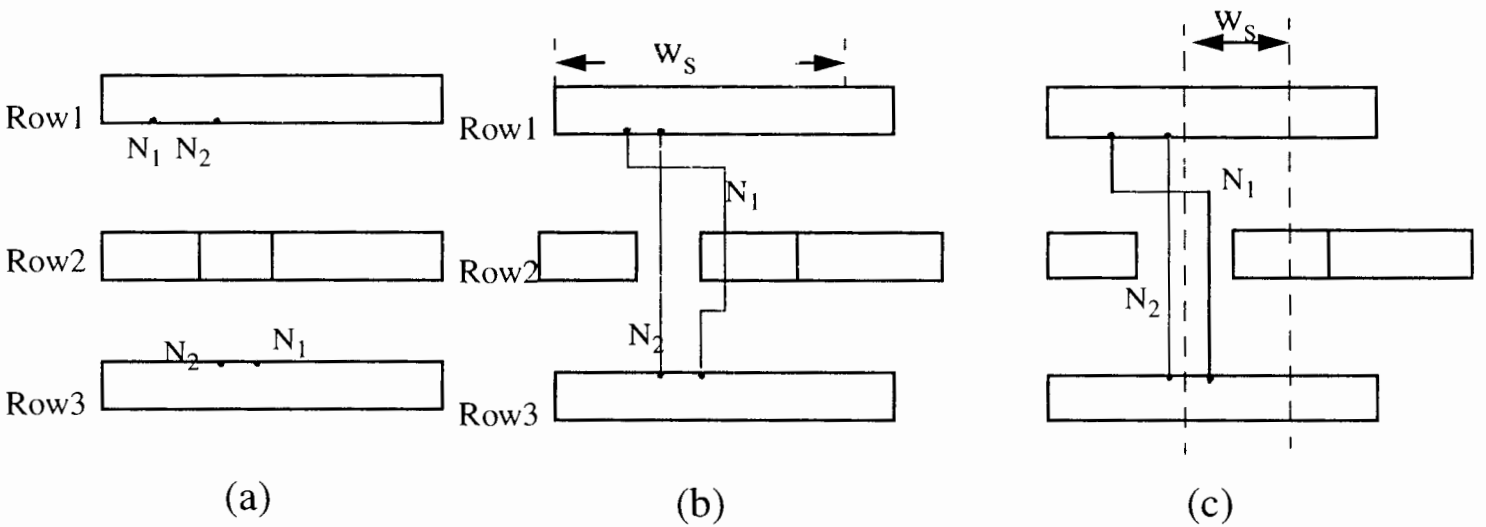


Fig.5-5 Advantage of window based routing (a) Net pin positions (b) Net route after a cell is moved for large window (c) Net route after a cell is moved for small window

6.RESULTS

DPLAYOUT is implemented in C under UNIX environment. We conducted experiments to evaluate the run-time and area efficiency of DPLAYOUT. We compared the results of DPLAYOUT with a standard-cell placement and routing tool (*SCR*) in the *ALLIANCE CAD* package [12], for a set of data path designs. *ALLIANCE* is a CAD package developed at University of Paris. We compared with this package because it is a complete CAD tool available in the public domain and using this package layouts can be generated from the behavioral description of a design. The design statistics and results are shown in Table I. The examples selected here represent bit-slices of a wide class of datapath circuits. Ex1, Ex7-10 are selected from DLX RISC processor implementation available in *ALLIANCE*. Ex2-6 are bit-slices of datapath circuits used in industry designs.

Ex1 is single bit-slice of an *adder-accumulator* and the layout generated by DPLAYOUT is shown in Fig.6-1. The graph corresponding to this netlist, placement before and after row merging and global router output are shown in Appendix2. Ex2-5 are bit-slices of datapath designs. Ex6 is a bit-slice of an ALU. Ex7 and Ex8 are single bit-slices of 8x16 bit *fifo* and 4-bit *ram* circuit, respectively. Ex9 is 8 bit *fifo* and Ex10 is 4-bit *ram*. All the results were obtained using the complete row-merging heuristic. The time shown is measured on a SUN SPARC-2 workstation. The total CPU time reported is the combined time for input parsing, placement, routing and layout file generation. No *SCR* time data comparison is done for the designs Ex2-Ex6 because their library is different from *ALLIANCE* library. For all the single bit-slice circuits (Ex1-Ex8), the area and run time of

DPLAYOUT is better than that of *SCR*. We achieved 98-99% improvement in placement time, 28-33% improvement in area and 8-80% in total time.

We also compared DPLAYOUT with *SCR* for non-bit-slice datapath circuits (Ex9,Ex10). The results of *fifo* show that even when the circuit is not partitioned into bit-slices, DPLAYOUT outperforms *SCR* for more regular datapath circuits (*fifos, register files* etc.). However, traditional placement methods [18] used in *SCR* won over our algorithms when the datapath circuits have more random logic associated with them (4-bit RAM results).

We also compared the efficiency of the bit-slice based layout generation approach with the non-bit slice based layout generation approach and the results are shown in Table II. All the bit-slices of the above 8x16 bit *fifo* and 4-bit *ram* are submitted to DPLAYOUT and *SCR* as one bit-slice. Considering the fact that area of the datapath circuits is proportional to the number of bit-slices, the area of the complete circuit must be close to n-times the area of single bit-slice, where n is the number of bit-slices in the circuit. The same should be true of the total CPU time (shown in rows 2 and 6 in Table-II). However, for both DPLAYOUT and *SCR*, the total time and area obtained using non-bit-slice approach are more than n-times the time and area of respective single bit-slices. This demonstrates that for datapath circuits, bit-slice based layout generation approach has better area and run-time efficiency over non-bit slice based layout generation approach.

We noticed that for the examples in Table-I, window size has no affect on the total area. Analysis of the results shows that because the placement heuristics used in DPLAYOUT preserve the data-flow, no cell movement is involved. However, for large designs often the cells are moved to accommodate the feed-throughs. Whenever there is a cell movement, window based technique gives better global routing

quality as we see in the previous chapter. In order to find the effect of window-based global routing, some more experiments needs to be conducted on large circuits using traditional placement techniques.

Finally, DPLAYOUT tool details are discussed in Appendix-1. The Verilog netlists and the generated layouts of some of the designs in Table-I are included in the Appendix-2.

Complexity analysis: Refer to sections 4.5 and 5.3 respectively for detailed discussion of the time complexities of the placement and global routing algorithms.

Table-I: DPLAYOUT Results and Comparison with SCR tool in ALLIANCE Package

			DPLAYOUT			SCR		
Design	# cells	# nets	place time(sec)	total time(sec)	area(sq.mm.)	place time(sec)	total time (sec)	area(sq.mm.)
Ex1	9	17	0.01	0.32	0.02	0.5	1.87	0.03
Ex2	17	37	0.01	4.7	0.008	-	-	-
Ex3	21	38	0.02	6.9	0.012	-	-	-
Ex4	16	31	0.01	4.1	0.007	-	-	-
Ex5	16	29	0.01	3.8	0.007	-	-	-
Ex6	11	23	0.02	1.5	0.004	-	-	-
Ex7	16	35	0.01	0.6	0.057	0.59	3.0	0.08
Ex8	58	107	0.03	7.1	0.18	2.4	7.75	0.25
Ex9	128	154	0.03	12.8	0.6	5.3	46	1.35
Ex10	232	284	0.07	94	1.45	10.5	54.25	1.37

Table-II: Comparison of bit-slice approach and non-bit-slice approach

			DPLAYOUT		SCR	
Design	# cells	# nets	total time(sec)	area (sq.mm.)	total time(sec)	area (sq.mm.)
FIFO-1	16	35	0.6	0.057	3.0	0.08
8*FIFO-1	128		4.8	0.456	24	0.64
FIFO-8	128	154	12.8	0.6	46	1.35
RAM1	58	107	7.1	0.18	7.75	0.25
4* RAM1	232		28.4	0.72	31.0	1.0
RAM4	232	284	94	1.45	54.25	1.37

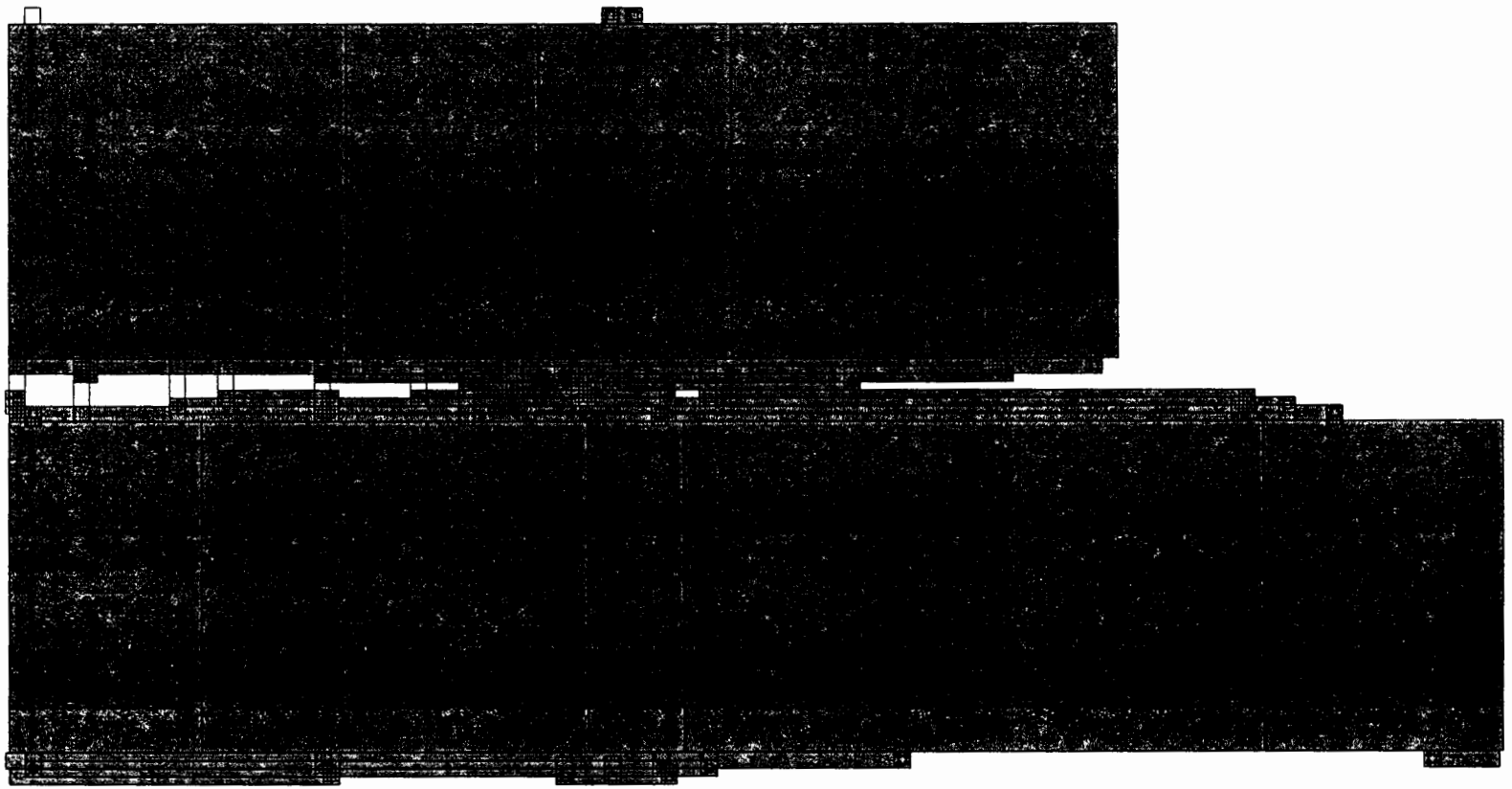


Fig.6-1 Single bit-slice *adder accumulator* circuit layout

7.CONCLUSIONS

The thesis work describes an efficient and fast approach for generating layouts of bit-sliced datapath circuits designed using standard-cell libraries. We developed efficient data-flow preserving heuristics for placement. The placement heuristics exploit the regularity characteristic of datapath designs and attempt to route a control signal in minimum number of channels. We proposed a *window-based* global routing technique which gives efficient routing without any rip-up rerouting when the cells are moved to accommodate the feed-throughs.

We also demonstrated that for standard cell based datapath circuits efficient layouts can be achieved when the circuits are partitioned into bit-slices and the bit-slices are handled separately. The developed tool is a general tool which can be easily integrated with any high-level synthesis system. The row merging algorithm leaves empty gaps within a bit-slice. So it needs to be improved. Also delay optimization algorithms can be included in the proposed placement and global routing heuristics to minimize the delay.

The placement heuristics proposed here are general and applicable to any regular logic like datapath and systolic arrays. The possibility of using the proposed methodology to solve the Register-Transfer level component layout generation problem needs to be investigated, in order to achieve efficient datapath layouts.

8. REFERENCES

[1]. K. Usami et.al., "Hierarchical Symbolic Design Methodology for Large-Scale Data Paths", in *IEEE Journal of Solid-State Circuits,* Vol. 26, No. 3, pp. 381-385, March 1991.

[2]. R. C. Mason and M. T. Fertsch, "A Bit-modular Cell Library Optimized for Datapath Applications", in *Proc. of ISCAS,* pp. 10-14, 1986.

[3]. Y. Tsujihashi et.al., "A High-Density Data-path Generator with Stretchable Cells", in *IEEE Journal of Solid-State Circuits,* vol. 29, No. 1, pp. 2-7, Jan.1994.

[4]. C. Sechen and A. Sangiovanni-Vincentelli, "The TimberWolf Placement and Routing Package", *IEEE J.Solid-State Circuits,* vol. 20, pp. 510, April 1985.

[5]. C. B. Shung et.al., "An Integrated CAD System for Algorithm-Specific IC Design", in *IEEE Trans. on CAD,* vol. 10, no. 4, pp. 447-462, April 1991.

[6]. W. K. Luk and A. A. Dean, "Multistack Optimization for Datapath Chip Layout", in *IEEE Trans. on CAD,* vol. 10, no. 1, pp. 116-129, Jan. 1991.

[7]. A. C. H. Wu and D. D. Gajski, "Partitioning Algorithms for Layout Synthesis from Register-Transfer Netlists", in *IEEE Trans. on CAD,* vol. 11, no. 1, pp. 453-463, April 1992.

[8]. C. E. Cheng and C. Ho, "SEFOP: A Novel Approach To Datapath Module Placement", in *Proc. of ICCAD,* pp. 178-181, Nov. 1993.

[9]. W. Swartz and C. Sechen, "A New Generalized Row-Based Global Router", in *Proc. of ICCAD,* pp. 491-498, 1993.

[10]. R. L. Rivest and C. M. Fiduccia, "A Greedy Channel Router", in *Proc. of Design Automation Conf.,* pp. 256-262, 1982.

[11]. N. Deo, *Graph Theory with Applications to Engineering and Computer Science*, Prentice-Hall International Inc., Reading, 1974.

[12]. *ALLIANCE CAD system-2.0,* Laboratoire MASI/CAO-VLSI, University Pierre et Marie Curie, PARIS, FRANCE.

[13]. R.Leveugle et.al., "Datapath implementation: bit-slice structure versus standard cells", in *Proceedings of EURO ASIC,* pp.83-88, 1992.

[14]. Jim Rowson, Bill Walker and Suresh Dholakia, "A Datapath Compiler for Standard cells and Gate Arrays", in *Proceedings of Custom Integrated Circuits Conference,* pp.149-152, 1987.

[15]. B.W.Kernighan and S.Lin, "An Efficient Heuristic Procedure for Partitioning Graphs", *Bell Syst.Tech.J.,* vol.49, no.2, pp.291-307, Feb.1970.

[16]. N.R.Quinn, "The Placement Problem as Viewed from the Physics of Classical Mechanics", *Proceedings of the 12th Design Automation Conf.,* pp.173-178, 1975.

[17]. W.W.Dai et.al., "Hierarchical Placement and Floorplanning in BEAR", *IEEE Trans. on CAD,* vol.8, No.12, Dec.1989.

[18]. N.A.Sherwani, "Algorithms for VLSI Physical Design Automation", *Kluwer Academic*, Reading, 1995.

[19]. W.A.Dees and P.G.Karger, "Automated Rip-up and Reroute Techniques," *Proceedings of Design Automation Conference,* 1982.

[20]. R.Nair, "A Simple Yet Effective Technique for Global Wiring", *IEEE Trans. Computer-Aided Design,* Vol. CAD-6, No.2, pp.165-172, March 1987.

[21]. A.Ng., P.Raghavan and C.Thompson, "Experimental Results for a Linear Program Global Router," *Computers and Artificial Intelligence,* 1987.

[22]. P.Raghavan and C.D.Thompson, "Multiterminal Global Routing: a Deterministic Approximation Scheme," *Algorithmica,* Vol.6, No.1, pp. 73-82, 1991.

[23]. J.Heisterman and T.Lengauer, "The Efficient Solution of Integer Programs for Hierarchical Global Routing, "*IEEE Trans. Computer-Aided Design,* CAD 10(6), pp.748-753, June 1991.

[24]. D.Gajski, Editor, *Silicon Compilation,* Addison-Wesley, Reading, MA, 1987.

[25]. Jerraya, Varinot, Jamier and Courtois, "Principles of the SYCO Compiler", *Proceedings of the 23rd Design Automation Conference,* 1986.

[26]. Marshburn et.al., "DATAPATH: A CMOS Datapath Silicon Assembler", *Proceedings of the 23rd Design Automation Conference,* 1986.

[27]. H.Chan, P.Mazumder and K.Shahookar, "Macro-cell and module placement by genetic adaptive search with bitmap-represented chromosome", *Integration: the VLSI Journal,* Vol.12(1), pp.49-77, November 1991.

[28]. K.Shahookar and P.Mazumder, "A genetic approach to standard cell placement using meta-genetic parameter optimization", *IEEE Trans. Computer-Aided Design,* pp.500-511, May 1990.

[29]. B.Krishnamurthy, "An improved mincut algorithm for partitioning vlsi networks", *IEEE Trans. on Computers,* pp.438-446, 1984.

[30]. Giovanni De Micheli, *Synthesis And Optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.

[31]. Donald E.Thomas and Philip R.Moorby, *The Verilog Hardware Description Language*, Kluwer Academic Publishers, 1995.

[32]. R.Lipsett, C.Schaefer and C.Ussery, *VHDL: Hardware Description and Design,* Kluwer Academic Publishers, 1993.

[33]. Thomas Lengauer, *Combinatorial Algorithms for Integrated Circuit Layout,* John Wiley and Sons Ltd. 1990.

[34]. E.Horowitz and S.Sahni, *Algorithms and data structures*, Galgotia Publications, 1982.

APPENDIX 1: PACKAGE DETAILS

A.1.1 Synopsis:

DPLAYOUT module generates gate-level layout of a design described in the form of a netlist.

A.1.2 Usage:

dplayout -n <netlist> -t <pdf_file> -p <port_side> -o <out_file> -w <width control factor(%of max. row width)> -g <pgcell> -s [<window size>] -d [<minimum channel density>] -m [allow row merging/not(1/0)]

where

netlist: is a file containing circuit description in Verilog format.

pdf_file: is pin description file. This file contains library information (cells, pins and their position).

port_side: is a file containing description of side of module ports (primary inputs and outputs).

pgcell: is a pin description file for power and ground cells.

out_file: is name of the output file in which DPLAYOUT writes CIF data.

factor: is aspect ratio control factor. It is used to control the width and hence the aspect ratio (defined as overall width/height).

size: is window size (in terms of columns).

density: is minimum channel density. The minimum channel density to be considered while doing global routing.

allow row merging/not: boolean flag that controls row merging.

Each file name above contains complete path, absolute or relative, with respect to directory from which dplayout is invoked.

A.1.3 Error Messages:

The following is description of the error messages and the actions to be taken when an error occurs. An error message *Internal error* indicates that there is a bug in the software which ultimately leads to inconsistency. The only action the user can take in such cases is report to the person responsible for code maintenance.

Netlist file not specified: Make sure that the netlist file is present and you have permissions to access it.

PDF file not specified: Make sure that the Library information file is present and you have permissions to access it.

Port sides file not specified: Make sure that the file containing the port side information is present and you have permissions to access it.

Output file not specified: Make sure that you specify a file to write the CIF output.

Module <name> port type not specified: The type (Input/Output) of module (<name>) port is not specified. Check the netlist and specify the type.

Net <name> declared more than once : The net (<name>) is declared more than once. Check the netlist and fix the bug in it.

Net/Port <name> not in the module port list : A net/port specified is not present in the module interface list. Check the netlist to fix the bug.

Module <name> instantiated in itself : A module can not be instantiated within itself. Check the netlist to fix it.

Module <name> has one unspecified port : In the module port list, extra delimiter(s) ',' appears.

Instance has one unspecified port: An instance port is not present in the corresponding module-port list. i.e. there is some inconsistency in the number of ports of an instance and a module.

Illegal format <pdf> file: Check the format of <pdf> file.

Illegal format <pgcel> file: Check the format of <pgcel> file.

Illegal format <port side> file: Check the format of <port side> file.

No Power_Ground cell file: Make sure that you specify a file that contains power and ground cells and you have access to it.

Too many ports for a module: Increase the constant MAX_MODPORTS, recompile and rerun.

Unable to identify top level of the hierarchy : There is inconsistency in the data structures, due to which the top level of hierarchy could not be determined. Report the bug.

Net <name> not in the port list of the top level module: A net specified in the <port side> file is not present in the netlist file.

Improper array declaration: Syntax error in Bus declaration. Check the netlist and fix it.

No instances to place or nets to route: Either no instances to place or no nets to route. So running DPLAYOUT does not make sense.

Illegal cell in pgfile: A cell in the <pgcel> file is illegal.

Bit-slice Instance without cells: Inconsistency. Report the bug.

Primitive cell encountered while grouping: Inconsistency. Report the bug.

Primitive cell encountered while assigning channels: Inconsistency. Report the bug.

Unable to place all the cells: Report the bug.

Improper instance row assignment: Inconsistency. Report the bug.

Net without any instance pins: There is a net connected to only module pins. i.e. not connected to any instance. The present version can not handle this case.

Internal Error.000: Inconsistency in the data structures. Report the bug.

Gap feed error: Improper gap within a row. Report the bug.

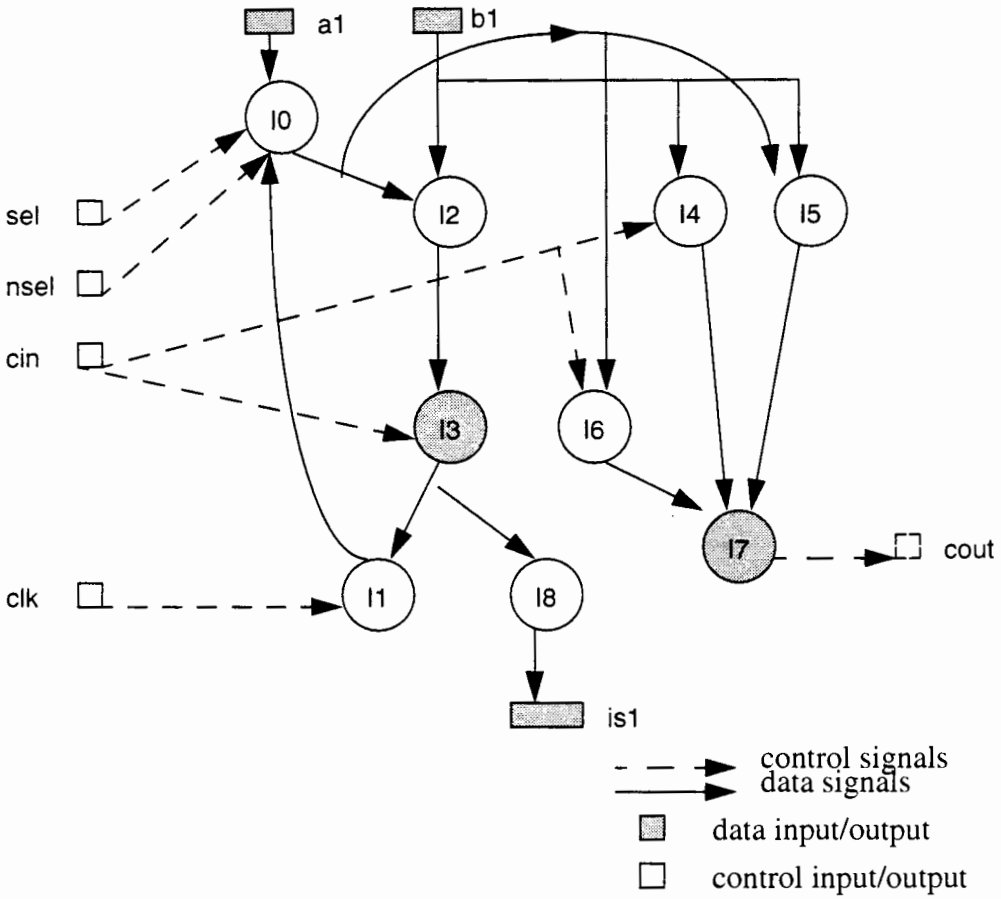
Unable to assign a feed: Shows the inefficiency of the feed assignment algorithm. Report the bug.

MST.. Internal error: Inconsistency while forming MST. Report the bug.

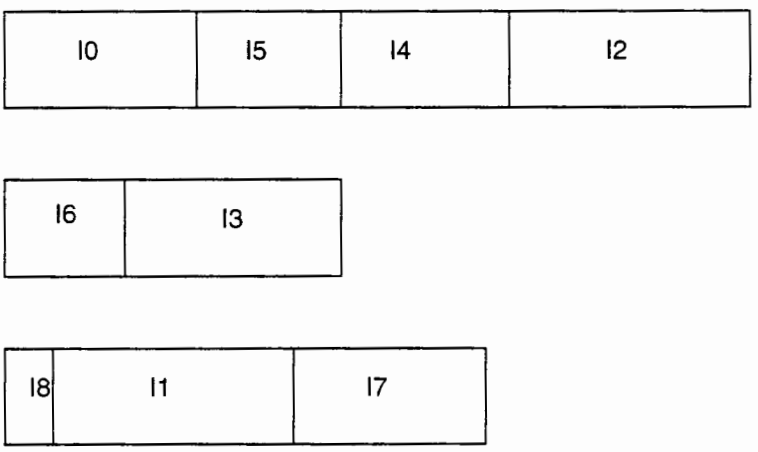
APPENDIX 2: TEST RUNS

```
// Example 1: 4-bit adder accumulator circuit. The layout for this circuit is Fig.6-1
module addaccu (a, b, sel, nsel, clk, cin, is, cout) ;
input[3:0] a,b ;
input sel,nsel,clk, cin ;
output[3:0] is ;
output cout ;
mid_slice bit0 (a[0],b[0],sel,nsel,clk,cin,is[0],cout0) ;
mid_slice bit1 (a[1],b[1],sel,nsel,clk,cout0,is[1],cout1) ;
mid_slice bit2 (a[2],b[2],sel,nsel,clk,cout1,is[2],cout2) ;
mid_slice bit3 (a[3],b[3],sel,nsel,clk,cout2,is[3],cout) ;
endmodule
// Bit-slice instantiated above
module mid_slice (a1, b1, sel, nsel, clk, cin, is1, cout) ;
input a1,b1,sel,nsel,clk,cin ;
output is1,cout ;
mx2_y i0 (a1,nsel,regout1,sel,mux1) ;
ms_y  i1 (s1,clk,regout1) ;
xr2_y i2 (b1,mux1,init1) ;
xr2_y i3 (cin,init1,s1) ;
a2_y  i4 (cin,b1,init4) ;
a2_y  i5 (mux1,b1,init5) ;
a2_y  i6 (cin,mux1,init3) ;
o3_y  i7 (init4,init5,init6,cout) ;
n1_y  i8 (s1,is1) ;
endmodule
```

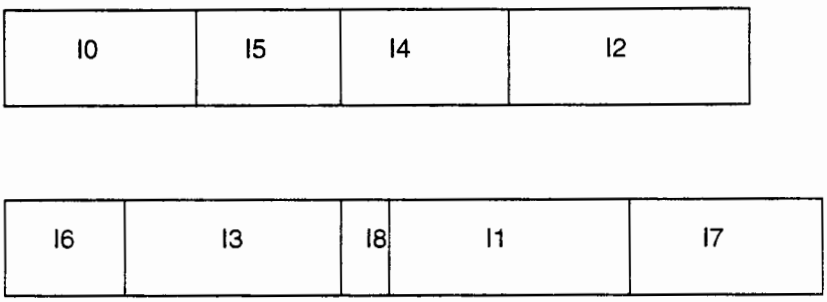
// The graph corresponding to this netlist, the placement before and after row merging and global router output are shown below.



Single bit-slice netlist represented as a graph



Placement before row merging



Placement after row merging

// Global router output

```
  Net A1 : ( id = 16)
Bot pin: at 3
Top pin(module): at 3
  Net B1 : ( id = 15)
Bot pin: at 75
Top pin(module): at 78
       Channel 1
  Net B1 : ( id = 15)
Top pin: at 51
Top pin: at 75
Top pin: at 105
  Net SEL : ( id = 14)
Top pin: at 27
Bot pin(module): at 81
  Net NSEL : ( id = 13)
Top pin: at 9
pin SEL side = 3 id = 4
Left pin(module):
  Net CIN : ( id = 11)
Bot pin: at 3
Top pin: at 69
Bot pin: at 39
pin CIN side = 3 id = 6
Left pin(module):
  Net REGOUT1 : ( id = 8)
Bot pin: at 153
Top pin: at 21
  Net MUX1 : ( id = 7)
Bot pin: at 9
Top pin: at 45
Top pin: at 123
Top pin: at 39
  Net INIT1 : ( id = 5)
```

```
Bot pin: at 57
Top pin: at 135
  Net INIT4 : ( id = 4)
Bot pin: at 159
Top pin: at 87
  Net INIT5 : ( id = 3)
Bot pin: at 165
Top pin: at 63
       Channel 2
  Net SEL : ( id = 14)
  Net CLK : ( id = 12)
Top pin: at 111
pin NSEL side = 3 id = 5
Left pin(module):
  Net CIN : ( id = 11)
Top pin: at 3
Top pin: at 39
  Net IS1 : ( id = 10)
Top pin: at 81
pin CLK side = 3 id = 7
Left pin(module):
  Net COUT : ( id = 9)
Top pin: at 177
pin COUT side = 4 id = 8
Right pin(module):
  Net S1 : ( id = 6)
Top pin: at 75
Top pin: at 69
Top pin: at 87
```

// Refer the code for description of the net ids and port sides, above.

```
// Example 2 : 12-bit slice datapath circuit
module whiffersullen (a, b, c, d, e, slew, reset, clock, duf, select,
tonset, takpul, xy);
input[11:0] a, b, c, d, e;
input slew, reset, clock, duf, select;
output[11:0] tonset, takpul, xy;
supply0 lo;
supply1 hi;
slice sl11 (a[11], b[11], c[11], d[11], e[11], tonset[11], takpul[11], xy[11],
slew, reset, clock, duf, select, lo,   a1110, lo,   b1110, hi,   c1110);
slice sl10 (a[10], b[10], c[10], d[10], e[10], tonset[10], takpul[10], xy[10],
slew, reset, clock, duf, select, a1110, a1009, b1110, b1009, c1110, c1009);
slice sl09 (a[09], b[09], c[09], d[09], e[09], tonset[09], takpul[09], xy[09],
slew, reset, clock, duf, select, a1009, a0908, b1009, b0908, c1009, c0908);
slice sl08 (a[08], b[08], c[08], d[08], e[08], tonset[08], takpul[08], xy[08],
slew, reset, clock, duf, select, a0908, a0807, b0908, b0807, c0908, c0807);
slice sl07 (a[07], b[07], c[07], d[07], e[07], tonset[07], takpul[07], xy[07],
slew, reset, clock, duf, select, a0807, a0706, b0807, b0706, c0807, c0706);
slice sl06 (a[06], b[06], c[06], d[06], e[06], tonset[06], takpul[06], xy[06],
slew, reset, clock, duf, select, a0706, a0605, b0706, b0605, c0706, c0605);
slice sl05 (a[05], b[05], c[05], d[05], e[05], tonset[05], takpul[05], xy[05],
slew, reset, clock, duf, select, a0605, a0504, b0605, b0504, c0605, c0504);
slice sl04 (a[04], b[04], c[04], d[04], e[04], tonset[04], takpul[04], xy[04],
slew, reset, clock, duf, select, a0504, a0403, b0504, b0403, c0504, c0403);
slice sl03 (a[03], b[03], c[03], d[03], e[03], tonset[03], takpul[03], xy[03],
slew, reset, clock, duf, select, a0403, a0302, b0403, b0302, c0403, c0302);
slice sl02 (a[02], b[02], c[02], d[02], e[02], tonset[02], takpul[02], xy[02],
slew, reset, clock, duf, select, a0302, a0201, b0302, b0201, c0302, c0201);
slice sl00 (a[00], b[00], c[00], d[00], e[00], tonset[00], takpul[00], xy[00],
slew, reset, clock, duf, select, a0100,     , b0100,     , c0100,     );
slice sl01 (a[01], b[01], c[01], d[01], e[01], tonset[01], takpul[01], xy[01],
```

```
slew, reset, clock, duf, select, a0201, a0100, b0201, b0100, c0201, c0100);
endmodule

// Bit-slice instantiated above
module slice (a, b, c, d, e, tonset, takpul, xy, slew, reset, clk, duf, sel,
acyin, acyout, icyin, icyout, rgin, rgout);
input a, b, c, d, e;
output tonset, takpul, xy;
input slew, reset, clk, duf, sel;
input acyin, icyin, rgin;
output acyout, icyout, rgout;
supply0 lo;
supply1 hi;
x592 adder (n1, n2, acyout, hi, a, lo, b, acyin);
x136 incr1 (n4,, icyout, n3, n2, icyin);
x420 xor1  (n6, n4, slew);
x352 dff1  (n7, n8, clk, n6, reset);
x416 mux1  (n9, n7, n8, duf);
x351 dff2  (rgout, n10, n14, n9);
x432 or1   (n12, n10, n11);
x416 mux2  (n3, n12, n16, sel);
x101 inv1  (tonset, n3);
x101 inv2  (n14, clk);
x422 major (n16, n15, c, d);
x318 lat1  (n18,, clk, n16);
x402 and1  (n11, n8, rgin);
x163 nand1 (takpul, n11, d, rgin);
x136 incr2 (, n15, icyout, d, n18, icyin);
x351 dff3  (n17,, clk, e);
x421 xor2  (xy, c, n17);
endmodule
```

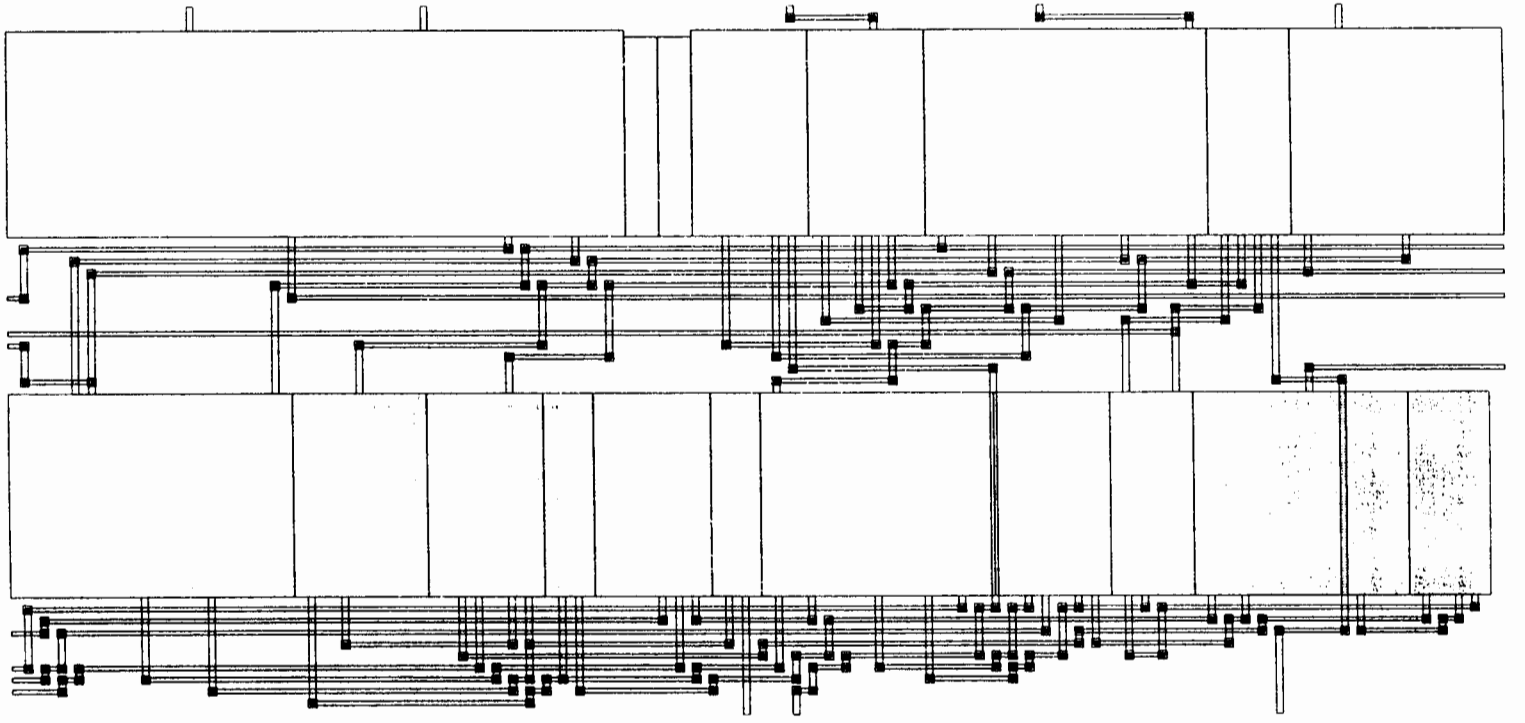


Fig.A-2-1 Single bit-slice datapath circuit layout

```
// Example 3 : 8-bit alu circuit
module alu (ainput, binput, dataout, aselect, binvert, bzero,
almode, logic, shift, clock);
input [7:0] ainput, binput;
output [7:0] dataout;
input aselect, binvert, bzero, almode;
input[2:0] logic;
input shift, clock;
supply0 lo;
alu_slice bit7 (ainput[7], binput[7], dataout[7], carry67,,
 shift67,       , aselect, binvert, bzero, almode, logic, shift, clock);
alu_slice bit6 (ainput[6], binput[6], dataout[6], carry56, carry67,
 shift56, shift67, aselect, binvert, bzero, almode, logic, shift, clock);
alu_slice bit5 (ainput[5], binput[5], dataout[5], carry45, carry56,
 shift45, shift56, aselect, binvert, bzero, almode, logic, shift, clock);
alu_slice bit4 (ainput[4], binput[4], dataout[4], carry34, carry45,
 shift34, shift45, aselect, binvert, bzero, almode, logic, shift, clock);
alu_slice bit3 (ainput[3], binput[3], dataout[3], carry23, carry34,
 shift23, shift34, aselect, binvert, bzero, almode, logic, shift, clock);
alu_slice bit2 (ainput[2], binput[2], dataout[2], carry12, carry23,
 shift12, shift23, aselect, binvert, bzero, almode, logic, shift, clock);
alu_slice bit1 (ainput[1], binput[1], dataout[1], carry01, carry12,
 shift01, shift12, aselect, binvert, bzero, almode, logic, shift, clock);
alu_slice bit0 (ainput[0], binput[0], dataout[0], binvert, carry01, lo,
 shift01, aselect, binvert, bzero, almode, logic, shift, clock);
endmodule

module alu_slice (ain, bin, out, cin, cout, shftin, shftout, asel,
binvert, bzero, almode, logic, shift, clk);
input ain, bin, cin, shftin, asel, binvert, bzero, almode, shift, clk;
output out, cout, shftout;
```

```
input[2:0] logic;

mux2 aselect (aoperand, ain, out, asel);
xor2 bselect (bchoice, bin, binvert);
not1 zeroinv (notzero, bzero);
and2 zselect (boperand, bchoice, notzero);
mux2 logicsel (ctoaddr, cin, logic[1], almode);
add2 adder (sum, cout, aoperand, boperand, ctoaddr);
xor2 negandor (carry, cout, logic[0]);
and2 agate (addmuxsel, logic[2], almode);
mux2 addmux (shftout, sum, carry, addmuxsel);
mux2 shiftmux (shftmuxout, shftout, shftin, shift);
dff  register (out,, clk, shftmuxout);
endmodule
```

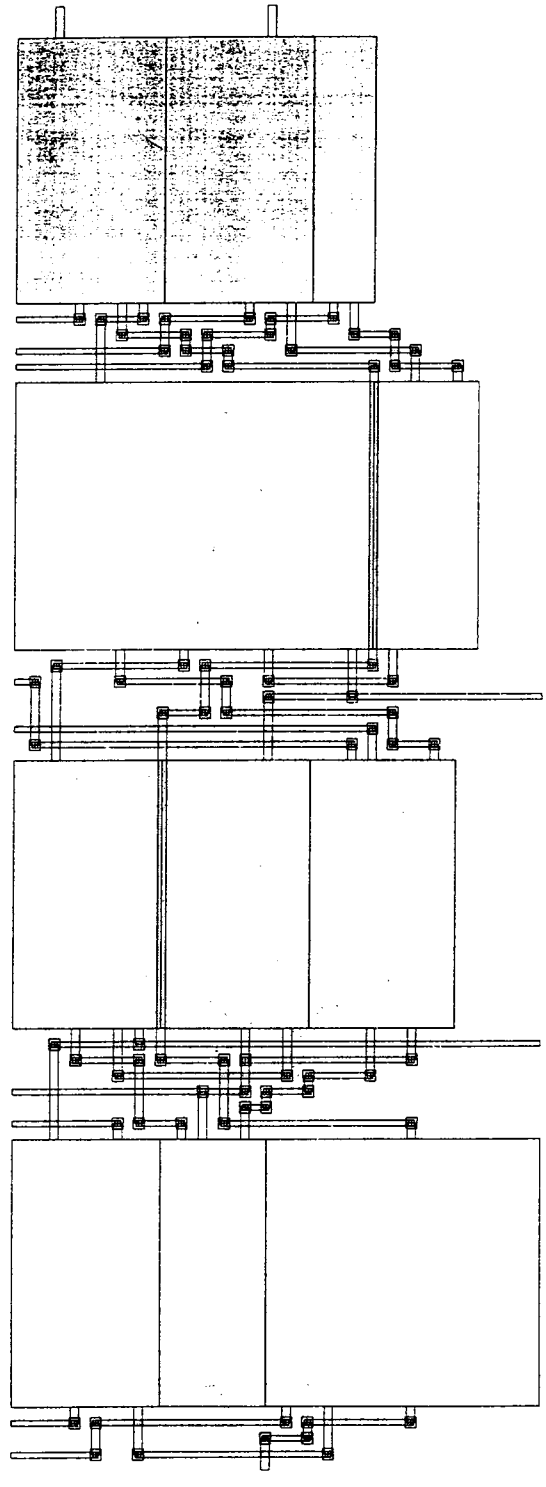


Fig.A-2-2 Single bit-slice *alu* circuit layout

```
// Example 4 : 16 stage, 1-bit FIFO circuit
module fifo (shram0, ra0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16) ;
input  shram0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16;
output  ra0 ;
fifo0 slice0( shram0, ra0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16) ;
endmodule
module fifo0 (shram0, ra0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16) ;
input  shram0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16;
output  ra0 ;
ms_y  m_16_0 (shram0, ck16, s016);
ms_y  m_15_0 (s016, ck15, s015);
ms_y  m_14_0 (s015, ck14, s014);
ms_y  m_13_0 (s014, ck13, s013);
ms_y  m_12_0 (s013, ck12, s012);
ms_y  m_11_0 (s012, ck11, s011);
ms_y  m_10_0 (s011, ck10, s010);
ms_y  m_9_0 (s010, ck9, s09);
ms_y  m_8_0 (s09, ck8, s08);
ms_y  m_7_0 (s08, ck7, s07);
ms_y  m_6_0 (s07, ck6, s06);
ms_y  m_5_0 (s06, ck5, s05);
ms_y  m_4_0 (s05, ck4, s04);
ms_y  m_3_0 (s04, ck3, s03);
ms_y  m_2_0 (s03, ck2, s02);
ms_y  m_1_0 (s02, ck1, ra0);
endmodule
```

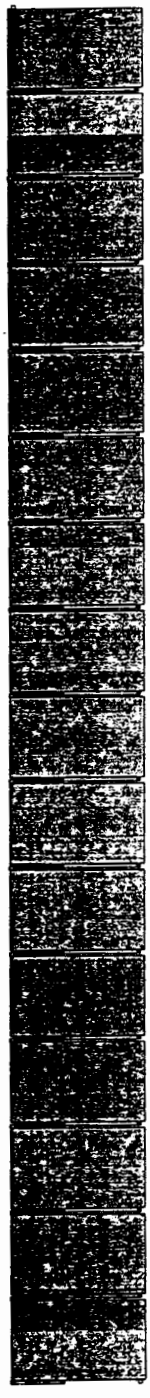


FIg.A-2-3 Single bit-slice *fifo* circuit layout

```
// 1 bit ram circuit
module ram (shram0, ra0, rb0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16,
        a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16,
        b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16) ;
input  shram0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16,
        a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16,
        b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16 ;
output  ra0, rb0 ;
ram0 slice0( a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15,
        a16, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14,
        b15, b16, shram0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16,ra0,rb0) ;
endmodule
module ram0( a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15,
        a16, b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14,
        b15, b16, shram0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16,ra0,rb0) ;
input  shram0, ck1, ck2, ck3, ck4, ck5, ck6,
        ck7, ck8, ck9, ck10, ck11, ck12, ck13, ck14, ck15, ck16,
        a1, a2, a3, a4, a5, a6, a7, a8, a9, a10, a11, a12, a13, a14, a15, a16,
        b1, b2, b3, b4, b5, b6, b7, b8, b9, b10, b11, b12, b13, b14, b15, b16 ;
output  ra0, rb0 ;
ms_y  m_16_0 (shram0, ck16, s016);
ms_y  m_15_0 (shram0, ck15, s015);
ms_y  m_14_0 (shram0, ck14, s014);
ms_y  m_13_0 (shram0, ck13, s013);
ms_y  m_12_0 (shram0, ck12, s012);
ms_y  m_11_0 (shram0, ck11, s011);
ms_y  m_10_0 (shram0, ck10, s010);
```

```
ms_y  m_9_0 (shram0, ck9, s09);
ms_y  m_8_0 (shram0, ck8, s08);
ms_y  m_7_0 (shram0, ck7, s07);
ms_y  m_6_0 (shram0, ck6, s06);
ms_y  m_5_0 (shram0, ck5, s05);
ms_y  m_4_0 (shram0, ck4, s04);
ms_y  m_3_0 (shram0, ck3, s03);
ms_y  m_2_0 (shram0, ck2, s02);
ms_y  m_1_0 (shram0, ck1, s01);
na2_y  am016 (a16, a016s, s016);
na2_y  am015 (a15, a015s, s015);
na2_y  am014 (a14, a014s, s014);
na2_y  am013 (a13, a013s, s013);
na2_y  am012 (a12, a012s, s012);
na2_y  am011 (a11, a011s, s011);
na2_y  am010 (a10, a010s, s010);
na2_y  am09 (a9, a09s, s09);
na2_y  am08 (a8, a08s, s08);
na2_y  am07 (a7, a07s, s07);
na2_y  am06 (a6, a06s, s06);
na2_y  am05 (a5, a05s, s05);
na2_y  am04 (a4, a04s, s04);
na2_y  am03 (a3, a03s, s03);
na2_y  am02 (a2, a02s, s02);
na2_y  am01 (a1, a01s, s01);
a4_y  oa410 (oa410s,a016s, a015s, a014s, a013s);
a4_y  oa420 (oa420s,a012s, a011s, a010s, a09s);
a4_y  oa430 (oa430s,a08s, a07s, a06s, a05s);
a4_y  oa440 (oa440s,a04s, a03s, a02s, a01s);
na4_y  oa450 (oa410s, oa420s, oa430s, oa440s, ra0);
na2_y  bm016 (b16, b016s, s016);
na2_y  bm015 (b15, b015s, s015);
```

```
na2_y  bm014 (b14, b014s, s014);
na2_y  bm013 (b13, b013s, s013);
na2_y  bm012 (b12, b012s, s012);
na2_y  bm011 (b11, b011s, s011);
na2_y  bm010 (b10, b010s, s010);
na2_y  bm09 (b9, b09s, s09);
na2_y  bm08 (b8, b08s, s08);
na2_y  bm07 (b7, b07s, s07);
na2_y  bm06 (b6, b06s, s06);
na2_y  bm05 (b5, b05s, s05);
na2_y  bm04 (b4, b04s, s04);
na2_y  bm03 (b3, b03s, s03);
na2_y  bm02 (b2, b02s, s02);
na2_y  bm01 (b1, b01s, s01);
a4_y  ob410 (ob410s,b016s, b015s, b014s, b013s);
a4_y  ob420 (ob420s,b012s, b011s, b010s, b09s);
a4_y  ob430 (ob430s,b08s, b07s, b06s, b05s);
a4_y  ob440 (ob440s,b04s, b03s, b02s, b01s);
na4_y  ob450 (ob410s, ob420s, ob430s, ob440s, rb0);
endmodule
```

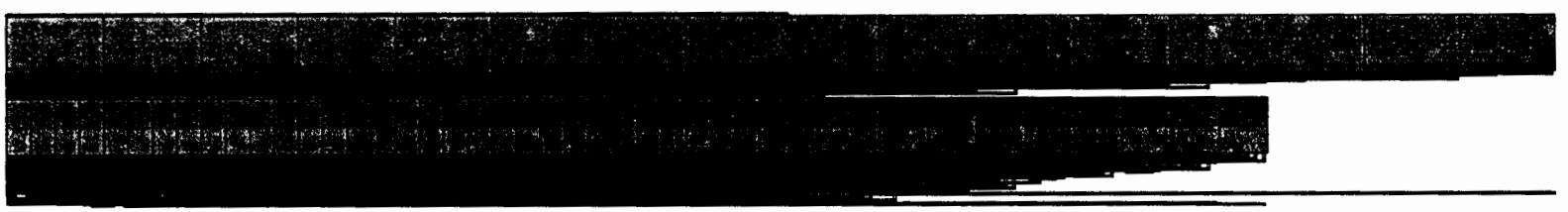


FIg.A-2-4 Single bit-slice *ram* circuit layout