INTEGRATION OF BEHAVIORAL AND LAYOUT SYNTHESIS: A CHIP SYNTHESIS APPROACH

Allen C-H Wu
Dissertation
University of California
Irvine, California 92717

April 20, 1992

Technical Report No. 92–36

Integration of Behavioral and Layout Synthesis: A Chip Synthesis Approach

Allen C-H. Wu

Technical Report #92-36
April 20, 1992

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

Chip synthesis deals with the transformation of a behavioral description into a fabricated chip. Typically, chip synthesis is carried out in three stages: behavioral, logic/sequential and layout synthesis. Since chip synthesis involves a multi-level synthesis task, integration and coordination of tasks for all levels of synthesis is the essential issue.

This dissertation addresses a chip synthesis paradigm and describes the key issues with regard to the integration of behavioral and layout synthesis for chip design. In order to successfully integrate all tasks in the chip synthesis process, a finite-state machine with a datapath (FSMD) design model and a sliced-layout architecture have been developed for chip synthesis. Using the sliced-layout architecture, a partitioning-based layout synthesis method and system have been developed to synthesize layout from generalized register-transfer (RT) netlists. In addition, based on the FSMD and the sliced-layout architecture, area and timing models are developed for behavioral synthesis. To incorporate layout information into behavioral synthesis, a unified representation is developed for behavioral synthesis. Using the unified representation and layout model, a layout-driven unit-binding approach is presented. Several sets of experiments were performed to validate the proposed approaches including the layout-synthesis method, the layout model and the layout-driven unit-binding task.

UNIVERSITY OF CALIFORNIA

IRVINE

Integration of Behavioral and Layout Synthesis: A Chip Synthesis Approach

DISSERTATION

submitted in partial satisfaction of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in Information and Computer Science

by

Chung-Hao (Allen) Wu

Dissertation Committee:

Professor Daniel D. Gajski, Chair

Professor Nikil D. Dutt

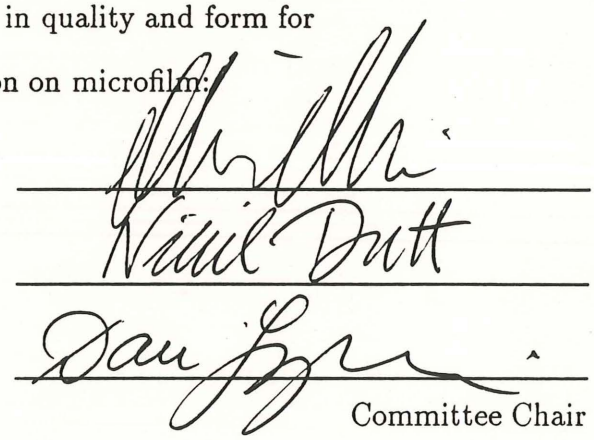
Professor Fadi J. Kurdahi

1992

The dissertation of Chung-Hao (Allen) Wu is approved,

and is acceptable in quality and form for

publication on microfilm:



Committee Chair

University of California, Irvine

1992

Dedication

To my grandparents

Contents

List of Figures

Acknowledgments

First and foremost, I would like to thank my advisor Professor Daniel D. Gajski for his guidance, faith and support throughout my study. I would also like to thank the other members of my doctoral committee, Professor Nikil D. Dutt and Professor Fadi J. Kurdahi.

I would also like to thank Professor Steve Lin to encourage me to continue pursuing my PhD. Otherwise, I might be still staying in Tucson, Arizona and enjoying my nine-to-five job plus hiking and fishing on the weekend.

I would like to acknowledge my colleagues in CADLAB. They include Roger Ang, Viraphol Chaiyakul, Gwo Dong Chen, Jim Frankin, Jie Gong, Tedd Hadley, Pradip Jha, Kenichi Kanehara, Young Kim, Jim Kipps, Joe Lis, Sanjiv Narayan, Loganath Ramachandran, Elke Rundensteiner, Frank Vahid and Nels Vander Zanden. I thank you for all your valuable discussion, assistance and friendship. I would also like to thank Bob Larsen of Rockwell International for his valuable discussions and suggestions. I like to extend my gratitude to my partner Viraphol Chaiyakul for his invaluable contribution to the quality-measure research.

Finally, I would like to thank the support I received from my family.

This work was supported by NSF grant ♯MIP-8922851 and California MICRO grant ♯90-046. I am grateful for their support.

Curriculum Vitae

Chung-Hao (Allen) Wu

EDUCATION:

- 1992: Ph.D. in Information and Computer Science at the University of California, Irvine.
  (Dissertation Title: "Integration of Behavioral and Layout Synthesis: A Chip Synthesis Approach").
- 1985: M.S. in Electrical and Computer Engineering at the University of Arizona, Tucson.
  (Thesis Title: "A Microprocessor–Based Ultrasonic System for Measuring Bladder Volumes").
- 1983: B.S. in Electronics Engineering at the National Taiwan Institute of Technology, Taipei, Taiwan.
- 1977: Diploma in Electronics Engineering at the Min-Hsin Institute of Technology, Hsinchu, Taiwan.

EXPERIENCE:

- 1988–1992: Research Assistant, Information and Computer Science, University of California, Irvine.
- 1986–1988: Electrical Engineer, Physiology Department, University Medical Center, University of Arizona.
- 1983–1985: Research Assistant, Electrical and Computer Engineering, University of Arizona, Tucson, Arizona.
- 1977–1980: Design Engineer, Dahsen Electronic Co., Taipei, Taiwan.

PUBLICATIONS:

- Conference and Journal Papers:

  1. N. Vander Zanden, Allen C-H Wu, and D. D. Gajski, "Technology Mapping with Layout Constraints," *Int Sym. on VLSI Technology, Systems and Applications,* Taiwan, 1989.

2. Allen C-H Wu, N. Vander Zanden, and D. D. Gajski, "An Algorithm for Transistor Sizing in CMOS Circuits," *The European Conf. on Design Automation*, Glasgow, Scotland, 1990.
3. Allen C-H Wu, G. D. Chen,and D. D. Gajski, "Silicon Compilation from Register-Transfer Schematics," *Int Sym. Circuits and Systems*, 1990.
4. Allen C-H Wu and D. D. Gajski, "Partitioning Algorithms for Layout Synthesis from Register-Transfer Netlists," *Int Conf. Computer-Aided Design*, 1990.
5. Allen C-H Wu and D. D. Gajski, "Glue-Logic Partitioning for Floorplans with A Rectilinear Datapath," *The European Conf. on Design Automation*, 1991.
6. N. Vander Zanden, Allen C-H Wu, and D. D. Gajski, "Layout Synthesis for Custom Layout," *The European Conf. on Design Automation*, 1991.
7. Allen C-H Wu, G. D. Chen and D. D. Gajski, "Evaluation Driven Layout Synthesis," *Int Sym. on VLSI Technology, Systems and Applications*, Taiwan, 1991.
8. Allen C-H Wu, V. Chaiyakul, and D. D. Gajski, "Layout Models for High-Level Synthesis," *Int Conf. Computer-Aided Design*, 1991.
9. Lawrence L. Larmore, D. D. Gajski, and Allen C-H Wu, "Layout Placement for Sliced Architecture," *IEEE Trans. on Computer-Aided Design*, vol. 11, no. 1, January, 1992.
10. Allen C-H Wu and D. D. Gajski, "Partitioning Algorithms for Layout Synthesis from Register-Transfer Netlists," *IEEE Trans. on Computer-Aided Design*, vol. 11, no. 3, March, 1992.
11. D. D. Gajski, N. Dutt, C-H Wu and Y-L Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.
12. D. D. Gajski, Allen C-H. Wu, and Viraphol Chaiyakul, "Layout Synthesis and Layout Models for Synthesis," *SASIMI'92*, 1992.
13. Viraphol Chaiyakul, Allen C-H Wu and D. D. Gajski, "Timing Models for High Level Synthesis," *EURO-DAC'92*, 1992.

- Technical Reports:

1. Allen C-H Wu, N. Vander Zanden, and D. D. Gajski, "An Algorithm for Transistor Sizing in CMOS Circuits," Tech. Report 89-04, ICS Dept., U.C. Irvine, 1989.
2. N. Vander Zanden, Allen C-H Wu, and D. D. Gajski, "Performance Optimization in Layout Driven Synthesis," Tech. Report 89-21, ICS Dept., U.C. Irvine, 1989.

3. Allen C-H Wu and D. D. Gajski, "SLAM: An Automated Structure to Layout Synthesis System," Tech. Report 89-40, ICS Dept., U.C. Irvine, 1989.
4. D. D. Gajski, Joseph Lis, N. Vander Zanden, and Allen C-H Wu, "Synthesis from VHDL: Rockwell-Counter Case Study," Tech. Report 90-09, ICS Dept., U.C. Irvine, 1990.
5. Allen C-H Wu and D. D. Gajski, "A New Partitioning Approach for Layout Synthesis from Register-Transfer Netlists," Tech. Report 90-10, ICS Dept., U.C. Irvine, 1990
6. Allen C-H Wu "Survey of Partitioning Techniques in Silicon Compilation," Tech. Report 91-15, ICS Dept., U.C. Irvine, 1991.
7. Allen C-H Wu, Viraphol Chaiyakul and D. D. Gajski, "Back-Annotation for Interactive Data Path Synthesis," Tech. Report 91-29, ICS Dept., U.C. Irvine, 1991.
8. Allen C-H Wu and D. D. Gajski, "Layout-Driven Allocation for High-Level Synthesis," Tech. Report 91-30, ICS Dept., U.C. Irvine, 1991.
9. Allen C-H Wu, Viraphol Chaiyakul and D. D. Gajski, "Layout Area Models for High Level Synthesis," Tech. Report 91-31, ICS Dept., U.C. Irvine, 1991.
10. Allen C-H Wu, Joe Lis and D. D. Gajski, "Partitioning-Based Algorithm for Pipelined Scheduling and Module Assignment," Tech. Report 91-32, ICS Dept., U.C. Irvine, 1991.
11. Viraphol Chaiyakul, Allen C-H Wu and D. D. Gajski, "Timing Models for High Level Synthesis," Tech. Report 91-70, ICS Dept., U.C. Irvine, 1991.
12. Tedd S. Hadley, Allen C-H Wu and D. D. Gajski, "An Efficient Multi-View Design Model for Real-Time Interactive Synthesis," Tech. Report 92-35, ICS Dept., U.C. Irvine, 1992.

Professional Activities:

- Reviewer for Design Automation Conference (DAC).
- Member IEEE, ACM.

Abstract of the Dissertation

Integration of Behavioral and Layout Synthesis: A Chip Synthesis Approach

by
Chung-Hao (Allen) Wu
Doctor of Philosophy in Information and Computer Science
University of California, Irvine, 1992
Professor Daniel D. Gajski, Chair

Chip synthesis deals with the transformation of a behavioral description into a fabricated chip. Typically, chip synthesis is carried out in three stages: behavioral, logic/sequential and layout synthesis. Since chip synthesis involves a multi-level synthesis task, integration and coordination of tasks for all levels of synthesis is the essential issue.

This dissertation addresses a chip synthesis paradigm and describes the key issues with regard to the integration of behavioral and layout synthesis for chip design. In order to successfully integrate all tasks in the chip synthesis process, a finite-state machine with a datapath (FSMD) design model and a sliced-layout architecture have been developed for chip synthesis. Using the sliced-layout architecture, a partitioning-based layout synthesis method and system have been developed to synthesize layout from generalized register-transfer (RT) netlists. In addition, based on the FSMD and the sliced-layout architecture, area and timing models are developed for behavioral synthesis. To incorporate layout information into behavioral synthesis, a unified representation is developed for behavioral synthesis.

Using the unified representation and layout model, a layout-driven unit-binding approach is presented. Several sets of experiments were performed to validate the proposed approaches including the layout-synthesis method, the layout model and the layout-driven unit-binding task.

Chapter 1

Introduction

Microelectronic technology has advanced tremendously in the past decade. In the early 1990s, VLSI technology reached the chip density with up to one million transistors. Chips of such complexity are very difficult to design by human designers. In order to successfully exploit new VLSI technology, design automation at the chip level is needed to facilitate the chip design process and to shorten the time-to-market cycle.

1.1 Chip Synthesis Overview

Chip synthesis deals with the transformation of a behavioral description into a fabricated chip. Typically, chip synthesis is carried out in three stages: behavioral synthesis, logic/sequential synthesis and layout synthesis, as shown in Figure 1.1.

Behavioral synthesis transforms a given behavioral description into a structural description. Figure 1.2 shows a generic behavioral-synthesis system for chip design [GDWL92]. The system consists of a *compiler*, a *scheduler*, a number of *allocators*, a *module selector*, a *component database (CDB)*, a *technology mapper* and a *microarchitecture optimizer*. The *compiler* compiles the input behav-

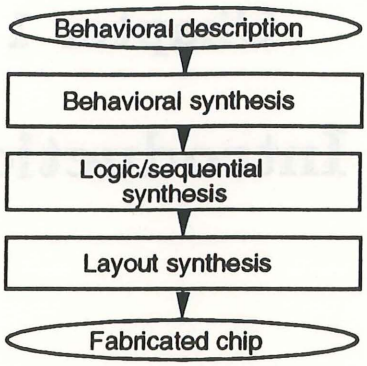


Figure 1.1: Chip synthesis.

ioral description into a design representation such as a control/data flow graph (*CDFG*). The *Scheduler* assigns operations into control steps. *Storage*, *Functional* and *Interconnect* unit allocators bind variables to registers and memories, operators to functional units and data transfers to buses. The *Module selector* determines the RT components to be used in the design, whereas the *Storage merger* groups registers into register files or groups a small memory set into a large memory. The component database (*CDB*) manages the RT component library and answers queries of component information in the synthesis process. The *Technology mapper* assigns real components from the *CDB* to the generic components in the structural description. The *Microarchitecture optimizer* improves the area and delay of the structural design.

Logic/sequential synthesis transforms a structural description generated from a behavioral-synthesis system into a hardware description for layout synthesis. A structural description usually includes a control-state table, a datapath RT netlist (or a set of Boolean expressions), an interface specification with timing or waveform diagrams, memory sizes and I/O ports information. Since each component

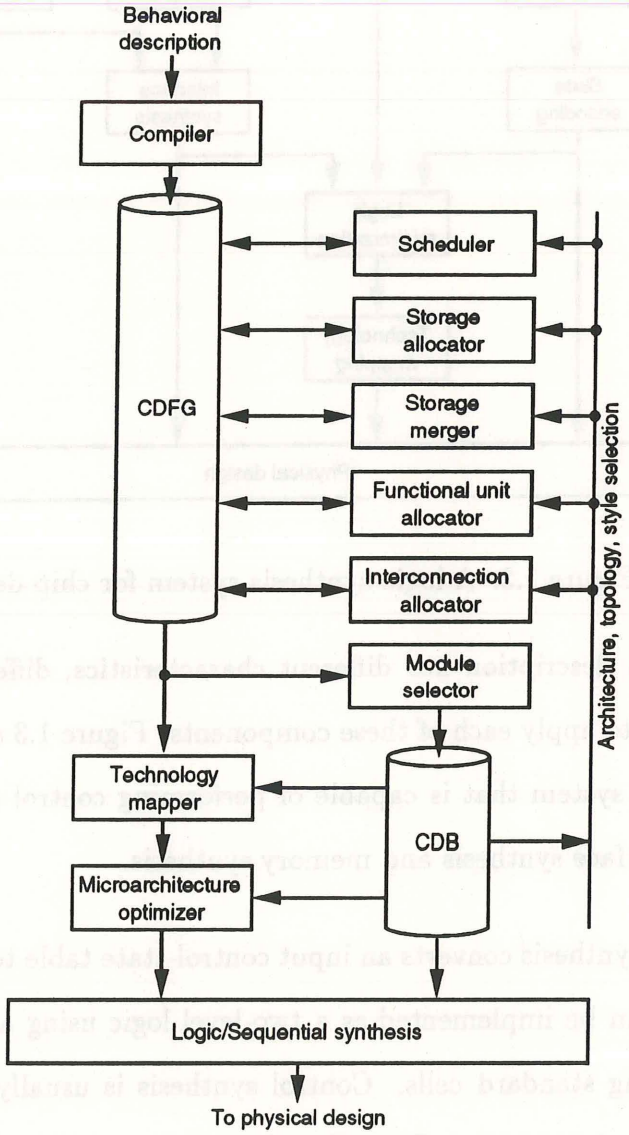


Figure 1.2: A behavioral-synthesis system for chip design.

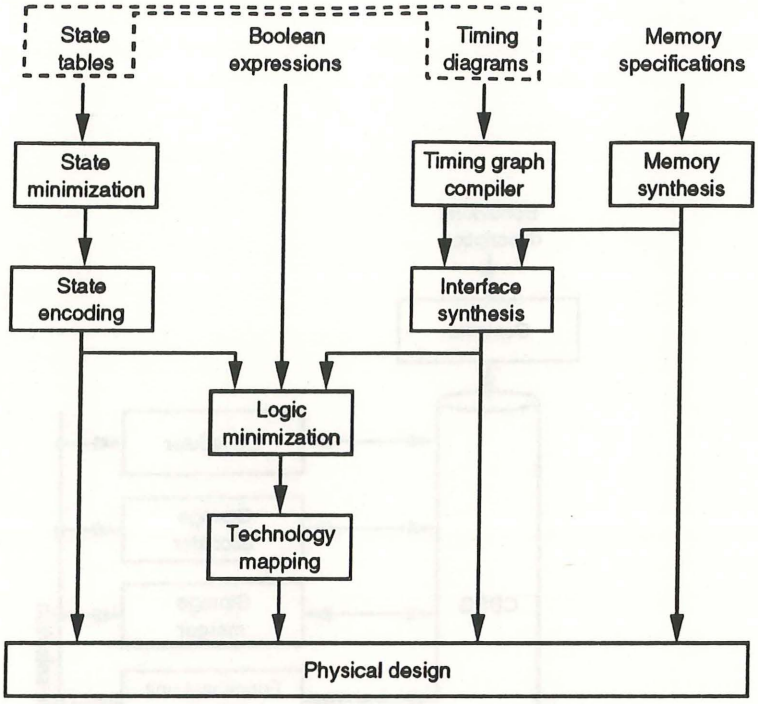


Figure 1.3: A logic-synthesis system for chip design.

in a structural description has different characteristics, different synthesis techniques require to apply each of these components. Figure 1.3 shows a hypothetical logic-synthesis system that is capable of performing control synthesis, functional synthesis, interface synthesis and memory synthesis.

Control synthesis converts an input control-state table to a control unit. The control unit can be implemented as a two-level logic using a PLA or a multiple-level logic using standard cells. Control synthesis is usually carried out in four stages: *State minimization*, *State Encoding*, *Logic minimization* and *Technology mapping* (Figure 1.3). State minimization reduces the number of states in the control-state table by eliminating redundant states. State encoding assigns binary codes to symbolic states. Logic minimization reduces the cost and improves the delay of the control logic. Technology mapping maps the control unit to a target

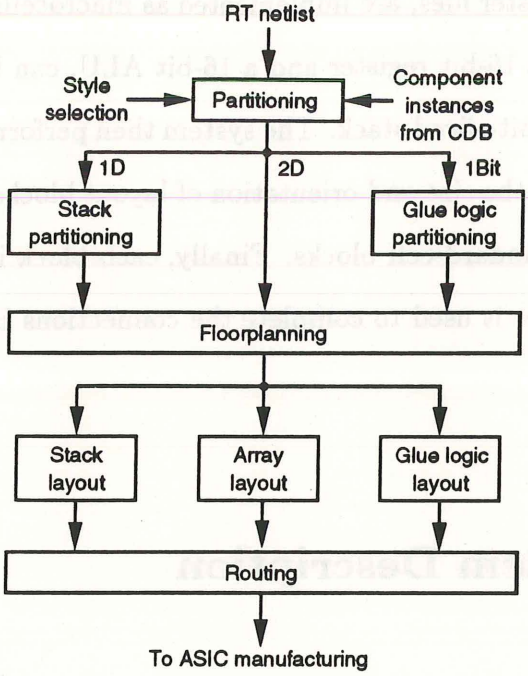


Figure 1.4: A layout-synthesis system for chip design.

implementation such as a PLA or random gates. Similarly, Boolean expressions for a functional unit can be synthesized using the logic minimization and technology mapping stages. Interface synthesis converts a given timing diagram into a set of random gates, latches and flip-flops to satisfy the given communication-protocol requirement. Memory synthesis generates a memory description to satisfy the given requirements such as size, access time and access protocol.

Layout synthesis transforms a RT netlist generated by a behavioral-synthesis system into a fabrication description of the chip. Figure 1.4 shows a representation layout-synthesis system for chip design. This system first partitions RT components into two groups, *Stack* and *Glue logic*, according to their style, dimension and connectivity. For instance, two-dimensional components, such as multipliers,

memories and register files, are implemented as macrocells. One-dimensional components, such as a 16-bit register and a 16-bit ALU, can be grouped together and implemented as a bit-sliced stack. The system then performs a floorplanning procedure to determine the size and orientation of layout blocks such as bit-sliced stack, macrocells and standard-cell blocks. Finally, each block is laid out independently and a global router is used to complete the connections between blocks.

1.2 Problem Description

Since chip synthesis is a multi-level synthesis task, integration and coordination of tasks for all levels of synthesis is the essential issue in chip synthesis. In order to successfully integrate all tasks in chip synthesis, it has the following primary requirements: a consistent and well-defined target architecture for all synthesis levels, a RT-based layout-synthesis method, a unified design model for all synthesis levels and the technique of incorporating layout synthesis into behavioral synthesis.

It is difficult to develop a general purpose synthesis system that will produce high-quality results for a variety of target applications. Thus, existing synthesis systems have focused their capabilities on a selected target architecture or application to reduce the complexity of the design process. A consistent and well-defined target architecture is required in chip synthesis. This allows the tools in different synthesis-level to use a consistent architecture and thus the coordination between different synthesis tasks can be established. Furthermore, to carry out the layout

generation in chip synthesis, a layout-synthesis method is needed to transform a RT netlist into a chip layout.

In order to integrate the tools in all synthesis levels, a unified design model is needed. Using such a model, synthesis tools can retrieve layout information in any design stage and use this information to guide the design process. Typically, there are two approaches to incorporate layout information into behavioral-level synthesis. The first approach uses some simple layout model to estimate design quality. The drawback of using simplified layout models is low accuracy of obtained estimates. The second approach feeds back the actual layout information by performing layout synthesis. This approach does provide accurate estimates; however, it is too slow for nontrivial designs. To provide a fast and accurate quality measure in chip synthesis, an accurate layout model that includes area and timing models is required. Furthermore, the technique that combines all models, including layout and design models, for chip synthesis is needed.

1.3 Contributions

This dissertation describes a number of important issues involved in chip synthesis (Figure 1.5). A design model called finite-state machine with a datapath (FSMD) and a layout architecture called sliced-layout architecture are presented. Using the sliced-layout architecture, a partitioning-based layout-synthesis method and system will be discussed (Figure 1.5(a)). Based on the FSMD and the sliced-layout architecture, area and timing models are developed for behavioral synthesis (Figure 1.5(b)). To incorporate layout information into behavioral synthesis, a

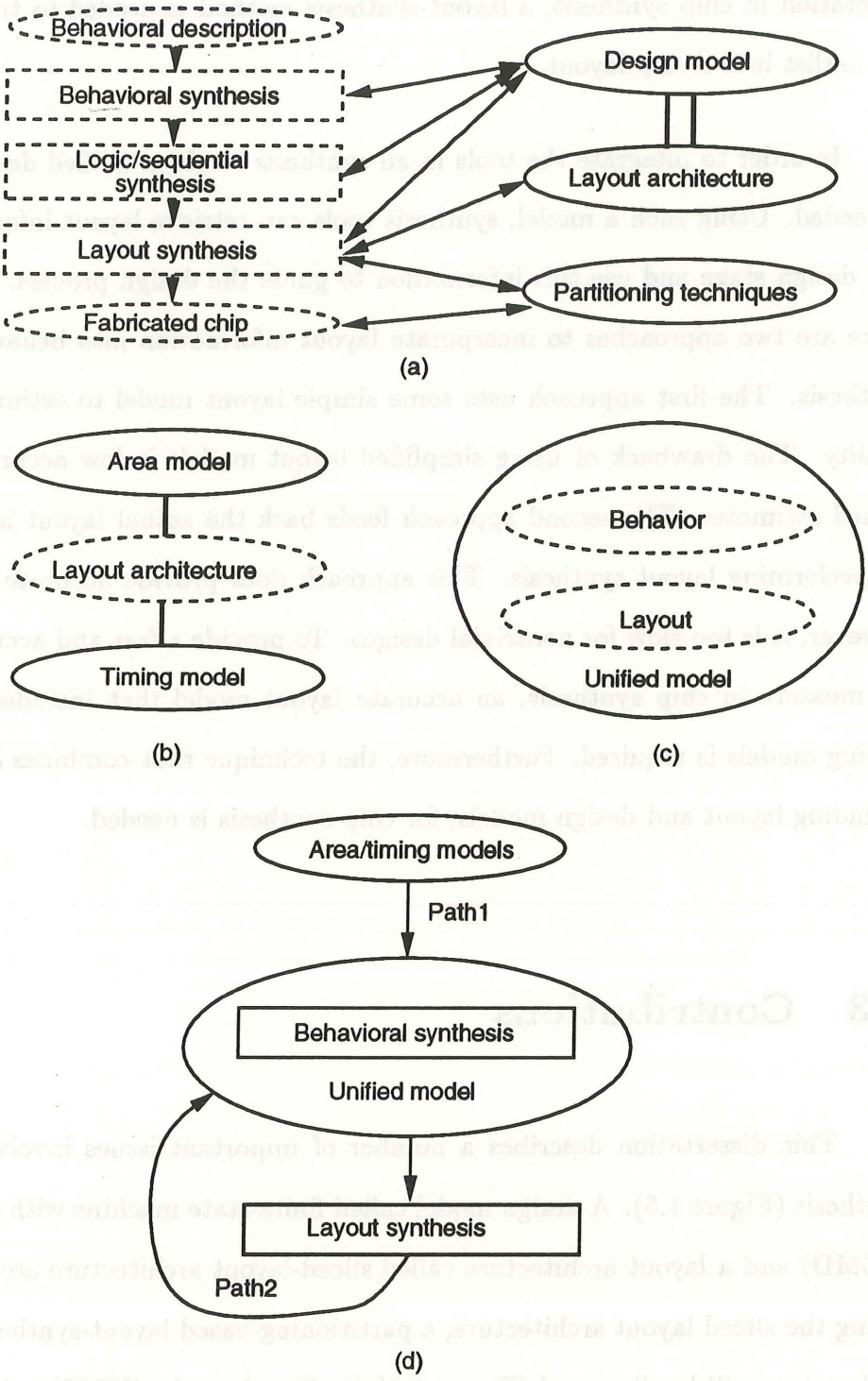


Figure 1.5: The essential issues in chip synthesis: (a) the design model and layout architecture, (b) the area/timing model, (c) a unified design model, (d) layout-driven behavioral synthesis.

unified design model is developed for behavioral synthesis (Figure 1.5(c)). Two different layout-driven unit-binding approaches, using feedback layout information (*Path2*) and using layout model (*Path1*), are presented (Figure 1.5(d)). The novel contributions of this work are described below.

- **The sliced-layout architecture.** We have modelled chips using the FSMD model. To realize the FSMD model, we developed a sliced-layout architecture for compilation of generalized RT netlists into layout for CMOS technology. This sliced-layout architecture combines over-the-cell routing and bit-sliced stack folding to produce flexible and high-density layout.
- **Partitioning-based layout synthesis for RT netlists.** Using the sliced-layout architecture, we developed a partitioning approach for layout synthesis from RT netlists that exploits the regularity of RT components. Three partitioning algorithms, component partitioning, stack partitioning and glue-logic partitioning, were developed to generate the chip floorplan. Using these algorithms, a layout-synthesis system (SLAM) was developed to bridge the gap between behavioral and layout synthesis tools.
- **Area and timing models for behavioral synthesis.** To obtain more realistic quality measures for behavioral synthesis, a layout model, including area and timing models, is developed. This layout model takes into account most technology factors such as layout architectures and technology mapping, and thus provides more accurate estimates than previous proposed models.
- **A unified design representation.** A unified model was developed to bridge the gap between behavioral and layout synthesis. This model provides three important features for incorporating layout information into behavioral synthesis. First, this model encapsulates both the structure and the

behavior of the design. Second, this model provides a structural hierarchy of chips. Third, this model provides a unified behavior/structure view of the design that is well suited for interactive synthesis. Using this proposed unified model, synthesis tools can retrieve layout information at any design level.

- **Incorporation of layout information into behavioral synthesis.** In order to incorporate layout information into behavioral synthesis, two methods, using the layout model and using the feedback layout information, were developed for behavioral synthesis. To provide a fast and accurate area measures in the datapath design process, we developed a new approach that uses our proposed layout model for datapath optimization. In order to use the area model, we model the datapath as a graph representation that explicitly reflects the datapath floorplan. To take advantage of layout information we formulate datapath binding as a graph partitioning problem. Contrary to the other datapath optimization algorithms which minimize the number and size of registers and muxes, our algorithm evaluates layout-area quality during datapath optimization. Our approach provides faster and more accurate area quality measures for datapath optimization than previous approaches.

1.4 Thesis Overview

This dissertation is organized as follows. Chapter 2 surveys related work in behavioral and layout synthesis and their applications. Chapter 3 presents the target architecture. Chapter 4 describes a layout-synthesis system for layout generation from generalized RT netlist. Chapter 5 presents the area and timing quality measures in behavioral synthesis. Chapter 6 describes a unified model for behavioral synthesis. Chapter 7 presents the layout-driven unit-binding approach. Chapter 8 summarizes the accomplishments of this research and outlines future work.

Chapter 2

Related Work

This chapter surveys related work in the area of behavioral and layout synthesis in particular the methods of incorporating layout information into behavioral synthesis and the linkage between layout and synthesis tools.

In this chapter, Sections 2.1, 2.2, and 2.3 describe three different approaches to incorporate layout information into behavioral synthesis. BUD [McFa86, McKo90] uses a layout model and a floorplanner to provide physical information for module selection in behavioral synthesis. Chippe [BrGa90] and Fasolt [Knap89] use a feedback-driven technique to guide design tasks in behavioral synthesis. Section 2.4 describes LASSIE [TrDi89] that is a layout-synthesis tool used to generate layout for behavioral-synthesis tools. Sections 2.5 and 2.6 describe two complete CAD systems for chip design: Cathedral [DRSC86, NGCD91] focuses on DSP chip design and LAGER [RaPB85, Shun91] focuses on algorithm-specific chip design. Finally, Section 2.7 summarizes the previous approaches.

2.1 BUD

BUD [McFa86, McKo90] introduces physical-level information to guide behavioral-synthesis tasks, in particular, for the module selection. It uses a hierarchical clustering technique to partition operations in the control/data flow graph into clusters based on their similarity and concurrency. BUD first forms a hierarchical clustering tree. In such a tree, the leaves represent the operations to be clustered, and the number of non-leaf nodes between two leaf nodes defines the relative similarity between these two operations.

The cluster tree guides the search of the design space. A series of different module (i.e., functional unit) sets is generated by starting with a cut-line at the root and by moving the cut-line toward the leaves. Any cut across the tree partitions the tree into a set of subtrees, each of which represents a cluster. For each cut-line a different module set is formed and the resulting design is evaluated. Thus, the first design has all operations in the same cluster, sharing the same tightly coupled datapath. The second is made up of two clusters that have the greatest distance between them, and so on. This process continues until a stopping criterion, such as the lack of improvement in the design for a predetermined number of module sets, is reached.

Each time a new unit set is selected, the design is evaluated. First, the functional units required to execute all operations in each cluster are assigned to the design. Next, all operations are scheduled into control steps. After scheduling has been completed, the lifetimes of all values are computed and the maximum number of bits required to store those values between any two control steps is determined. The number of interconnections between each pair of clusters is also

2.3 Fasolt

Fasolt [Knap89] is an interactive feedback-driven datapath synthesis tool that uses layout information to drive decision making in the scheduling and allocation tasks. Fasolt first constructs a schedule and a RT-level datapath structure. Then, it uses a layout estimator to perform macrocell-style placement and routing. The resulting layout is back annotated with geometric information. Fasolt uses this information to merge buses and to resolve scheduling conflicts caused by the bus merging. The process is repeated until there is no more improvement.

2.4 LASSIE

LASSIE [TrDi89] is a layout-synthesis system that synthesizes layout from structural netlists generated by the behavioral synthesis tools [TLWN90]. In LASSIE, the design process is divided into four steps: structural binding, partitioning, placement and routing. Structural binding maps the generic RT components onto a set of module generators that are available in the given technology such as bit-sliced stack or standard cells. When a stack contains a wide range bit-width of bit-sliced units, a large amount of empty space exists within the datapath bounding box. Partitioning is used to cluster similar sizes of bit-sliced units together to form a stack and thus to reduce the empty space in the bounding box of a large stack. Partitioning is also used to divide a large datapath into multiple modules when the datapath is too large to fit into a chip. Finally, LASSIE performs placement and routing procedures to produce datapath layouts. In essence, LASSIE

determined. With this information, the number and size of registers, multiplexers and wires within each cluster is known so that the length, width and area of each cluster can be estimated using an area model. An approximate floorplanner based on a min-cut partitioning algorithm is used to estimate the total chip area. I/O pads are placed around the boundary of the chip for the global variables given in the behavioral description and for any off-chip memory. Finally, the datapath delays and the clock period are estimated using a timing model based on register-transfer delay. The total area and clock-cycle time are used to compare the design with others that have been encountered in the search.

2.2 Chippe

Chippe [BrGa90] is an expert system that uses a closed loop design iteration technique to perform behavioral synthesis. Chippe uses a "knobs and gauges" approach to simplify the communication between experts and tools. Knobs specify the design goal such as area, performance and power. Gauges indicate a set of scalar quality measures of the present state of the design. The design process begins with an initial design. An area and timing model is used to evaluate the design quality. The quality measures are then feedbacked to the tools to guide the design decisions. The process is repeated until the given constraints are satisfied. Chippe uses a simple datapath and control-unit (PLA) layout model for area and delay measures but it does not consider the effects of wiring delay and area.

provides a framework and implements a structural binding (mapping) step that adapts the output of behavioral synthesis tools to different datapath-layout styles.

2.5 Cathedral

Cathedral is a CAD system for DSP chip design. In the early version, Cathedral II [DRSC86] uses a microcoded processor architecture that consists of a number of customized highly programmable datapaths called execution units. The execution units are connected by dedicated buses and controlled by a microcoded controller. This architecture has very high programmability; however, it is only suited for low-speed DSP chips. In order to achieve high-speed DSP design, Cathedral III [NGCD91] uses a hardwired lowly multiplexed datapath architecture that can adapt different-style of functional units to satisfy different performance requirements.

Cathedral takes a DSP algorithm as inputs, performs memory allocation, datapath scheduling and allocation, and then generates a structural output. Finally, the datapaths are generated using a datapath layout assembler [CNSD90] and the control unit is implemented using a PLA or standard cells.

2.6 LAGER

LAGER [RaPB85, Shun91] is a CAD system for algorithm-specific IC design. LAGER consists of two main subsystems: a behavioral mapper and a silicon assembler. The behavioral mapper contains three component: a high-level compiler

(Silage [Hilf85]), a control compiler (RL [RiHi88]) and a control mapper (Sass). Silage transforms a signal processing like algorithm onto a predefined architecture that contains one or more datapaths and one control unit. RL and Sass generates a control-sequence for the target control architecture. The silicon compiler transforms the output generated by the behavioral mapper onto final layout. The silicon complier first generates a netlist for each functional unit, then simulates the netlist, followed by the layout generation, verification and performance estimation, and finally generates the mask file. In their implementation, the datapath is implemented using a bit-sliced stack while the control unit is implemented using a PLA or standard cells.

2.7 Summary

In summarizing the existing approaches, the following observations can be made:

- There are two approaches to incorporate layout information into behavioral synthesis. The first approach uses some simple layout model to estimate design area quality. The drawback of using simplified layout models is low accuracy of obtained estimates. The second approach feeds back the actual layout information by performing layout synthesis. This approach does provide accurate estimates; however, it is too slow for nontrivial designs.

- There is no complete and well-defined layout-synthesis tool for layout generation from generalized RT netlist.

- The existing chip-synthesis systems use a top-down design methodology target toward some application-specific architecture. One drawback of using the top-down design methodology is that the layout can be obtained only after the completion of the higher level tasks including behavioral synthesis and logic synthesis. In the other words, higher-level synthesis tools have to make design decisions without layout information or using some simplified layout model.

From these observed shortcoming of current efforts to chip synthesis, the following goals of the research presented in this dissertation can be stated:

- Definition of a general target architecture for various applications.
- Development a layout-synthesis technique and system for layout generation from generalized RT netlists.
- Definition and development of an accurate layout model that provides fast and accurate estimates for behavioral synthesis.
- Development of various layout-driven techniques for behavioral synthesis.

The remainder of this dissertation will present the details of the approach that has been taken to achieve these goals.

Chapter 3

Target Architecture

This chapter presents the design model called finite-state machine with a datapath (FSMD) and the layout architecture called slice-layout architecture.

The remainder of this chapter is organized in the following manner. Section 3.1 describes the FSMD design model. Section 3.2 presents the layout architecture in which Section 3.2.1 describes our motivation by introducing several commonly used layout architectures including standard cells, bit-sliced macros with abutment and bit-sliced macros with channel routing; Section 3.2.2 presents the sliced-layout architecture. Finally, Section 3.3 summarizes our target architecture.

3.1 Finite-State Machine with a Datapath

In computer science and engineering, finite-state machine (FSM) is one of the most popular design models. The FSM model usually works well for a small set of states (a few to several hundred states). However, for a complex design (up to several thousand states), the FSM model becomes too complicated to comprehend by human designers. In order to make the FSM model usable for more complex

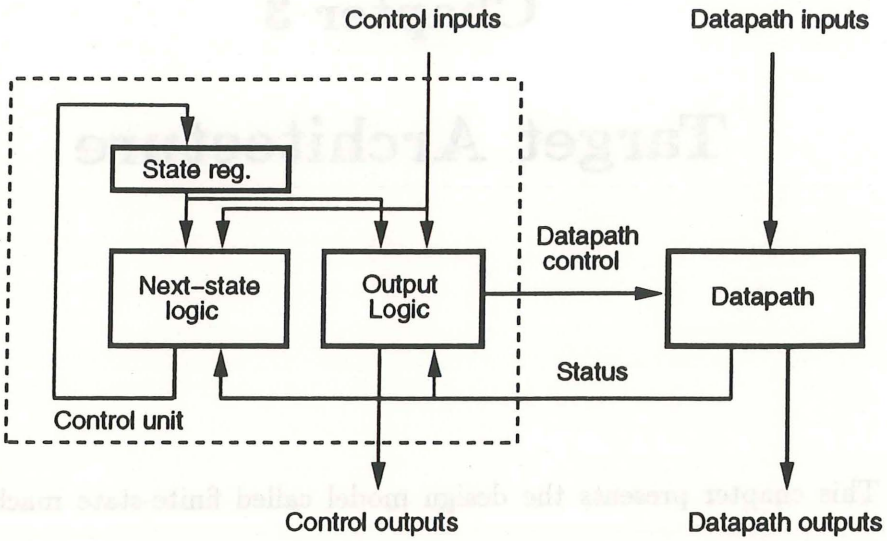


Figure 3.1: Generic FSMD block diagram

designs, Gajski et al. [GDWL92] extend the FSM model to a finite-state machine with a datapath (FSMD) model.

FSMD model is particularly useful to describe digital systems on the register-transfer level. Figure 3.1 shows a generic FSMD implementation that consists of a control unit and a datapath. Typically, a datapath contains a set of register-transfer components to perform data processing tasks. The control unit consists of a state register, next-state logic and control output logic that specify the control signals for the datapath to execute data computations in each state and to determine the sequences for the next state.

3.2 Layout Architecture

To realize the FSMD design model, this section presents the sliced-layout architecture for physical design. Section 3.2.1 discusses the motivation and overviews the other layout architectures. Section 3.2.2 describes the sliced-layout architecture.

3.2.1 Motivation

Surveys of VLSI products reveal that most fabricated chips can be described by register-transfer (RT) schematics or netlists. In addition to gates, latches and flip-flops, schematics include RT components such as registers, counters, adders, ALUs, shifters, multiplexers, bus drivers and register files. The products in this category, including DMA controllers, bus controllers, disk controllers and programmable I/O interfaces, fit well into the FSMD model.

The commonly used layout architecture for such designs is standard cells. Using the standard-cell architecture, the chip microarchitecture is decomposed into gates, such as NAND, NOR, AND and OR gates, and storage components, such as latches and flip-flops. Each of these elements is laid out manually or using a cell generator. All cells have the same height but different width. The cells are placed in rows and every two rows are separated by a routing area called routing channel. Standard-cell architecture usually uses excessive routing area and does not take advantage of the replicability of RT components that consists of many identical bit slices. Each bit-slice can be laid out as one cell instead of several standard cells. Area reduction is obtained due to several reasons: (1) the logic

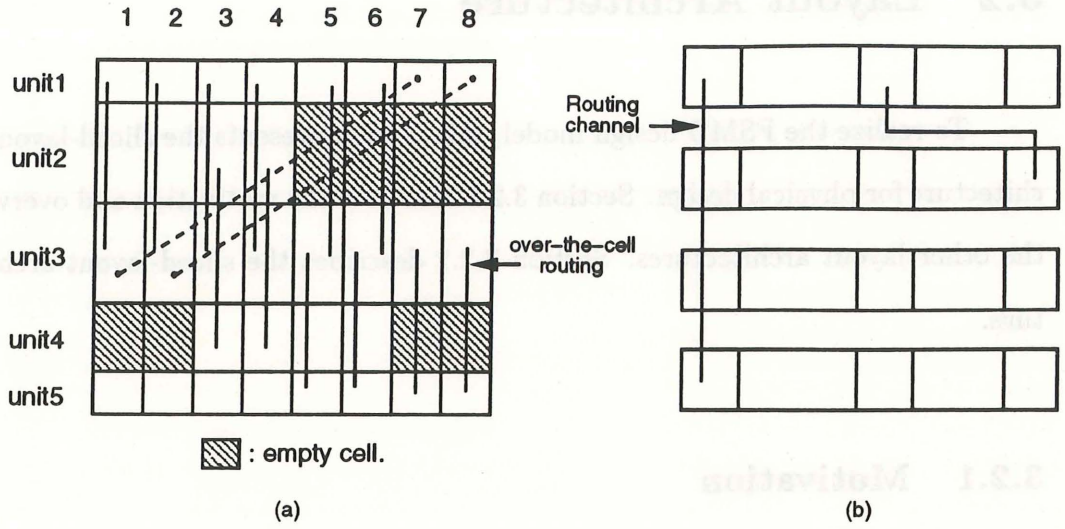


Figure 3.2: Two datapath layout architectures: (a) bit-slice abutment, (b) bit-sliced macros with channel routing.

of the bit-slice can be better optimized by use of complex gates, (2) transistors can be better packed and connected, and (3) connections can be achieved through abutment instead of through the wiring in the channel. Performance improvement is obtained through reduced wire length and proper transistor sizing.

In order to exploit the regularity of RT components, two commonly used datapath layout architectures have been developed. The first layout architecture uses abutment to connect different bit slices and over-the-cell routing for connecting different units inside one bit slice. This layout architecture is widely used in microprocessor [JaJe85, Joha79, LuDe89, Sout83] and DSP chips [PWSE86, RDVG88]. For such datapath-oriented designs, a chip will contain one or more datapath stacks and one control unit. This architecture, however, wastes area if units with varying bit-widths are in the same stack, as commonly found in designs like DMA and bus controllers. Furthermore, the connection of bit-slices with different indices is difficult. For example (Figure 3.2(a)), when 4-bit and 8-bit units are laid out in

this bit-sliced style, 4-bit slices are wasted for each 4-bit unit. In addition, if bits 7 and 8 of *unit1* must be connected to bits 1 and 2 of *unit3* (Figure 3.2(a)), routing across bit-slices must be introduced.

The second layout architecture stacks bit-sliced macros vertically with routing channels between units, as shown in Figure 3.2(b). Using this layout architecture, several units with smaller bit-widths can be placed in the same row in order to reduce wasted space. However, routing channels for wire connections between the units contribute to low area utilization.

To alleviate the problems of mentioned architectures, in the next section we describe a new layout architecture called the sliced-layout architecture.

3.2.2 Sliced-Layout Architecture

The sliced-layout architecture uses a stack of bit-sliced RT units. Figure 3.3 shows a bit-sliced unit that has the same width and a fixed number of second-metal routing tracks over each unit (13 tracks in our implementation). The unit's height, on the other hand, varies with the unit's functionality. Using the over-the-cell routing strategy, the data signals run vertically in second-metal layer over the bit slices. Power, ground, carry and control lines are routed horizontally in the first-metal or polysilicon layer between the bit slices. The layout of each unit is designed manually. The stack layout is produced by a parameterizable generator [WuCG90] according to the given bit-width and I/O pin positions. The stack grows horizontally when the bit-width increases and grows vertically when the number of units increases. A four-bit ALU stack is shown in Figure 3.4.

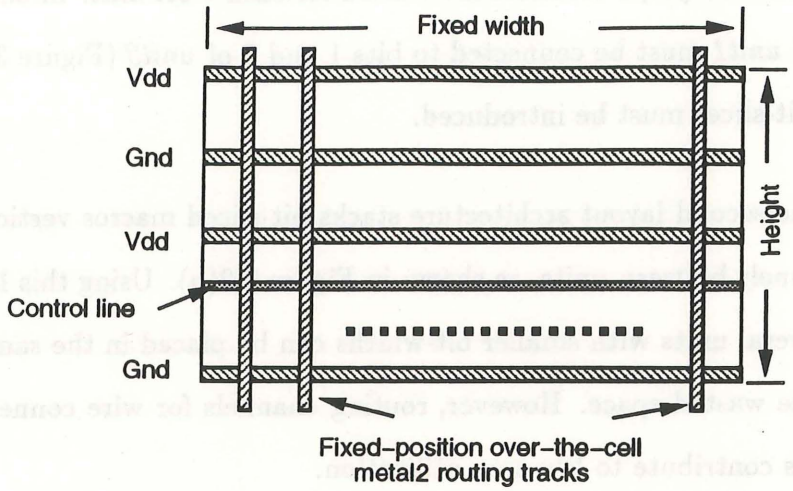


Figure 3.3: The sliced unit structure.

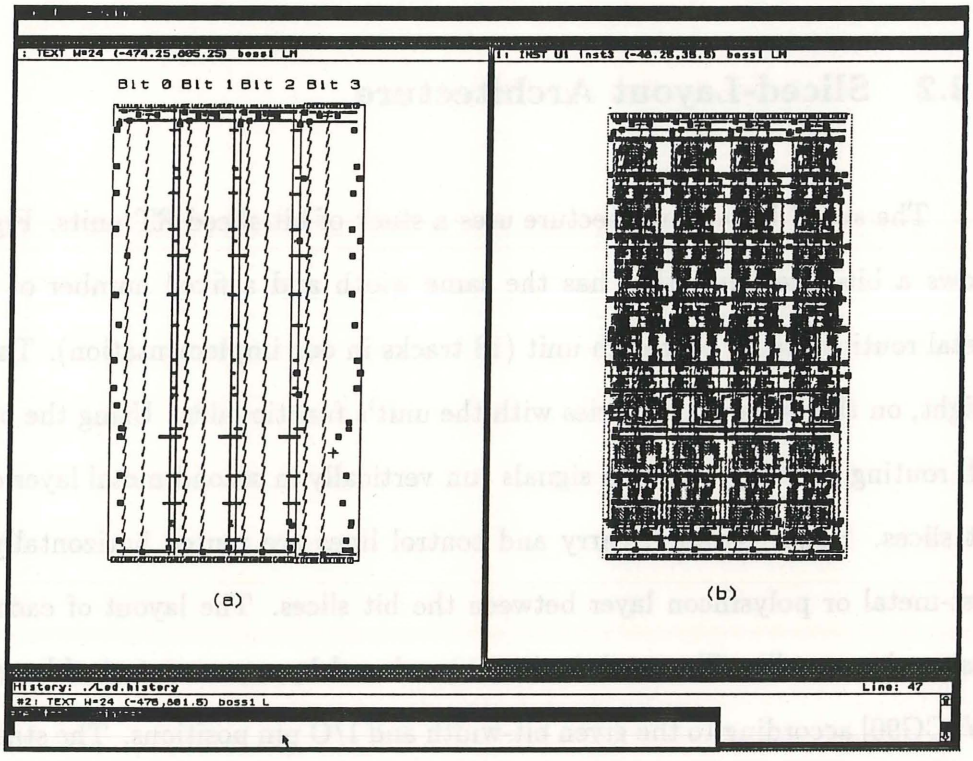


Figure 3.4: A four-bit ALU stack: (a) instance, (b) layout.

When several units of different bit-widths are stacked, units are ordered by bit-width from the top to the bottom and aligned by the least significant bit. Figure 3.5(a) shows an example consisting of three registers *Reg1, Reg2* and *Reg3.* The bit width of *Reg1* is eight, while the bit widths of *Reg2* and *Reg3* are five and four, respectively. The five least significant bits of *Reg1* are connected to *Reg2* while the four most significant bits of *Reg1* are connected to *Reg3.* Figure 3.5(b) shows the floorplan that contains a step-shaped triangular area in the stack bounding box. This area can be used for stack folding or for placement of the non-sliceable glue-logic components (Figure 3.5(c)). Furthermore, a routing channel called a switch box will be inserted in the stack to connect bit-slices with different indices. In our example, the interconnections between *Reg1* (bits 4-7) and *Reg3* (bits 0-3) are not routable without a switch box. Therefore, a switch box is inserted, as shown in Figure 3.5(c). Using the same switch box, signals may enter or exit the sliced stack from the left or right.

In the sliced-layout architecture, the stack can be laid out in two different styles. If the netlist contains only a few sliceable components then we use an unfolded stack structure as shown in Figure 3.6(a). In this case, the glue-logic components are placed into the empty space in the stack bounding box. When more space is needed for glue-logic components, they are placed on the left, right, top, and bottom sides of the stack bounding box. On the other hand, if the stack contains a large number of sliceable units with highly varying bit widths, the stack structure can be folded as shown in Figure 3.6(b). The glue-logic components are then placed at the sides of the stack bounding box. Furthermore, if the height of a folded stack is higher than a given height constraint, the stack will be partitioned into several stacks that can be either folded or unfolded.

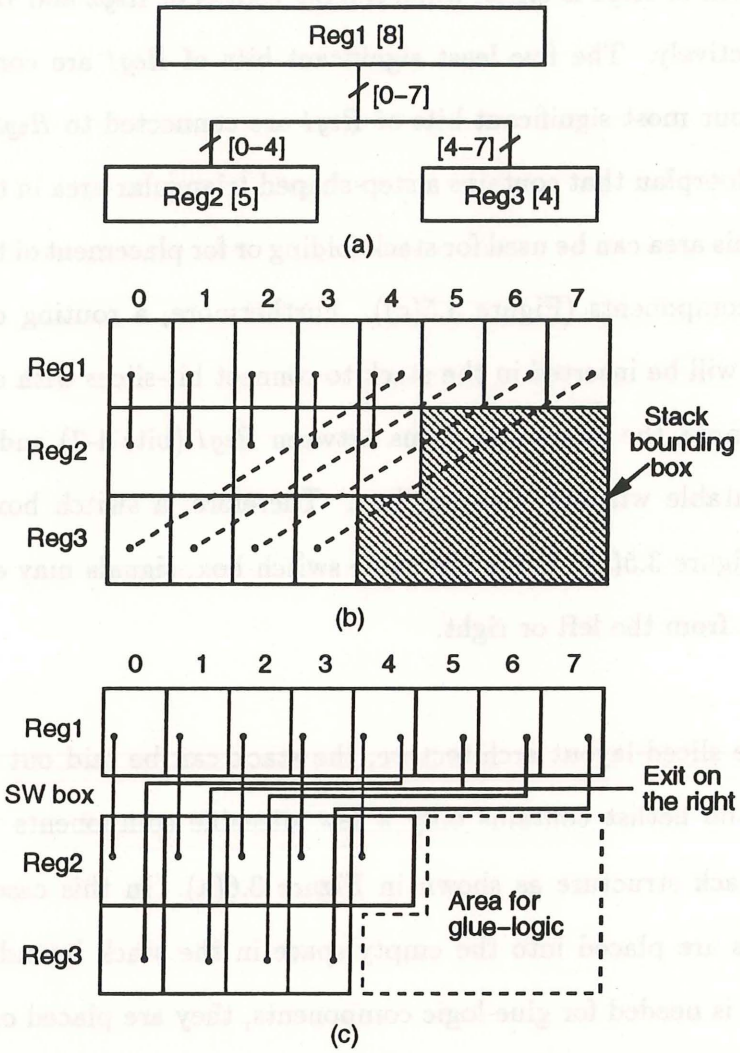
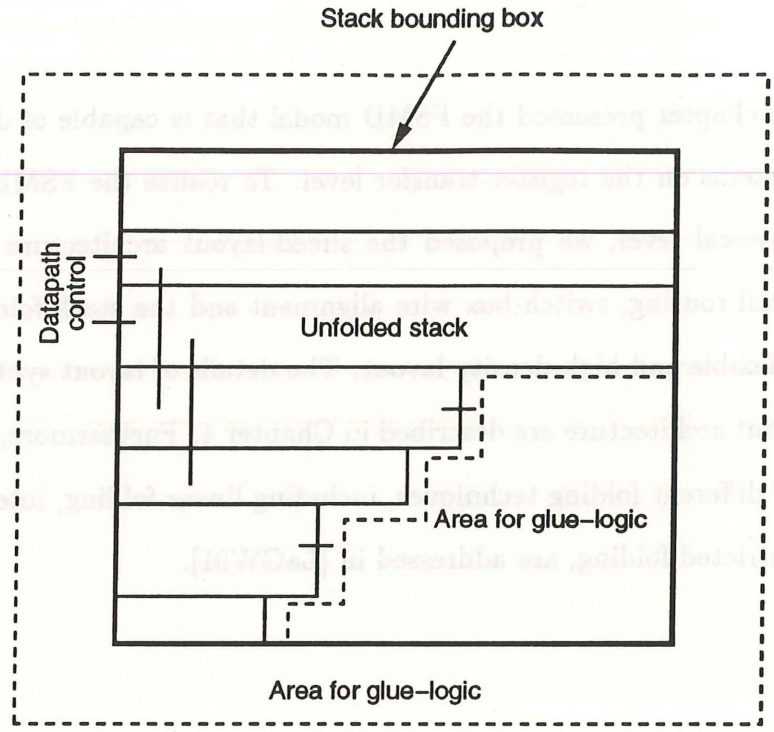
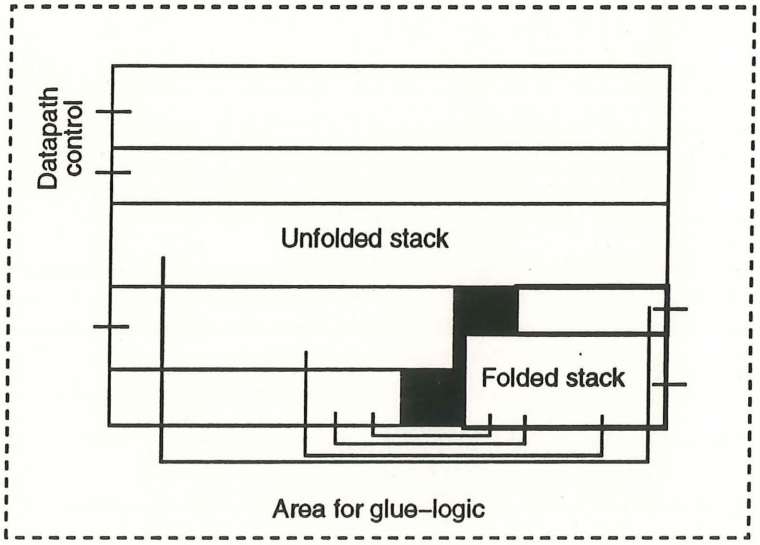


Figure 3.5: Switch box insertion for wire alignment: (a) RT netlist, (b) floorplan, (c) switch box insertion.



(a)



(b)

Figure 3.6: Two sliced-layout architectures: (a) unfolded stack, (b) folded stack.

3.3 Summary

This chapter presented the FSMD model that is capable of describing most digital systems on the register-transfer level. To realize the FSMD design model at the physical level, we proposed the sliced-layout architecture that combines over-the-cell routing, switch-box wire alignment and the stack folding method to produce flexible and high-density layout. The details of layout synthesis using the sliced-layout architecture are described in Chapter 4. Furthermore, the theoretical studies of different folding techniques, including linear folding, interleaved folding and unrestricted folding, are addressed in [LaGW91].

Chapter 4

Layout Synthesis

This chapter describe a layout synthesis system for layout generation from generalized register-transfer (RT) netlist. The system uses a new partitioning approach and the sliced-layout architecture discussed in Chapter 3 to generate the layout by considering the component layout-style, floorplan, and critical paths simultaneously. This improves the overall area utilization and minimizes the critical wire lengths, which in turn yields better performance.

The remainder of this chapter is organized in the following manner. Section 4.1 describes the overview of the system. Section 4.2 presents the partitioner and floorplanner (SLAM). Sections 4.3, 4.4, and 4.5 describe three partitioning algorithms, component partitioning, stack partitioning and glue-logic partitioning, used in SLAM. Section 4.6 presents the experimental results. Finally, Section 4.7 concludes our approach.

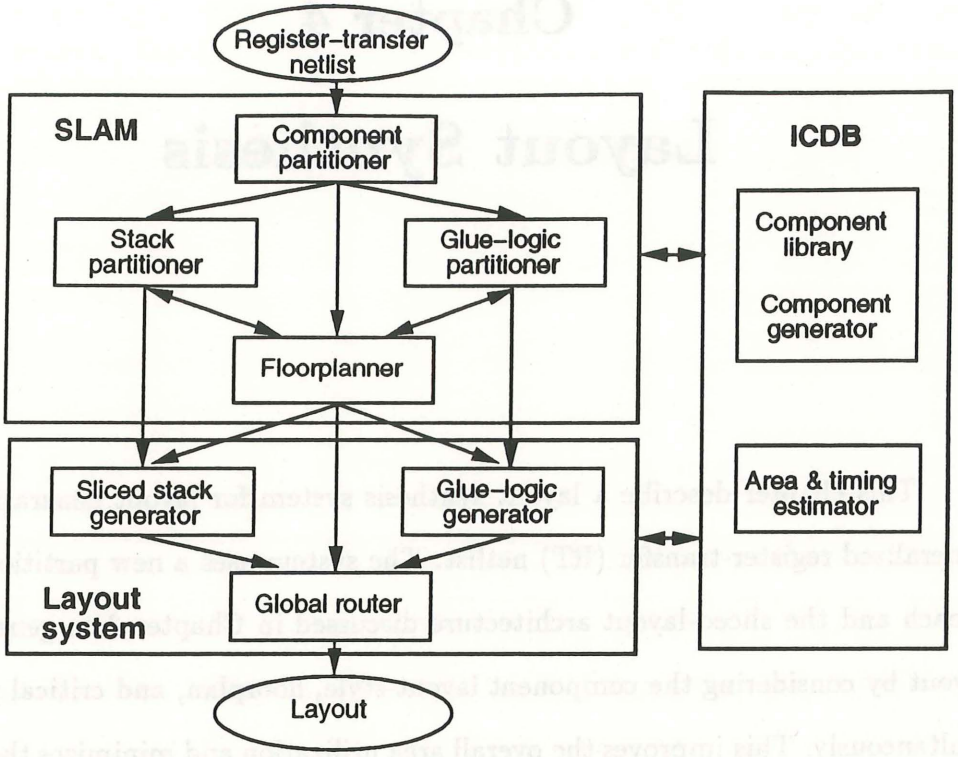


Figure 4.1: The system block diagram.

4.1 System Overview

Our system [WuCG90] is performed in a top-down fashion through three partitioning phases: (1) partitioning of the RT schematic into bit-slice and glue-logic components, (2) partitioning of bit-sliced components into several bit-sliced stacks, and (3) partitioning of glue-logic components into groups to fit area blocks around the stacks. The system consists of three parts: *SLAM* [WuGa89], *ICDB* [ChGa90] and *Layout system* (Figure 4.1). SLAM partitions the netlist into bit-sliced and glue-logic component sets. Each component set is partitioned further into clusters to form the final floorplan. ICDB provides the partitioner and the floorplanner component information (e.g., delay and area) and layout information

(e.g., aspect ratio and I/O positions) to select the best suited layout style for each component. The layout system consists of a bit-sliced stack generator, a glue-logic generator and a global router to generate final layout.

4.2 SLAM

SLAM is a partitioner and floorplanner that takes RT netlists (VHDL netlists) as inputs, partitions netlists into bit-slice and glue-logic groups, performs bit-sliced stack partitioning, dissects empty space into area blocks, assigns glue-logic components to area blocks and forms the floorplan. Using the sliced-layout architecture described in Chapter 3, SLAM performs three partitioning phases: (1) component partitioning, (2) stack partitioning and (3) glue-logic partitioning to generate the final floorplan [WuGa90].

In the first phase, the component partitioner performs the component partitioning to separate component instances into sliceable or non-sliceable sets. All of the necessary information for each component, including type, area and delay, is provided by the database ICDB. In the component partitioning, the algorithm tries to minimize the total area by exploiting the regularity of components and selecting the best suited layout style for each component. For example, a regularly-structured component, such as an 8-bit adder, a register or a comparator, is preferred to be laid out as a bit slice. On the other hand, consider a netlist that contains a 4-bit comparator performing an equal-to function, while the database contains a comparator bit-slice performing larger-than, less-than and equal-to functions. This comparator bit-slice consists of 160 transistors. However,

in this design we only need a comparator performing an equal to function that can be implemented using glue-logic components with 60 transistors. Thus, the glue-logic implementation for this comparator is less costly in terms of area and power consumption.

In the second phase, the stack partitioner performs stack folding to minimize the layout area of bit-sliced components. Since bit-sliced units often have varying bit-widths, the sliced layout architecture generates an empty space within the stack bounding box. A folding method is used to place small units into the empty space and thus reduce the stack height. The stack-folding is a two-dimensional area-filling process that considers both the bit-widths and the heights of the bit-sliced units.

In the third phase, the floorplanner dissects the empty space around the stack into area blocks by considering the given aspect ratio and the size of the glue-logic unit. In addition, the floorplanner estimates the transistor sizes of each area block. The area estimation is provided by ICDB. Then, the glue-logic partitioner uses a seed-based multiway partitioning to assign glue-logic components into area blocks. The seed-based multiway partitioning algorithm is an extension of the KLFM mincut partitioning algorithm [KeLi70, FiMa82]. In order to minimize the wire length on the critical paths, the algorithm performs two clusterings: (1) component clustering and (2) net clustering. In component clustering, the algorithm groups the components on the critical path and assigns them to the area block with the highest interconnect closeness. The net clustering is an extension of the terminal propagation strategy [DuKe85] to take into account the external connections between blocks. During the partitioning process, the algorithm clusters a set of nets (i.e., seed nets) from each partitioning set. The algorithm assigns a higher

weight to the seed nets so that the components connected to the seed nets will be attracted to their connecting nets.

Finally, each glue-logic block is generated using a layout generator [LiGa87] and the stack module is generated by generators using Mentor Graphic GDT tools. A global router then finishes the detailed routing between modules to generate the final layout.

4.3 Component Partitioning

The purpose of component partitioning is to determine the layout style (e.g., bit-slice or glue-logic) for each RT component. The component layout-style depends on the component type, the component connectivity and the overall floorplan. First, components must be partitioned by type since some components, such as counters, registers and ALUs, are sliceable while others, such as decoders and encoders, are not. Second, small size components can be implemented in two ways. For example, a 2-bit ALU can be implemented using NAND and NOR gates or as a bit-sliced unit. The implementation decision for such a component depends on its connectivity. For instance, if a component in question is strongly connected to other glue-logic components, then a glue-logic layout style may be more suitable for this component in order to reduce the wiring area between bit-sliced stack and glue-logic module. Third, the component layout style also depends on the final floorplan. For example, using the folded-stack architecture, small bit-sliced units are folded into empty space in the stack bounding-box which is described in the next section. If a small unit doesn't fit into the stack, this unit might better be

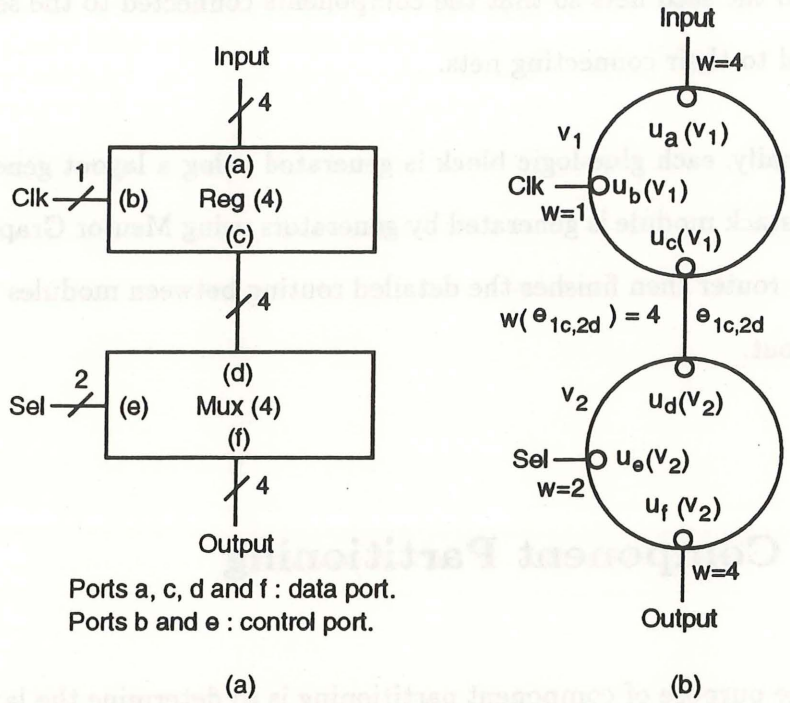


Figure 4.2: Graph representation of the RT netlist: (a) RT netlist, (b) its graph representation.

laid out using a glue-logic component in order to reduce the overall layout area. By exploiting the bit-slice property of RT components and selecting the best suited layout style for each component, the area utilization can be improved.

A weighted and labeled undirected hypergraph $G =< V, E >$ is formed from the schematic. $V=\{v_i \mid i = 1..n\}$, denotes a set of components in the schematic, and *comp_type*(v_i) denotes the component type of v_i, *Glue_Logic* or *Bit_Slice*. $u_j(v_i)$ denotes the port j of v_i, while *port_type*$(u_j(v_i))$ denotes the port type of $u_j(v_i)$, a *control* port or a *data* port. $E=\{e_{ij,kl}\}$ denotes a set of edges where $e_{ij,kl}$ is the edge between $u_j(v_i)$ and $u_l(v_k)$. In addition, $w(e_{ij,kl})$ denotes the weight of $e_{ij,kl}$ which represents the number of wires between $u_j(v_i)$ and $u_l(v_k)$.

A graph example generated from the schematic in Figure 4.2(a) is shown in Figure 4.2(b). There are two components in the schematic, a 4-bit *Reg* and a 4-bit *Mux*. In the graph, *Reg* and *Mux* form two nodes v_1 and v_2. Each node has two data ports, $\{u_a(v_1),u_c(v_1)\}$ and $\{u_d(v_2),u_f(v_2)\}$, and one control port $u_b(v_1)$ and $u_e(v_2)$. In Figure 4.2(b), $e_{1c,2d}$ is the edge between $u_c(v_1)$ and $u_d(v_2)$ where $w(e_{1c,2d})$=4.

Two linking costs, $C_{control}(v_i)$ and $C_{data}(v_i)$, are used to evaluate the connectivities between components. For a node i with m ports, the linking costs are

$$C_{control}(v_i) = \sum_{j=1}^{m} w(e_{ij,kl}) \tag{4.1}$$

where $port\_type(u_l(v_k))$=*control*, and

$$C_{data}(v_i) = \sum_{j=1}^{m} w(e_{ij,kl}) \tag{4.2}$$

where $comp\_type(v_k)$=*Bit_Slice* and $port\_type(u_l(v_k))$=*data*.

The linking cost $C_{control}(v_i)$ is the number of wires connected to v_i from other *Glue_Logic* nodes or from control ports of other *Bit_Slice* nodes, while $C_{data}(v_i)$ is the number of wires connected to v_i from data ports of other *Bit_Slice* nodes. For example, if v_1 in Figure 4.2(b) is a *Bit_Slice* and $u_f(v_2)$ connects to another *Bit_Slice* then $C_{control}(v_2)$=2 and $C_{data}(v_2)$=8.

Algorithm 4.1 describes the component partitioning procedure. The input to the algorithm is a RT netlist (S) with n components. The procedure *build_graph*(S) builds the RT-netlist graph. The function *area_estimation*(v_i) returns the cheapest implementation of component v_i (i.e., *Glue_Logic* or *Bit_Slice*). The procedure *link_cost*(v_i) returns the linking costs of component v_i. The component partitioning is divided into four steps:

1. Initial component type assignments. If bit-slice implementations are not available or the glue-logic implementation is cheaper for the nodes, the algorithm first labels such nodes as *Glue_Logic*. Otherwise, the nodes that meet the following two conditions are labeled as *Bit_Slice*: (1) the component can be laid out as a bit-slice and (2) the component's bit-width is larger than a user specified threshold. If these conditions are not met, the algorithm will label nodes as "undecided" type.
2. Type assignments for the undecided nodes. Since the undecided nodes can be laid out as a bit-slice or a glue-logic component, the algorithm takes into account the connectivity by calculating the linking cost of undecided nodes. For an undecided node v_i, if $C_{control}(v_i) > C_{data}(v_i)$ then node v_i is labeled as *Glue_Logic*, otherwise node v_i is labeled as *Bit_Slice*.
3. Final assignment. In this step, the algorithm re-evaluates the connectivities among nodes to finalize the component type assignments. The algorithm first calculates the linking cost for all of the nodes. Then the algorithm evaluates the connectivities of nodes that can be laid out as both *Glue_Logic* and *Bit_Slice*. For evaluating a node v_i, there are three possible cases: (1) if $C_{control}(v_i) > C_{data}(v_i)$ and node v_i is a *Bit_Slice* then node v_i is re-labeled as *Glue_Logic*, (2) if $C_{data}(v_i) > C_{control}(v_i)$ and node v_i is a *Glue_Logic* then node v_i is re-labeled as *Bit_Slice* and (3) if conditions (1) or (2) do not apply, a node v_i keeps its original component type.
4. Reassignment during folding. During the stack folding stage, the algorithm may re-label nodes as *Glue_Logic* if the components can not fit into the bit-sliced stack. Details of this phase will be described in the next section.

Algorithm 4.1. Component Partitioning.
Let
S be the register-transfer netlist;
BW be the minimum bit-width for a bit-slice implementation;

```
Component_Partitioning(S){
    build_graph(S);
    /*Initial component type assignment*/
    for i = 1 to n do{
        if (v_i is not sliceable) then
            comp_type(v_i) = Glue_Logic;
        else if (v_i is sliceable) then{
            type = area_estimation(v_i);
            if (type = Glue_Logic) then
                comp_type(v_i) = Glue_Logic;
            else if (type = Bit_Slice) then{
                if (bitwidth(v_i) > BW) then
                    comp_type(v_i) = Bit_Slice;
                else
                    comp_type(v_i) = undecide;
            }
        }
    };
    /*Assign component type to the undecided nodes*/
    for i = 1 to n do{
        if (comp_type(v_i) = undecide) then{
            {C_control(v_i),C_data(v_i)} = link_cost(v_i);
            if (C_control(v_i) > C_data(v_i)) then
                comp_type(v_i) = Glue_Logic;
            else
                comp_type(v_i) = Bit_Slice;
        }
    }
    /*Final assignment*/
    for i = 1 to n do{
        {C_control(v_i),C_data(v_i)} = link_cost(v_i);
        if (v_i can be laid out as both Bit_Slice and Glue_Logic) then{
            if (C_control(v_i) > C_data(v_i) AND comp_type(v_i) = Bit_Slice) then
                comp_type(v_i) = Glue_Logic;
            else if (C_data(v_i) > C_control(v_i) AND
                comp_type(v_i) = Glue_Logic) then
                comp_type(v_i) = Bit_Slice;
        }
```

```
    }
    Reassignments during folding (see Algorithm 4.2);
}
```

Complexity analysis: It takes $O(n + m)$ time to build a graph where n is the number of components and m is the number of edges in the netlist. In addition, it takes $O(n)$ time each for initial type assignment, type assignment for undecided nodes and final assignment. Therefore, the complexity of component partitioning algorithm is $O(n + m)$.

4.4 Stack Partitioning

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

Using the folding method, we describe a stack-partitioning algorithm for minimizing the area of the bit-sliced units as follows: the algorithm first calculates the routing-area cost between unfolded and folded units. The routing area is proportional to the number of wires between unfolded and folded units. For instance, the number of wires crossing the cutline between *CompA* and *CompB* is

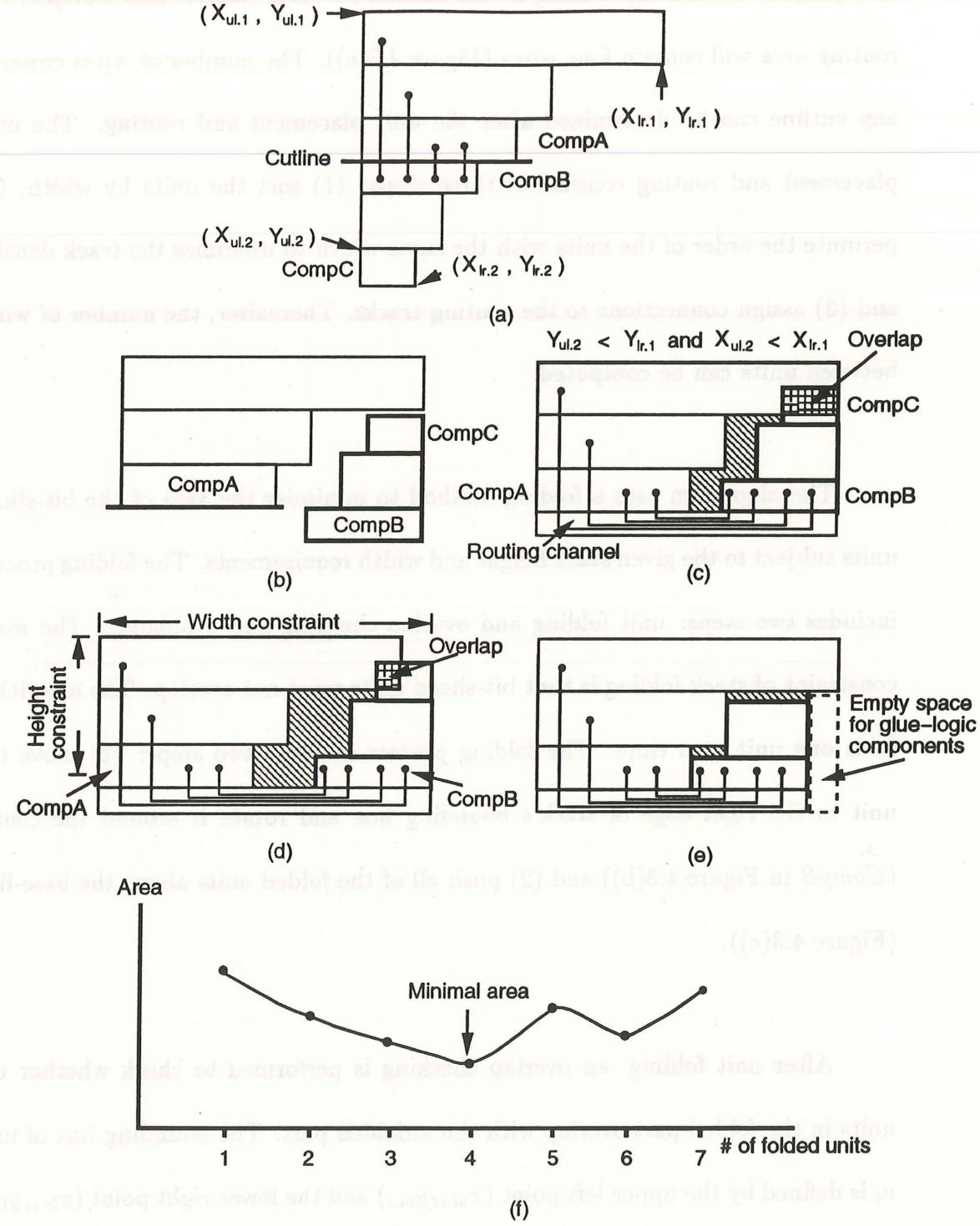


Figure 4.3: Stack partitioning based on folding: (a) linear placement, (b) unit folding, (c) overlap checking, (d) height and width constraint checking, (e) width compression, (f) stack area.

four (Figure 4.3(a)). By folding at the cutline between *CompA* and *CompB*, the routing area will contain four wires (Figure 4.3(c)). The number of wires crossing any cutline can be determined after the unit placement and routing. The unit placement and routing consists of three steps: (1) sort the units by width, (2) permute the order of the units with the same width to minimize the track density and (3) assign connections to the routing tracks. Thereafter, the number of wires between units can be computed.

The algorithm uses a folding method to minimize the area of the bit-sliced units subject to the given stack height and width requirements. The folding process includes two steps: unit folding and overlap checking and avoidance. The main constraint of stack folding is that bit-sliced units must not overlap. The algorithm folds one unit at a time. The folding process includes two steps: (1) move the unit to the right edge of stack's bounding box and rotate it around the center (*CompB* in Figure 4.3(b)) and (2) push all of the folded units above the base-line (Figure 4.3(c)).

After unit folding, an overlap checking is performed to check whether the units in the folded part overlap with the unfolded part. The bounding box of unit u_i is defined by the upper-left point $(x_{ul.i}, y_{ul.i})$ and the lower-right point $(x_{lr.i}, y_{lr.i})$ of unit u_i (Figure 4.3(a)). The overlapping occurs if there exists $u_i \in$ {unfolded bit-sliced units}, $u_j \in$ {folded bit-sliced units} and $x_{ul.j} < x_{lr.i}$ and $y_{ul.j} < y_{lr.i}$.

If an overlap occurs during the folding process, the folded units will be shifted to the right to avoid overlap until the given width constraint is reached. If the overlap still exist, the overlapped folded units will be removed to form a new stack. In Figure 4.3(d), *CompC* overlaps with unfolded units. Thus, it will be removed to form a separate stack. Moreover, if the stack height is taller than the given height constraint, then the units exceeding the given height are moved to form a new stack. For example, *CompA* and *CompB* in Figure 4.3(d) will be removed to form a separate stack. In this case, the cut line will be placed at the height constraint baseline.

In each folding iteration, the algorithm computes the total stack area that is the area of the minimum bounding box covering all the units and wires. For multiple stacks, the total area is computed as the sum of bounding boxes of individual stacks and the routing area between stacks. The algorithm folds the units one at a time and selects the stack partition with the minimum area. For instance, Figure 4.3(f) shows an area curve that was generated by executing the folding process repeatedly. Each data point represents the total area for a particular stack partition. For instance, the area data-point 1 in Figure 4.3(f) indicates the area of the bounding box covering the unfolded stack, as shown in Figure 4.3(a). The partition with the minimal area (e.g., *partition #4*) was selected as the final stack partition.

In each folding pass, the components that were deleted and leftover components are combined to produce the next stack using the same algorithm. Moreover, the stand-alone or leftover small bit-sliced units that do not fit in any stacks will be moved to the glue-logic module. For example, assume that the example in Figure 4.3(d) has the minimal area by moving *CompC* to form a new stack. Since *CompC* is a stand-alone unit, it will be relabeled as *Glue_Logic*.

After selecting the stack partition, the algorithm performs a width compression to reduce the empty space between the unfolded and folded units (Figure 4.3(e)). The empty space in the bounding box can be used for placing the glue-logic components will be described in the next section.

Algorithm 4.2 describes the stack partitioning. The input to the algorithm is a set of bit-sliced units. The procedure *place_and_route*() places units using the min-cut algorithm [KeLi70] and assigns routing tracks using the left-edge algorithm [HaSt71]. The function *overlap_check* returns "1" when the units in the unfolded and folded stacks are overlapped; otherwise, return "0". The procedure *unit_shift_adjustment*() arranges the units in the folded stack to avoid overlap.

Algorithm 4.2. Stack Partitioning.
Let
U=$\{u_i \mid i = 1..n\}$ be a set of bit-sliced units;
U_{unfold}=$\{u_j \mid j = 1..m\}$ be a set of unfolded units;
U_{fold}=$\{u_k \mid k = 1..p\}$ be a set of folded units;
U_{new}=$\{u_l \mid l = 1..r\}$ be a set of units for the new stack;
U_{unfold_final} be the final set of unfolded units;
U_{fold_final} be the final set of folded units;
$W_{constraint}$ be the width constraint;

```
    H_constraint be the height constraint;
    W_stack be the stack width;
    H_stack be the stack height;
    A_new be the total area of a new partition;
    A_min be the minimum total area;
Stack_Partitioning(U){
    place_and_route(U);
    A_min = area_estimation(U);
    if (empty space is too large for glue-logic components OR
        H_stack > H_constraint) then{
        U_unfold = U;
        U_fold = φ;
        for i = 1 to n do{
            /*folding u_i*/
            U_unfold = U_unfold − {u_i};
            U_fold = U_fold ∪ {u_i};
            overlap = overlap_check(U_unfold, U_fold);
            if (overlap = true) then{
                unit_shift_adjustment(U_fold);
                overlap = overlap_check(U_unfold,U_fold);
                if (overlap = true) then
                    U_new = U_new ∪ {u_i∈U_fold AND u_i overlaps with u_j∈U_unfold};

            }
            if (H_stack > H_constraint) then
                U_new = U_new ∪ {u_i∈{U_fold, U_unfold}
                    and u_i exceeds the height constraint};
            A_new = area_estimation(U_new);
            if (A_min > A_new AND H_constraint ≥ H_stack) then{
                U_fold_final = U_fold;
                U_unfold_final = U_unfold;
                A_min = A_new;
            }
        }
    }
    if (U_new ≠ φ AND U_new contains only small units) then
        Glue_Logic = Glue_Logic ∪ U_new;
    else
        Stack_Partitioning(U_new);
}
```

Complexity analysis: For each stack folding process, the algorithm folds at most n units. The overlap checking procedure takes $O(n)$ time. Therefore, the algorithm takes $O(n^2)$ time for each stack folding process. Since at most n stacks can be generated, the complexity of stack partitioning algorithm is $O(n^3)$.

4.5 Glue-Logic Partitioning

After forming the bit-sliced stack, a glue-logic partitioning is performed to assign the glue-logic components into clusters and to fill the empty area around the stack. The glue-logic partitioning algorithm consists of two phases: capacity estimation and iterative partitioning. In the capacity estimation phase, the algorithm partitions layout area into area blocks to satisfy the required height, width and aspect ratio. In addition, the algorithm estimates the transistor capacity of each area block.

In the iterative phase, the algorithm uses a seed-based multiway partitioning to assign glue-logic components into the area blocks. Seed-based partitioning reduces total wire length in two ways: it clusters the components on the critical path and it moves the components toward their connecting ports. The algorithm runs iteratively and selects the partition with the minimum total area as the final floorplan.

4.5.1 Definitions

Let $S(T)$ denote the size of glue-logic components in number of transistors. $T_{critical-path}$ denotes a set of components on the critical path from an input port to an output port in the design.

Let $B=\{b_i \mid i = 1..k\}$ denote a set of orthonormal edges defining a constraint area (usually used by a datapath stack) not available for the layout of glue-logic. Similarly, let $B_F=\{b_{TOP}, b_{BOTTOM}, b_{LEFT}, b_{RIGHT}\}$ denote the rectangle of the final module layout. A set of ports on an edge b_i is denoted by $P(b_i)$.

The area A around the constraint area is partitioned into rectangular area blocks $A=\{a_i \mid i = 1..j\}$. Each a_i is defined by a 4-tuple $<b_{ir}, b_{il}, b_{it}, b_{ib}>$, where b_{ir},b_{il},b_{it} and b_{ib} are the right, left, top and bottom edges of the block. In addition, $c(a_i)$ denotes the transistor capacity of block a_i.

The topological relations between area blocks, the constraint area and module boundary are specified by an adjacency graph $G(V, E)$, where $V = B \cup B_F \cup A$ and $e_{ij} \in E$ if and only if two vertices v_i and v_j have a common edge. The term $w(e_{ij})$ denotes the number of wires crossing the edge e_{ij}.

The floorplan example in Figure 4.4(a) consists of one constraint rectilinear-area $B=\{b_1...b_8\}$ and one module area $B_F=\{b_{TOP}, b_{BOTTOM}, b_{LEFT}, b_{RIGHT}\}$ with the given width ($W_{constraint}$) and height ($H_{constraint}$). The area around B and inside B_F is partitioned into 6 area blocks, that is, $A=(a_1...a_6)$. This example will be used

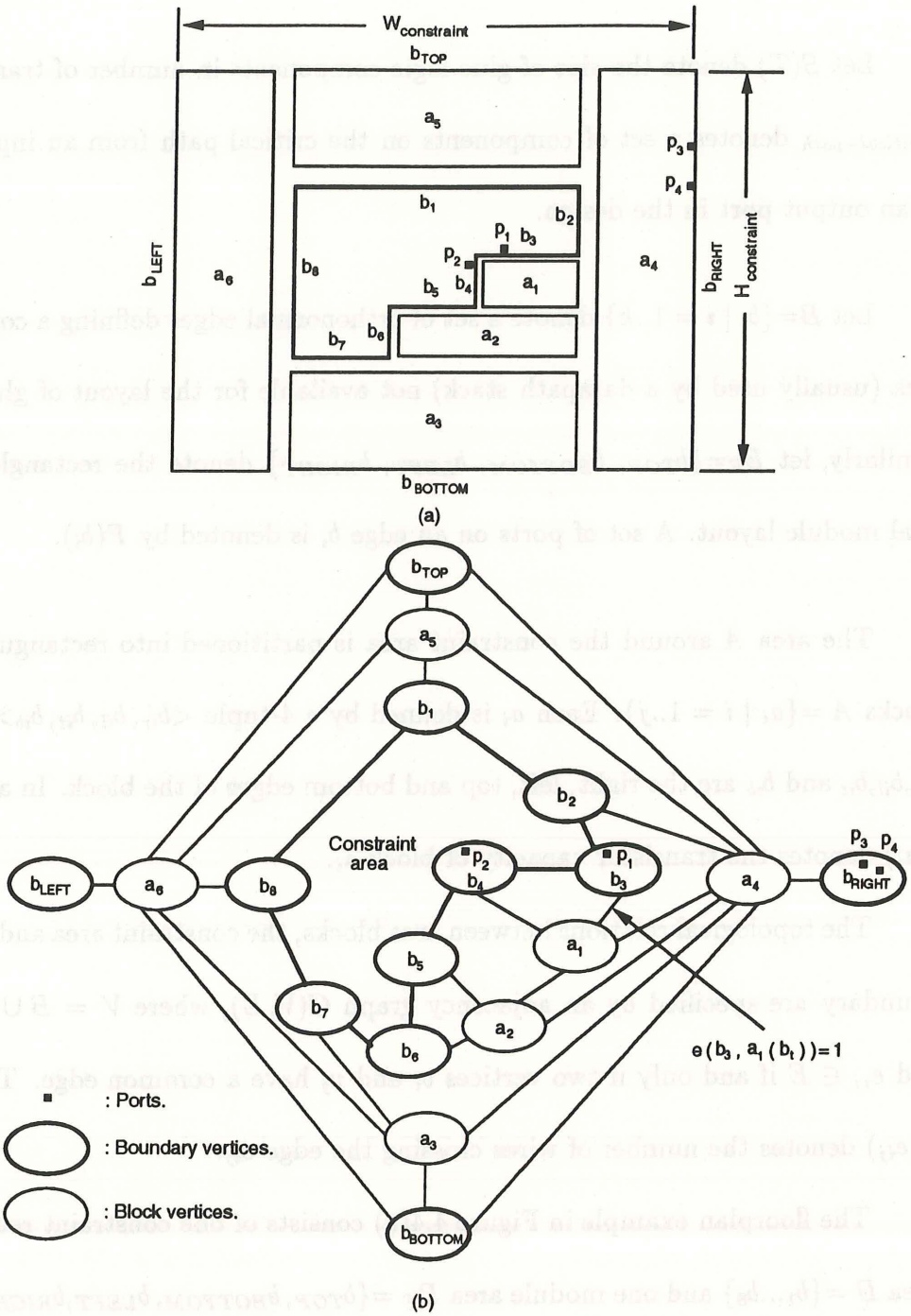


Figure 4.4: The adjacency graph formation: (a) a floorplan example, (b) its corresponding adjacency graph.

throughout this section. Figure 4.4(b) shows the adjacency graph of the floorplan in Figure 4.4(a). The adjacency graph in Figure 4.4(b) consists of 6 block vertices ($A = \{a_1, a_2, ..., a_6\}$) corresponding to the area blocks in Figure 4.4(a). There are 12 boundary vertices corresponding to the 8 edges of the constraint rectilinear-area ($B = \{b_1, b_2, ..., b_8\}$) and the 4 module edges ($B_F = \{b_{TOP}, b_{BOTTOM}, b_{LEFT}$ and $b_{RIGHT}\}$). In addition, $P(b_3) = \{p_1\}$, $P(b_4) = \{p_2\}$ and $P(b_{RIGHT}) = \{p_3, p_4\}$. The adjacency edges connect vertices with common boundaries. For example in Figure 4.4(b), there is an edge between the boundary vertex b_3 and the block vertex a_1 because these two vertices have a common boundary in b_3. Since there is a port p_1 on the boundary vertex b_3, $w(b_3, a_1) = 1$.

4.5.2 Capacity Estimation

In the capacity estimation phase, the algorithm first dissects the empty area into area blocks satisfying the height ($H_{constraint} \geq H_{module}$), width ($W_{constraint} \geq W_{module}$) and aspect ratio ($Aspect\_Ratio_{constraint} = W_{module}/H_{module}$) constraints, where H_{module} and W_{module} are the actual height and width of the layout module. Then the algorithm determines the transistor capacity of each area block. There are five possible area blocks (Figure 4.5): *In_block*, *Left_block*, *Right_block*, *Top_block* and *Bottom_block*.

In our implementation, we use a strip-layout style for glue-logic layout generation. In the strip-layout architecture, P and N transistors are placed in separate rows where a pair of P and N transistor rows is called a strip. For an area block, transistors can be placed into rows with vertical or horizontal orientation. The area estimation is provided by an estimator embedded in the database (ICDB) [ChGa90]. Our area models formulate area estimation as a function of transistor and wire density. Given a netlist and an area block, the estimator performs min-cut partitioning to estimate wiring density between transistor rows and provides the transistor capacity of the area block.

The algorithm first dissects the empty area in the bounding box *In_block* into rectangles. Since the number of ports on the edges of constraint area are given, the global routing area in the bounding box can be estimated. For example in Figure 4.5, the empty area is dissected into two rectangles a_1 and a_2 with heights and widths (h_1, w_1) and (h_2, w_2), respectively. Both horizontal and vertical layout orientations are tried on each block. The algorithm selects the one with the highest area utilization.

After estimating the transistor capacity of the empty area in the bounding box, the algorithm then estimates the transistor capacity of the outer blocks. For simplicity, we assume the initial capacity of *Left_block* and *Top_block* are zero. As a result, the algorithm only needs to estimate the transistor capacity of *Right_block* and *Bottom_block*. The algorithm places transistor rows into *Right_block* and

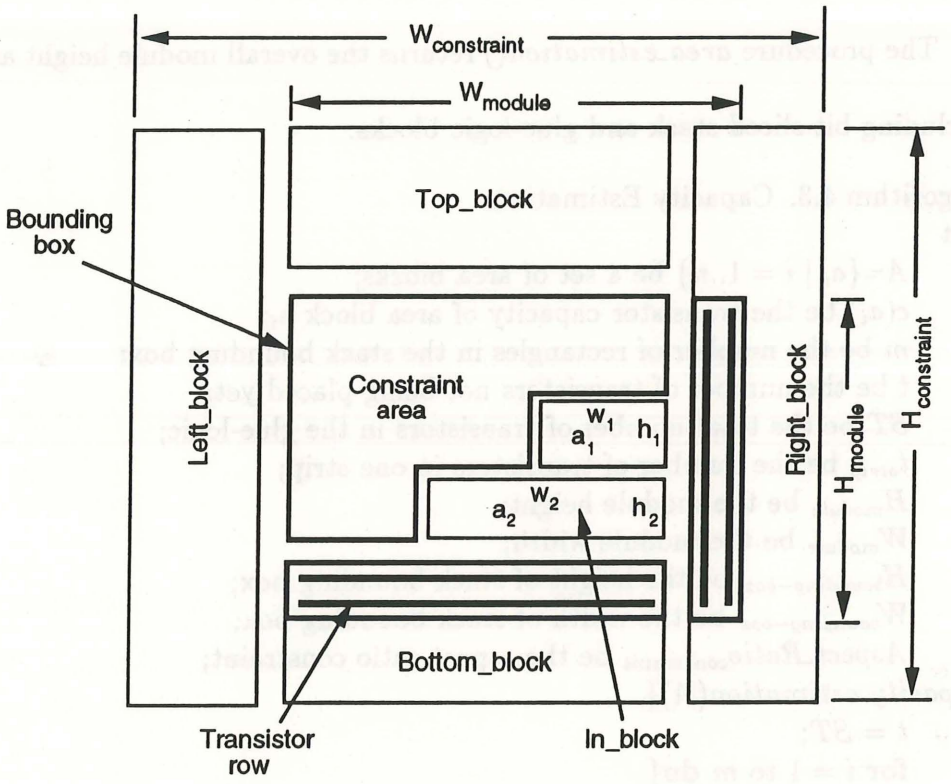


Figure 4.5: Area dissection and capacity estimation.

Bottom_block one strip at a time according to the given aspect ratio. Finally, the algorithm estimates the transistor capacity of *Right_block* and *Bottom_block*.

Algorithm 4.3 describes the capacity estimation. The input to the algorithm is a set of area blocks. The procedure *transistor_capacity_estimation*(a_i) returns the maximum number of transistors that can be placed in the given area-block a_i. The procedure *area_estimation*() returns the overall module height and width including bit-sliced stack and glue-logic blocks.

Algorithm 4.3. Capacity Estimation.
Let
A=$\{a_i \mid i = 1..n\}$ be a set of area blocks;
$c(a_i)$ be the transistor capacity of area block a_i;
m be the number of rectangles in the stack bounding box;
t be the number of transistors not being placed yet;
ST be the total number of transistors in the glue-logic;
t_{strip} be the number of transistors in one strip;
H_{module} be the module height;
W_{module} be the module width;
$H_{bounding-box}$ be the height of stack bounding box;
$W_{bounding-box}$ be the width of stack bounding box;
$Aspect\_Ratio_{constraint}$ be the aspect ratio constraint;

```
capacity_estimation(A){
    t = ST;
    for i = 1 to m do{
        c(a_i) = transistor_capacity_estimation(a_i);
        t = t − c(a_i);
    }
    c(Left_block) = 0;
    c(Top_block) = 0;
    c(Right_block) = 0;
    c(Bottom_block) = 0;
    W_module = W_bounding-box;
    H_module = H_bounding-box;
    while (t > 0) do{
        if (Aspect_Ratio_constraint > W_module/H_module) then{
            /*place transistor rows into right_block*/
```

```
            t_strip = transistor_capacity_estimation(H_module);
            c(Right_block) = c(Right_block) + t_strip;
        }
        else{
            /*place transistor rows into bottom_block*/
            t_strip = transistor_capacity_estimation(W_module);
            c(Bottom_block) = c(Bottom_block) + t_strip;
        }
        t = t - t_strip;
        /*estimate the module height and width for the new partition*/
        W_module = area_estimation(W_bounding-box, c(Right_block));
        H_module = area_estimation(H_bounding-box, c(Bottom_block));
    }
}
```

Complexity analysis: For m rectangles in the bounding box, it takes $O(m)$ time to estimate the transistor capacity of m rectangles. For the transistor capacity estimations of *Right_block* and *Bottom_block*, it takes approximately $O(n)$ time where $n = t/t_{strip}$. Thus, the complexity of the capacity estimation algorithm is $O(m+n)$.

4.5.3 Iterative Partitioning

Seed-Based Multiway Partitioning

The seed-based multiway partitioning algorithm is an extension of the KLFM min-cut partitioning algorithm. Let $A=\{a_i \mid i = 1..n\}$ be the set of pre-defined area blocks and $c(a_i)$ be the transistor capacity of area block a_i. The total number of transistors of the glue-logic components is $S(T) \leq \sum_{i=1}^{n} c(a_i)$. The algorithm

performs min-cut partitioning repeatedly based on the cut-set size of the partition (C_a, C_b) where $C_a = c(a_i)$ and $C_b = \sum_{j=i+1}^{n} c(a_j)$, $i = 1..n-1$.

In order to minimize the wire length on the critical paths, all the components on the critical paths are clustered together. These components are called seed components. Moreover, the components connected to ports on the constraint area and the module boundaries are placed close to those ports. The nets connected to the ports on the constraint-area edges and the module edges are called seed nets. The seed-based approach is an extension of the terminal propagation strategy [DuKe85] that takes into account the external connections between blocks. The algorithm performs seed-net clustering for both cut-sets before the partitioning process takes place. Using the hierarchical-clustering technique [John67], the algorithm is capable of successively fusing seed nets into clusters for each cut-set at different stages of the partitioning process. The seed clustering is divided into two parts: seed-component clustering and seed-net clustering.

In the component clustering, the algorithm first groups the components on the critical path ($T_{critical-path}$). Then, the algorithm evaluates the connectivities between the nets in $T_{critical-path}$ and their connecting ports. The components in $T_{critical-path}$ will be placed into the area block with the maximum "closeness" cost.

To take external connections into account, the net clustering determines the seed nets for both cut-sets. During the partitioning process, the seed nets will

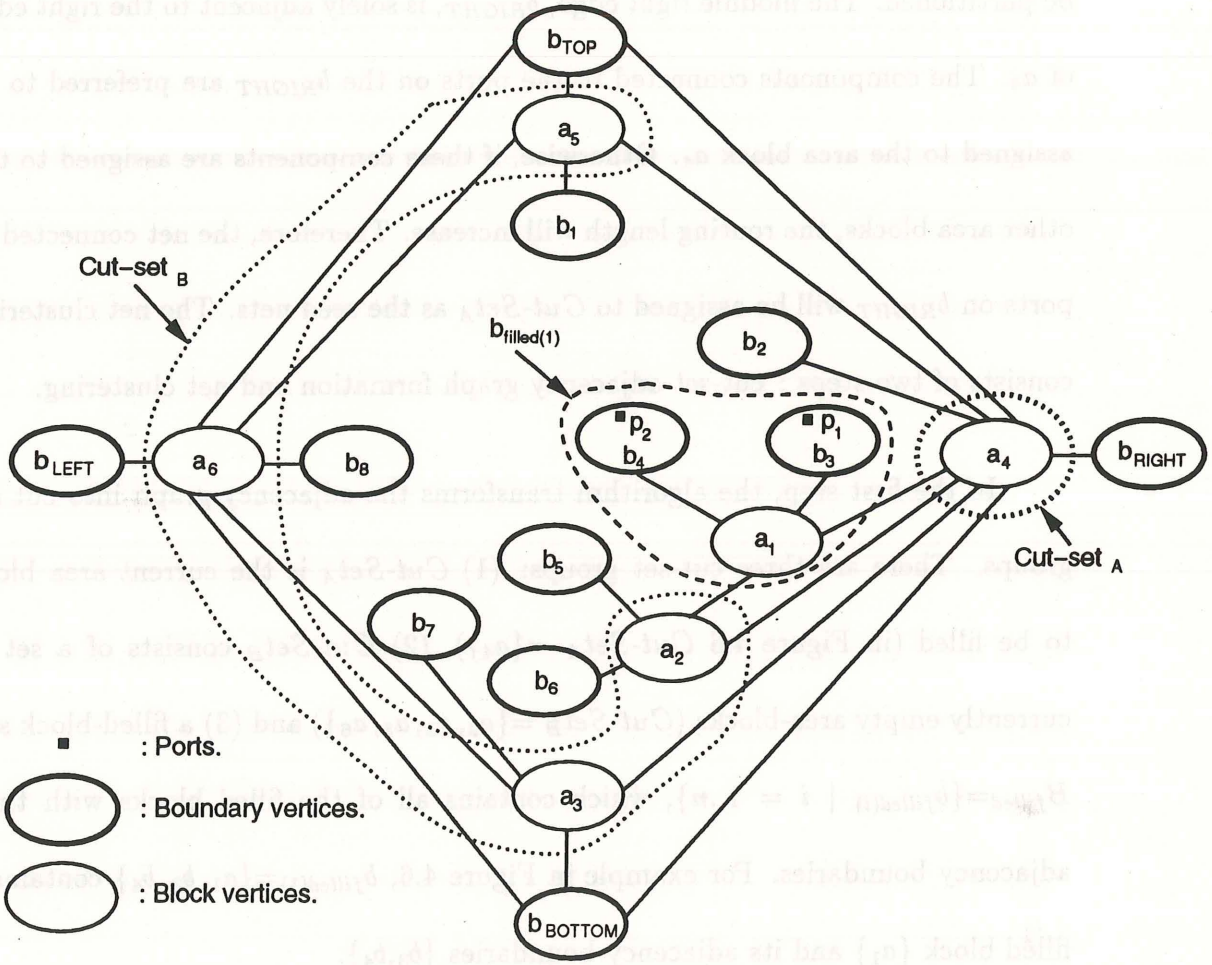


Figure 4.6: Cut-set adjacency graph.

be assigned a higher weight to enhance the connectivity. This approach pulls the components connected to the seed nets toward their connecting ports. For example in Figure 4.6, the area block a_4 in $Cut\text{-}Set_A$ is the current area block to be partitioned. The module right edge, b_{RIGHT}, is solely adjacent to the right edge of a_4. The components connected to the ports on the b_{RIGHT} are preferred to be assigned to the area block a_4. Otherwise, if these components are assigned to the other area blocks, the routing length will increase. Therefore, the net connected to ports on b_{RIGHT} will be assigned to $Cut\text{-}Set_A$ as the seed nets. The net clustering consists of two steps : cut-set adjacency graph formation and net clustering.

In the first step, the algorithm transforms the adjacency graph into cut-set groups. There are three cut-set groups: (1) $Cut\text{-}Set_A$ is the current area block to be filled (in Figure 4.6 $Cut\text{-}Set_A = \{a_4\}$), (2) $Cut\text{-}Set_B$ consists of a set of currently empty area-blocks ($Cut\text{-}Set_B = \{a_2, a_3, a_5, a_6\}$) and (3) a filled-block set, $B_{filled} = \{b_{filled(i)} \mid i = 1..n\}$, which contains all of the filled blocks with their adjacency boundaries. For example in Figure 4.6, $b_{filled(1)} = \{a_1, b_3, b_4\}$ contains a filled block $\{a_1\}$ and its adjacency boundaries $\{b_3, b_4\}$.

In the second step, the algorithm determines seed nets for both cut-sets. The nets and ports that are solely adjacent to certain cut-sets are called the seed nets of that cut-set. For example in Figure 4.6, b_2 and b_{RIGHT} are solely adjacent to the block vertex a_4 in $Cut\text{-}Set_A$. Thus, the nets and ports of b_2 and b_{RIGHT} will be assigned as the seed nets of $Cut\text{-}Set_A$. On the other hand, $\{b_1, b_5, b_6, b_7, b_8, b_{LEFT}\}$

are solely adjacent to the block vertices in $Cut\text{-}Set_B$. Thus, the nets and ports of $\{b_1, b_5, b_6, b_7, b_8, b_{LEFT}\}$ are the seed nets of $Cut\text{-}Set_B$.

The Algorithm

In the capacity estimation phase, the algorithm determines the transistor capacity of area blocks. Initially, the algorithm assumes $c(Top\_block)$ and $c(Left\_block)$ are zero. However, the transistor rows in *Bottom_block* can also be placed in *Top_block*. Further, the transistor rows in *Right_block* can be placed in *Left_block*. In this phase, the algorithm uses an iterative partitioning method to find the minimum area partition by rearranging the capacity of area blocks.

Based on the example in Figure 4.4, the algorithm initially assigns $c(a_6)=0$, $c(a_4)=c(Right\_block)$, $c(a_3)=c(Bottom\_block)$ and $c(a_5)=0$. The algorithm then rearranges the capacity of area blocks during partition by assigning strips from a_4 to a_6 and from a_3 to a_5. In order to minimize routing, finally, the algorithm performs seed-based multiway partitioning to assign components into area blocks according to the assigned transistor capacity of area blocks. After each partitioning iteration, the total layout area is calculated. The total layout area consists of three parts:

1. Constraint area, $Area_{constraint-area}$, which is the area of bit-sliced stacks or macrocells.

2. Glue-logic area. After partitioning, the glue-logic components are placed into a set of area blocks $A=\{a_i \mid i = 1..n\}$. The area for each block a_i, $Area_{glue-logic-block(i)}$, is estimated.

3. Routing area. After partitioning, the glue-logic components are placed into a set of area blocks. The cutlines crossing the boundary are estimated. The routing area, $Area_{global-routing}$, between two area blocks is calculated in terms of the number of cutlines crossing these two blocks. The total layout area is

$$Total\_Area = Area_{constraint-area} + \sum_{i=1}^{n} Area_{glue-logic-block(i)} + Area_{global-routing} \tag{4.3}$$

Algorithm 4.4 describes the glue-logic partitioning. The input to the algorithm is a set of area blocks and a glue-logic netlist. The procedure *capacity_estimation*() is described in Algorithm 4.3. The procedure *Seed_based_multiway_partition*() assigns glue-logic components into area blocks. The function *total_area_calculation*() returns the total area-cost using Equation 4.3. The set M keeps track of the set of area blocks with the minimum total area. The algorithm runs iteratively and selects the partition with the minimum total area as the final floorplan.

Algorithm 4.4. Glue-Logic Partitioning.
Let

GL be a glue-logic netlist;
$A=\{a_i \mid i = 1..n\}$ be a set of area blocks;
$M=\{a_i \mid i = 1..n\}$ be a set of area blocks for the final floorplan;
$c(bottom\_tr\_row)$ denote the transistor capacity of one strip in *Bottom_block*;
$c(right\_tr\_row)$ denote the transistor capacity of one strip in *Right_block*;

Glue_Logic_Partitioning(A, GL)\{
/*initial partitioning*/

```
/*Determine c(In_block), c(Right_block) and c(Bottom_block)*/
capacity_estimation(A);
Build adjacency graph;
c(a4) = c(Right_block);
c(a3) = c(Bottom_block);
Seed_based_multiway_partition(A,GL);
Total_Area = total_area_calculation();
/*iterative partitioning*/
while (c(a4) > 0){
      c(a6) = c(a6) + c(right_tr_row);
      c(a4) = c(a4) - c(right_tr_row);
      Seed_based_multiway_partition(A,GL);
      Total_Area_new-partition = total_area_calculation();
      if (Total_Area > Total_Area_new-partition){
            M = {a_i | i = 1..n};
      Total_Area = Total_Area_new-partition;
      }
}
while (c(a3) > 0){
      c(a5) = c(a5) + c(bottom_tr_row);
      c(a3) = c(a3) - c(bottom_tr_row);
      Seed_based_multiway_partition(A,GL);
      Total_Area_new-partition = total_area_calculation();
      if (Total_Area > Total_Area_new-partition){
            M = {a_i | i = 1..n};
            Total_Area = Total_Area_new-partition;
      }
}
}
```

Complexity analysis: The seed-based multiway partitioning uses the bucket-list data structure [FiMa82] which has the complexity of $O(p)$ for each iteration where p is the number of pins. There are two iterative processes taking $O(m_1)$ and $O(m_2)$ time, where m_1 is the number of strips in *Right_block* and m_2 is the number of strips in *Bottom_block*. The complexity of glue-logic partitioning algorithm is $O((m_1+m_2)p)$.

4.6 Results

SLAM currently runs on SUN3/SUN4 workstations under the UNIX operating system. Several examples have been tested. The layouts were generated using a 3μm CMOS technology.

The first example is a controlled counter [Arms89] that consists of approximately 50% sliceable components and 50% non-sliceable components. The final floorplan and layout are shown in Figure 4.7 and Figure 4.8. It consists of an unfolded stack and a folded stack with a glue-logic block. The second example is the MARK1 simple computer [SiBN82] which includes 32, 16, 13, and 3 bit register-transfer components and simple gates (Figure 4.9 and Figure 4.10). The register-transfer schematics of both examples were generated by VSS [LiGa88]. The third example is the digital section of a DSP chip consisting of an ALU, registers, flip-flops, a shifter, counters, latches and simple gates. The final layout is shown in Figure 4.11.

Using the same layout generators, we compared the layouts generated using our partitioning algorithms with sliced-layout architecture to that using a manual floorplanning with traditional layout architectures. In the second case, the register-transfer schematics were first partitioned into modules consisting of bit-sliced and glue-logic components. For example, the controlled-counter example was partitioned to a set of bit-sliced units (such as a up/down counter, registers

and drivers) and a glue-logic unit. Since the bit-sliced units of this example have varying bit widths, we use the layout architecture of bit slices with routing channel. Each bit-sliced unit was laid out individually. Then, we used an interactive floorplanner to find the minimum area floorplan by placing units using an exhaustive search. To estimate the wire length on the critical path, we first identified the components on the critical path from the final layout and then measured the wire length connecting all those components. The results in Figure 4.12 show that the layouts were 10% smaller and the wire lengths on critical paths were 20%-25% shorter when our partitioning algorithms and layout architecture were used.

We have also tested the glue-logic partitioning capability on an example of 515 gates (approximately 3000 transistors), 254 I/O ports, and 900 nets. The algorithm partitioned the design into five blocks. The final layout is shown in Figure 4.13. We have tested the partitions with and without seed clustering. The experimental results show that using seed clustering the total wire length on the critical path is 15% shorter.

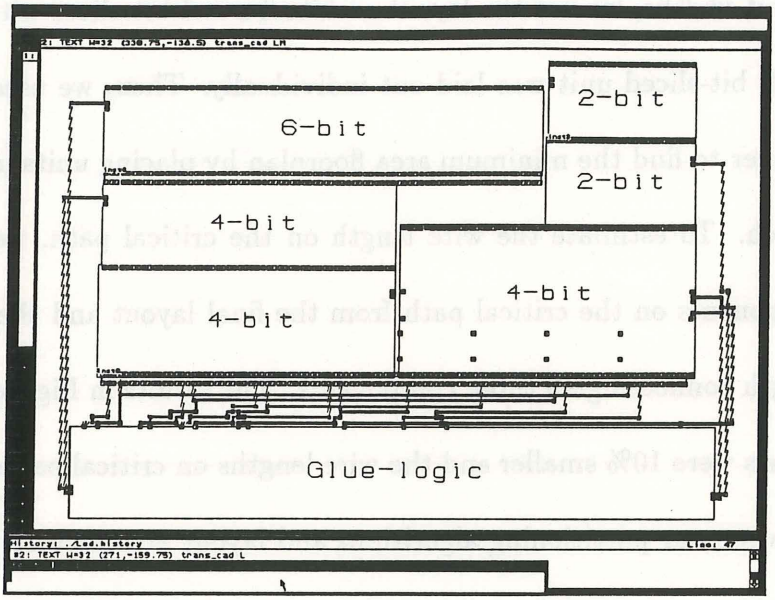


Figure 4.7: The floorplan of the controlled counter example.

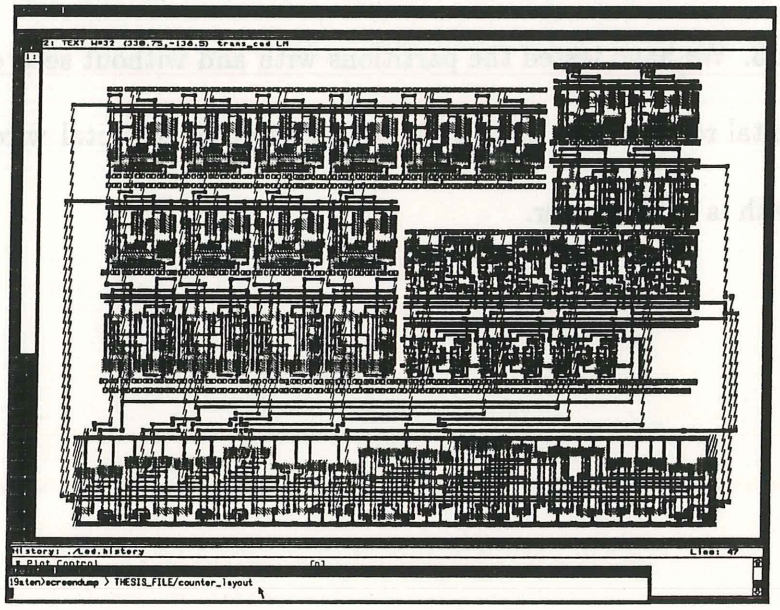


Figure 4.8: The layout of the controlled counter example.

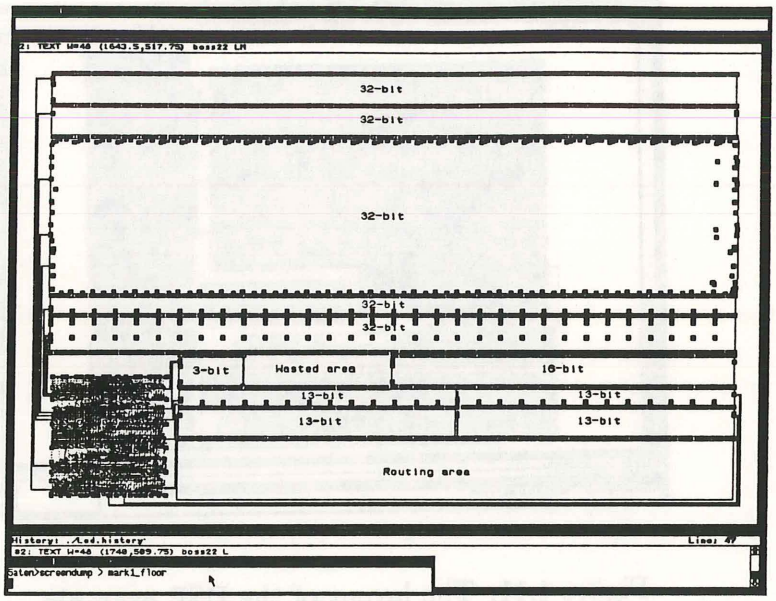


Figure 4.9: The floorplan of the MARK1 simple computer.

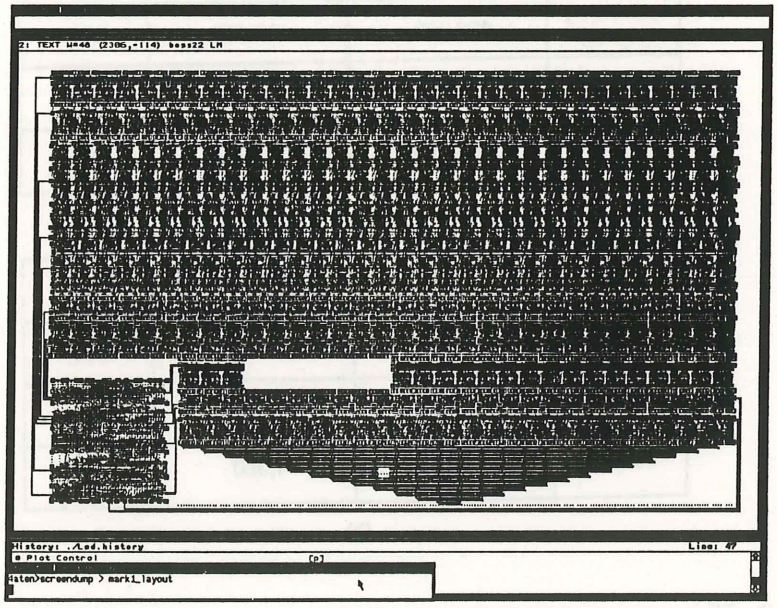


Figure 4.10: The layout of the MARK1 simple computer.

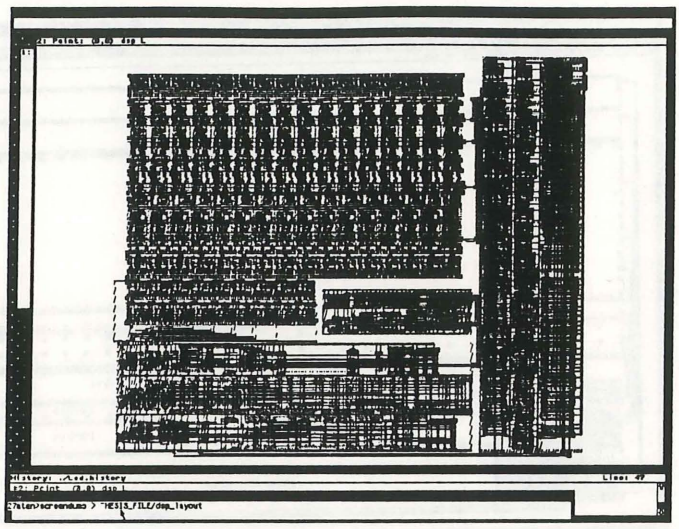


Figure 4.11: The layout of the DSP example.

Example	Our (A) um^2	Manual (B) um^2	A/B
Controlled counter	450,328	498,883	.902
DSP	7,056,000	7,896,042	.893
MARK1	11,220,000	12,701,040	.883

(a)

Example	Our (A) um	Manual (B) um	A/B
Controlled counter	594	765	.776
DSP	2,665	3,650	.730
MARK1	3,885	4,950	.784

(b)

Figure 4.12: The comparisons of our partitioning and floorplanning with a manual partitioning and floorplanning: (a) total area, (b) the critical path wire length.

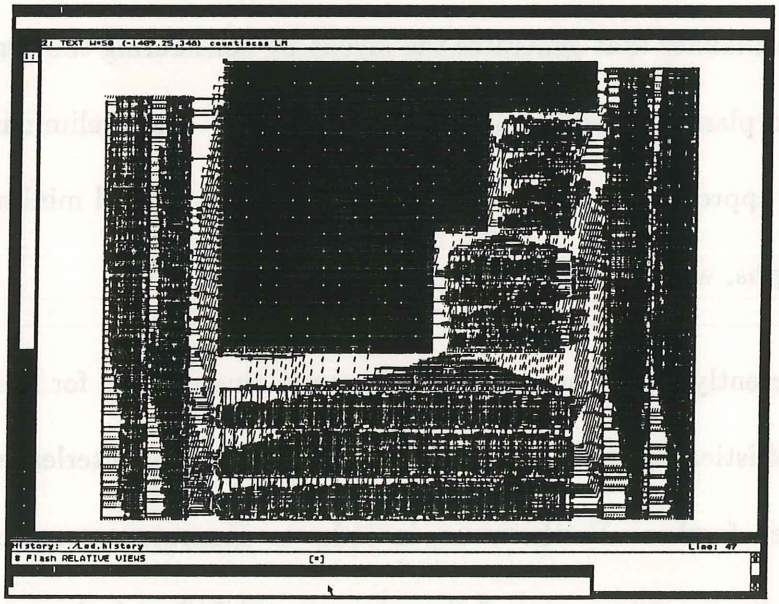


Figure 4.13: The layout of a glue-logic partitioning example.

4.7 Conclusions

In this chapter, we described a new partitioning methodology for layout generation from register-transfer netlists based on the sliced-layout architecture. We described a new algorithm for netlist partitioning and one for partitioning of bit-sliced components. We also described a seed-based multiway partitioning algorithm based on area capacity. Our partitioning algorithms are carried out in a top-down manner that generates the layout by considering the component layout style, floorplan and critical paths simultaneously. The preliminary results show that this approach improves the overall area utilization and minimizes the critical wire lengths, which in turn yields better performance.

Currently, SLAM uses a simple linear-folding method for stack partitioning. More sophisticated folding technique [LaGW91], such as interleave folding, should be studied further. Furthermore, in order to generate a complete chip, SLAM needs to incorporate a general floorplanner and I/O-pad placement and routing algorithm.

Chapter 5

Quality Measures

Traditionally, the number and size of functional units, storage units and connections or the number of AND, OR and NOT operators in the Boolean expression of the unit are used as area quality-measure in behavioral synthesis. However, these area measures assume that layout area is directly proportional to the number and size of RT components and do not take into account layout technology factors such as layout styles, component libraries, and impact of floorplanning, placement and routing. These factors often greatly affect the final layout of the design. Similarly, the number of control steps is usually used as performance quality-measure in behavioral synthesis. This performance measure is valid only if the clock cycle is fixed. However, the number of control steps do not reflect the total execution time which is equal to the product of control steps and the clock cycle.

This chapter presents two quality measures, area and performance, used to support design decisions and to determine the quality of the final synthesized design in behavioral synthesis. This proposed layout model takes into account

most technology factors such as layout architectures and technology mapping, and thus provides more accurate estimates than previous proposed models. Two main factors of quality measures, accuracy and fidelity, are also addressed in this chapter.

The remainder of this chapter is organized in the following manner. Section 5.1 describes the relationship between structural and physical designs. Section 5.2 describes the area measures including datapath and control unit. Sections 5.3 describes the performance measures including datapath delay, control delay and clock estimation. Section 5.4 presents the experimental results. Finally, Section 5.5 concludes our approach.

5.1 The Relationship between Structural and Physical Designs

Behavioral synthesis transforms an input behavioral description into a structural design composed of a datapath netlist and a control unit. Typically, the datapath netlist consists of a set of generic register-transfer (RT) components (e.g., functional, interconnect and storage units). The control unit specifies the control signals for executing register transfers in each state, as well as the sequencing for the design's next state (Figure 5.1).

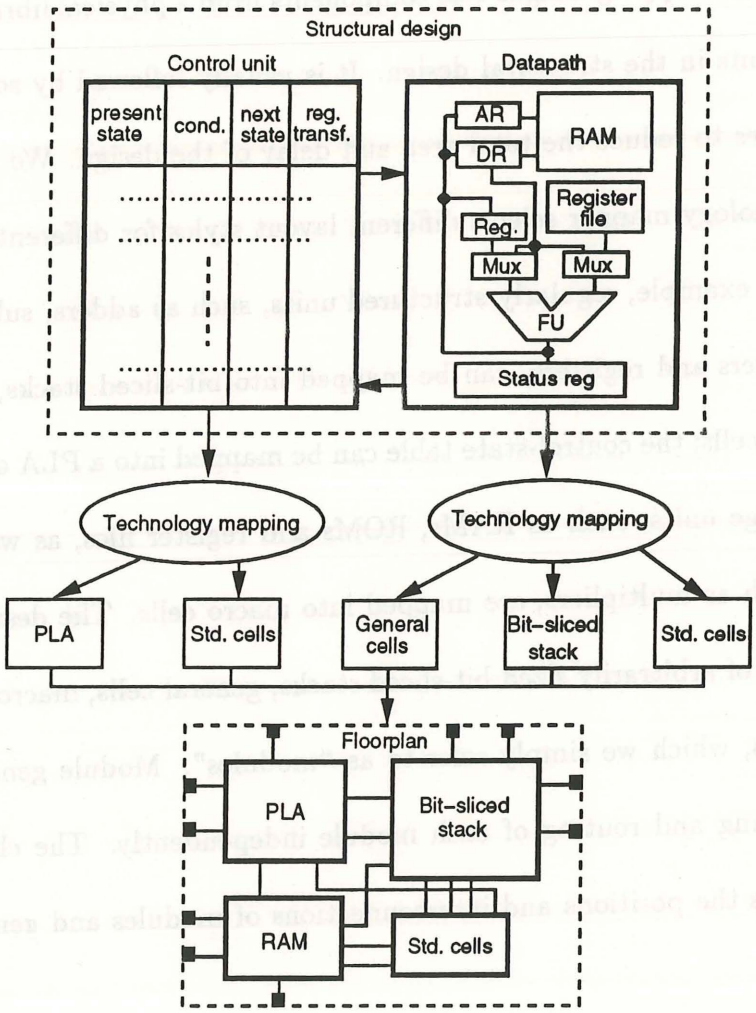


Figure 5.1: The relationship between structural and physical designs.

The process of generating fabrication data for custom or semicustom technologies from a structural design consists of several steps (Figure 5.1), including technology mapping, module generation, floorplanning, placement and routing. Technology mapping assigns real components from a physical library to the generic components in the structural design. It is usually followed by some optimization procedures to reduce the total area and delay of the design. We also assume that the technology mapper selects different layout styles for different parts of the design. For example, regularly structured units, such as adders, subtracters, ALUs, multiplexers and registers, can be mapped into bit-sliced stacks, general cells or standard cells; the control-state table can be mapped into a PLA or standard cells; and storage units, such as RAMs, ROMs and register files, as well as functional units, such as multipliers, are mapped into macro cells. The design may contain a mixture of arbitrarily sized bit-sliced stacks, general cells, macros and standard-cell blocks, which we simply refer to as "modules". Module generators perform floorplanning and routing of each module independently. The chip floorplanner determines the positions and interconnections of modules and generates the chip layout.

The total area of a design is the sum of the area of its modules, I/O pads and pad drivers, the chip routing areas and the remaining wasted areas, as illustrated in Figure 5.1. Since macros are predesigned, their areas and shapes can be obtained directly from component libraries. However, the areas and shapes of datapath

and control modules mapped into bit-sliced stacks, standard cell blocks or PLA macrocells vary greatly depending on their layout styles and architectures. In the next two sections, I will discuss area and performance measures for datapaths and control units using some sample layout styles.

5.2 Area Measures

This section describes area measures for the datapath and control unit of the FSMD model using two different layout styles for CMOS technology.

5.2.1 Datapath

A datapath consists of a set of regularly structured RT components, such as ALUs, multiplexers, latches, drivers and shifters. Datapath layout is accomplished with a stack of functional and storage units that are placed one above the other. Each unit consists of bit slices and all units have bit slices of the same width. However, bit slices in different units may be of different height. All bit slices are aligned starting with the least-significant bit (LSB) and distinct units are stacked on top of another. Thus, the stack grows horizontally when the bit width increases, and it grows vertically when the number of units increases (Figure 5.3(a)). Each bit slice of a unit may be a handcrafted custom cell or may be implemented with one

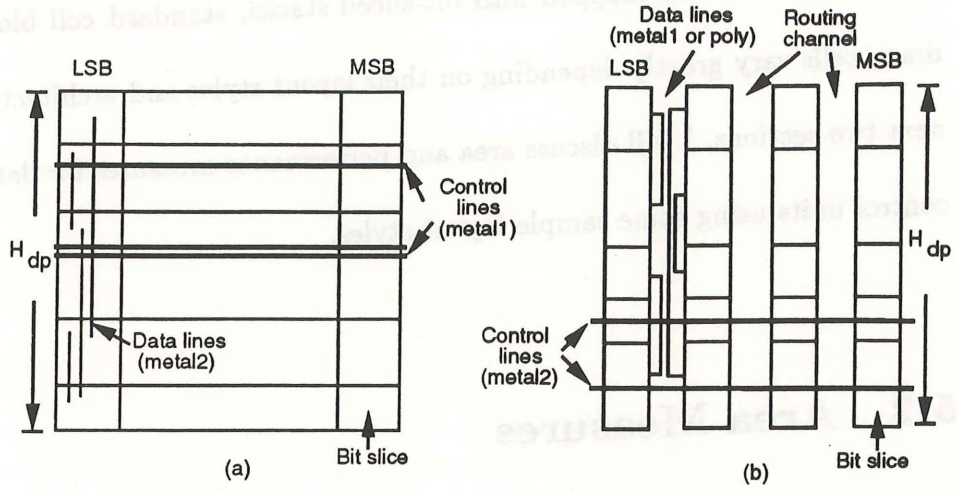


Figure 5.2: Two data path layout architectures using: (a) custom cells, (b) standard cells.

row of connected standard cells as shown in Figure 5.2(a) and (b), respectively. The difference between custom and standard-cell styles is in the layers used for routing of control and data wires, use of custom or standard cells and routing of data lines over the cells or in a separate channel.

In the first layout architecture (Figure 5.3(b)), diffusion strips for P and N transistors are placed horizontally. Power and ground wires run horizontally in the first metal layer. The control lines common to different bit slices in each unit run also horizontally in the first metal layer. Data lines connecting distinct units in each bit slice run vertically in the second metal layer. In the second layout architecture (Figure 5.3(c)), a bit slice of each unit consists of one or more standard cells. P and N diffusion strips are placed vertically. Power and ground wires run vertically in the first metal layer. Control lines run over the standard cells in the

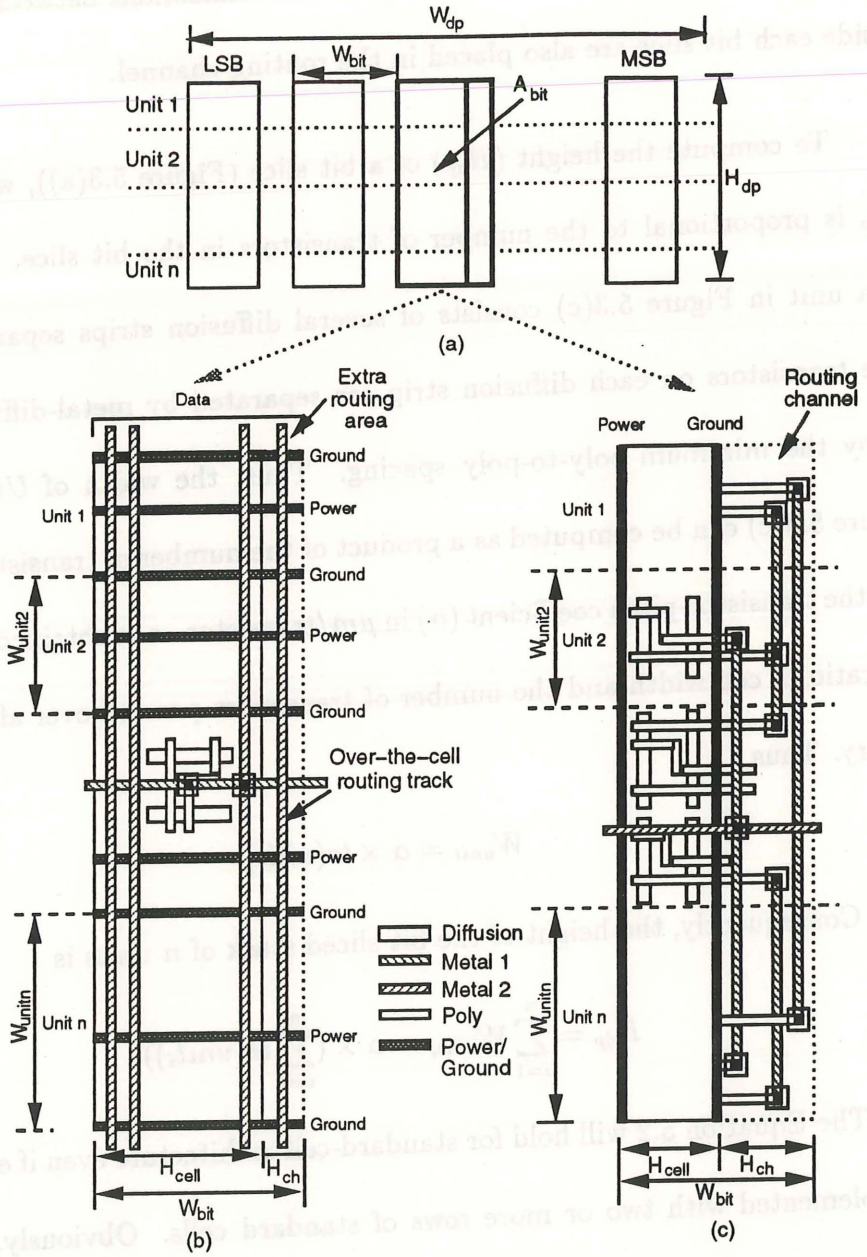


Figure 5.3: The layout models: (a) datapath stack, (b) custom cell architecture, (c) standard cell architecture.

second metal layer. Data lines are placed in the routing channel and run vertically in the first metal or the polysilicon layer. The connections between standard cells inside each bit slice are also placed in the routing channel.

To compute the height (H_{dp}) of a bit slice (Figure 5.3(a)), we observe that H_{dp} is proportional to the number of transistors in the bit slice. Each bit slice of a unit in Figure 5.3(c) consists of several diffusion strips separated by gaps. The transistors on each diffusion strip are separated by metal-diffusion contacts or by the minimum poly-to-poly spacing. Thus, the width of *Unit* (W_{unit}) in Figure 5.3(c) can be computed as a product of the number of transistors ($tr(unit)$) and the transistor-pitch coefficient (α) in μm/transistor. α is obtained by averaging the ratio of cell width and the number of transistors per cell over all units in the library. Thus,

$$W_{unit} = \alpha \times tr(unit). \tag{5.1}$$

Consequently, the height of the bit-sliced stack of n units is

$$H_{dp} = \sum_{i=1}^{n} W_{unit_i} = \alpha \times (\sum_{i=1}^{n} tr(unit_i)). \tag{5.2}$$

The Equation 5.2 will hold for standard-cell architecture even if each bit slice is implemented with two or more rows of standard cells. Obviously, a different coefficient α' must be used in that case. Thus, the height of the bit-sliced stack of n units with an m-row implementation is

$$H_{dp} = \alpha' \times (\sum_{i=1}^{n} tr(unit_i))/m. \tag{5.3}$$

Similar assumptions can be made for custom-cell architecture shown in Figure 5.3(b). Although the P and N strips are placed horizontally in several rows, W_{unit} can be computed by Equation 5.2 using a different transistor-pitch coefficient α''. This assumption holds because the height of the unit slice (H_{cell}) is a constant and the unit width (W_{unit}) thus must reflect the size of the cell in number of transistors.

The width W_{bit} of a bit slice is equal to the sum of the height of the unit slice (H_{cell}) and the height of the routing channel (H_{ch}). For both layout architectures H_{cell} is a constant since all unit slices are predesigned to be of the same height. H_{ch} is calculated as a product of the wire pitch (β), and the difference between the number of estimated routing tracks (Trk_{est}) required to completely connect all nets in one bit slice and the number of available over-the-cell routing tracks (Trk_{top}). Thus,

$$H_{ch} = \begin{cases} 0; & \text{if } Trk_{top} \geq Trk_{est} \\ \beta \times (Trk_{est} - Trk_{top}); & \text{if } Trk_{top} < Trk_{est} \end{cases} \tag{5.4}$$

where Trk_{top} in standard-cell architecture is equal to zero, and coefficient β is equal to the sum of the minimal wire width and the minimal spacing between two metal wires. An estimate for the required number of tracks in each bit slice can be obtained only after the position of each unit in the bit slice is determined. A fast algorithm with pseudo linear time complexity, such as the min-cut algorithm [FiMa82] can be used for this purpose. The required number of tracks can be estimated by the maximum density which is defined as the maximum number of

connections across any cut perpendicular to the channel. A better estimate can be obtained by using some simple routing algorithms, such as the left-edge algorithm [HaSt71] which has $O(n \log n)$ complexity where n is number of nets. Thus, the datapath area (A_{dp}) can be calculated as a product of the number of bits (b_w) and the area of one bit slice, i.e.,

$$A_{dp} = b_w \times H_{dp} \times (H_{cell} + H_{ch}). \tag{5.5}$$

The Equation 5.5 gives an upper bound on the datapath area. The bound is proportional to the product of the number of transistors and the number of routing tracks. The number of transistors can be approximated from the Boolean expressions describing each unit slice or counted from its schematic. The number of tracks can be approximated by the track density after a linear placement. Better estimates can be achieved with algorithms of higher complexity. Since the number of components in the datapath is small, those more accurate estimates are not necessarily computationally intensive.

5.2.2 Control Unit

The control unit in a FSMD model can be described by the control state-table which specifies the next-state and control signals as a function of present states and conditional or status signals in a tabular form. Figure 5.4(a) shows a sample control state table with the input present-state and conditional/status

signals and the output next-state and control signals. The present states are encoded as binary values $p_k...p_1p_0$, where $k \geq \lceil log_2 m \rceil - 1$ and m is the number of states. Similarly, next states are encoded as binary values $r_k...r_1r_0$. Each output c_i controls a functional, storage or interconnection components in the datapath, whereas the outputs r_j specify the next state.

The control unit consists of a state register and control logic. There are two commonly used techniques for implementation of control units: standard cells and programmable logic arrays (PLA). This section describes area estimates for standard-cell and PLA implementations based on sum-of-product expressions of next-state and control signals (Figure 5.4(b)). In reality, a number of optimization procedures, such as logic minimization or PLA folding, are applied in order to reduce the size of the control logic. We ignore the impact of optimization and give an upper bound on the control-unit area.

Standard-Cell Implementation

To simplify area estimates for standard cell implementation, we make a number of assumptions. We assume that a product term in the sum-of-product expression for each output signal includes each present-state signal and (in the worst case) each conditional/status signal; these inputs to the product term may be complemented. We also assume that each product term is implemented by an AND gate

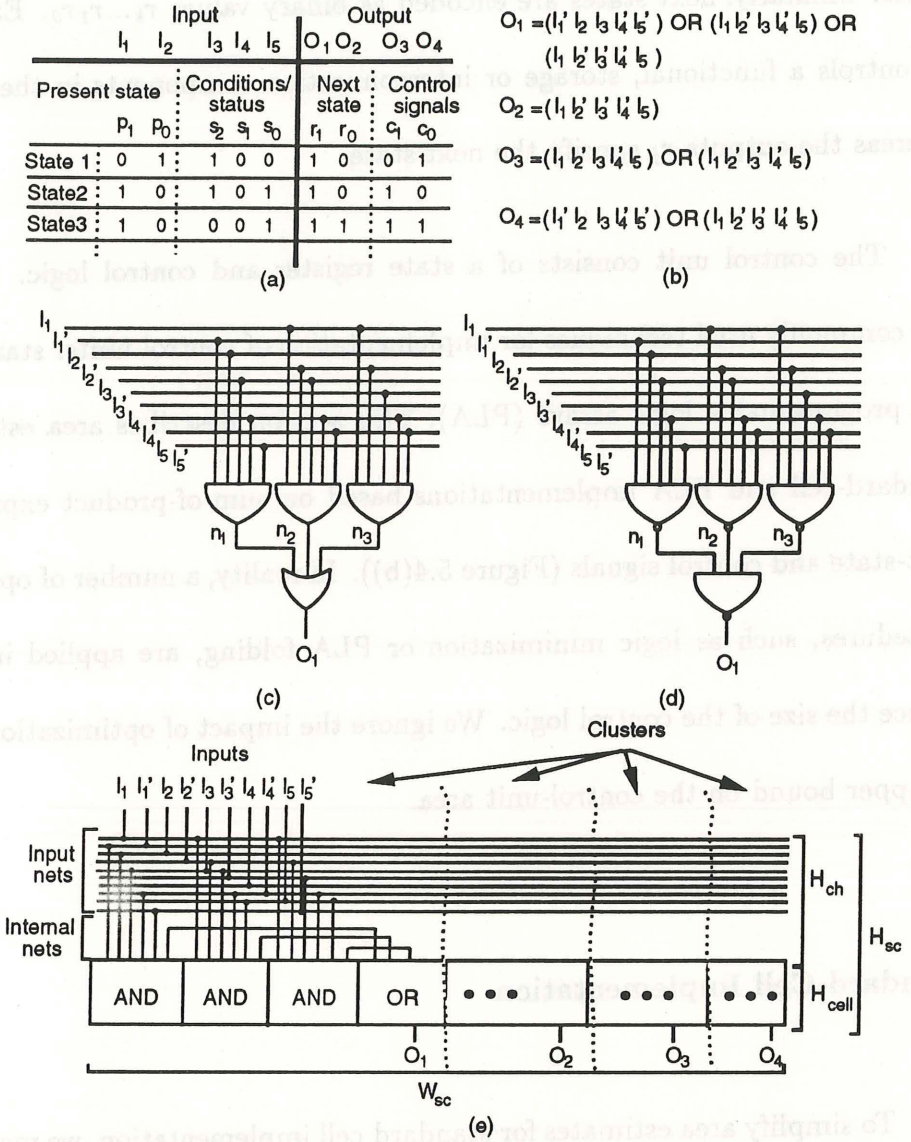


Figure 5.4: Control unit description: (a) state table, (b) Boolean equations for output signals, (c) two-level AND-OR implementation, (d) two-level NAND-NAND implementation, (e) standard cell layout style.

and the sum-of-products by an OR gate, as illustrated in Figure 5.4(c) for output O_1. Obviously, we can replace the AND-OR implementation with an equivalent NAND-NAND implementation as shown in Figure 5.4(d).

We assume that all the gates for implementation of the control unit are placed in a single row of cells, the inputs appear at the top, and the outputs appear at the bottom, as shown in Figure 5.4(e). We also assume that all the gates needed for the implementation of an output signal are clustered together as shown for signal O_1 in Figure 5.4(e). This requirement for strong clustering prevents sharing of an AND gate between two expressions with the same product term. The layout area of the control unit using the standard-cell implementation (A_{sc}) is then equal to the product of width (W_{sc}) and height (H_{sc}), that is,

$$A_{sc} = W_{sc} \times H_{sc}, \tag{5.6}$$

where W_{sc} is proportional to the number of transistors and H_{sc} is proportional to the number of routing tracks. The number of transistors can be computed from the sum-of-product expression for each output signal.

In CMOS technology, each n-input AND or OR gate has $2n + 2$ transistors. Note that n-input NAND and NOR gates need only $2n$ transistors. Since each product term in a sum-of-product expression is implemented with one AND gate and one OR gate (e.g., Figure 5.4(c)), we can compute the required number of transistors. Each literal in a product term contributes 2 transistors, each product

term contributes 2 transistors in the AND gate and 2 transistors in the OR gate and the OR gate contributes an additional 2 transistors.

Let $occur(O_i)$ and $term(O_i)$ be the number of occurrences of literals and number of terms in the sum-of-product expression of the signal O_i, respectively. Let $tr(Reg)$ be the number of transistors in one-bit state register. Thus, for an AND-OR implementation of the control unit, the width of the control unit is computed as

$$W_{sc} = \alpha \times \quad ((\sum_{i=1}^{\lceil log_2 m \rceil + n} (2term(O_i)occur(O_i) + 4term(O_i) + 2)) + (\lceil log_2 m \rceil \times tr(Reg))), \tag{5.7}$$

where m is the number of states and n is the number of control signals and α is the transistor-pitch coefficient as defined in Section 5.2.1.

The height H_{sc} is computed as the sum of the standard-cell height (H_{cell}) and the channel height (H_{ch}). H_{ch} is proportional to the number of tracks used by input signals and internal nets connecting AND and OR gates. We assume that each input signal requires two tracks for the true and complemented values. We assume that there are $\lceil log_2 m \rceil$ state signals and C conditional/status signals. The maximum number of tracks required for routing internal nets in each cluster is equal to the number of terms in the sum-of-product expression for a particular output signal. Since clusters do not overlap, the maximum number of required tracks to route all internal nets is equal to the largest number of terms used in any

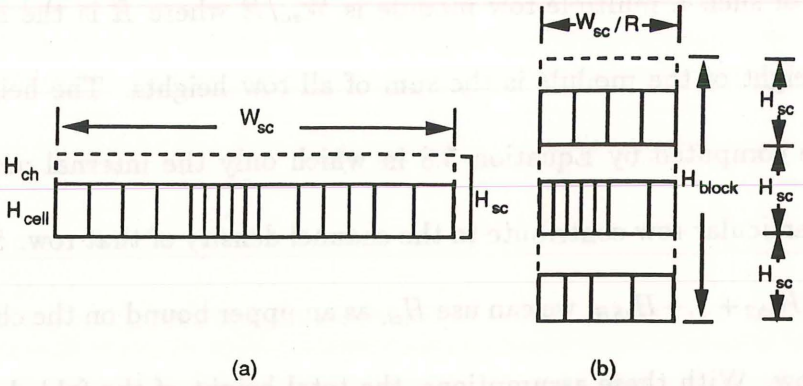


Figure 5.5: Different aspect ratios of the control logic: (a) one-row implementation, (b) three-row implementation.

particular sum-of-product expressions of an output signal. Thus, the height of the control unit H_{sc} is

$$H_{sc} = H_{cell} + \beta \times (2(\lceil log_2 m \rceil + C) + MAX(\forall_{i=1}^{\lceil log_2 m \rceil + n} term(O_i))), \qquad (5.8)$$

where β is the wire pitch in the channel.

Although we assumed a single-row layout, control logic modules with different aspect ratios are needed in reality. Different aspect ratios can be obtained by laying the control logic module in several rows. We approximate this folding process by assuming that we can evenly partition the single-row layout of Figure 5.5(a) into three rows of equal width as shown in Figure 5.5(b). We assume that input and output pins are positioned at the top and bottom of the control unit. Further, we assume that inputs reach all rows using existing polysilicon lines in each row as feedthroughs, and output nets are routed vertically (over the cells) in metal2. We also assume that output clusters are not split across folded rows. Then, the

width of such a multiple-row module is W_{sc}/R where R is the number of rows. The height of the module is the sum of all row heights. The height of each row can be computed by Equation 5.8 in which only the internal nets of clusters in that particular row contribute to the channel density of that row. Since $H_{ch} \times R \geq H_{ch1} + H_{ch2} + ... + H_{chR}$, we can use H_{ch} as an upper bound on the channel height for each row. With these assumptions, the total height of the folded implementation (H_{block}) is $H_{sc} \times R$, while its width is W_{sc}/R. Hence, we can still use the Equation 5.6 as an estimate for control-logic layouts of different aspect ratios.

Programmable Logic Array

A programmable logic array (PLA) is frequently used to implement combinatorial and sequential logic, and in particular, the control units of FSMDs. A PLA consists of AND and OR arrays supported by input and output buffers, input and output latches and product term buffers as shown in Figure 5.6. Input buffers are needed to drive the AND array, product term buffers to drive the OR array and output buffers to drive the external logic. The input and output latches are used for the sequential logic.

The width of the PLA module (W_{PLA}) is the sum of the width of the input AND array (W_{in}), the width of product-term buffers (W_p) and the width of the OR array (W_{out}) (Figure 5.6(b)). W_{in} equals the number of inputs (n) multiplied

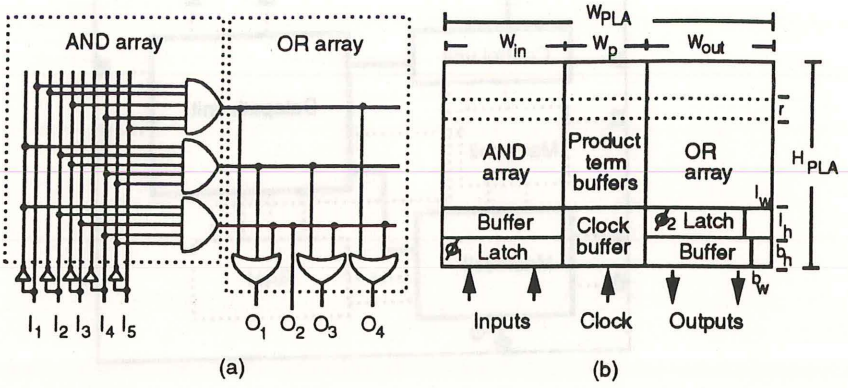


Figure 5.6: PLA layout model: (a) logic mapping, (b) layout model.

by the the maximum of the latch width (l_w) and the buffer width (b_w). Similarly, W_{out} equals the product of $MAX(l_w, b_w)$ and the number of outputs (m).

The height of the PLA (H_{PLA}) is computed as a sum of the latch height (l_h), buffer height (b_h) and the height of the AND-OR plane. The height of the AND-OR plane is determined by the product of the number of distinct product terms (p), and the transistor-row pitch (r). Thus, the area of a PLA is

$$A_{PLA} = (((n + m) \times MAX(l_w, b_w)) + W_p) \times (l_h + b_h + r \times p). \quad (5.9)$$

5.3 Performance Measures

We model the chip layout as a set of connected blocks that include control units, datapaths, macrocells and memories, as shown in Figure 5.7. Typically, a datapath consisting of a set of regularly structure components is implemented

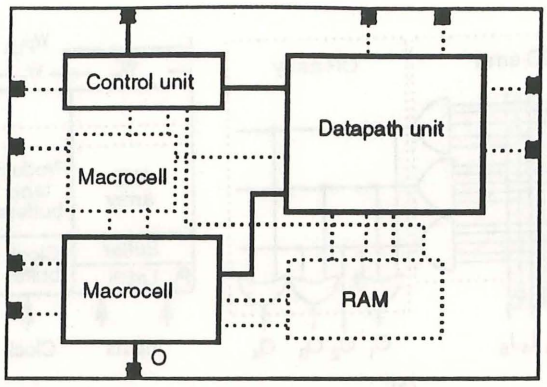


Figure 5.7: Constituents of a chip.

using a bit-sliced stack, standard cells or macrocells. A control unit is implemented using a PLA or standard cells. Macrocells include some predefined components such as multipliers and barrel shifters. Memories include register files, RAMs and ROMs.

The remainder of this section is organized in the following manner. Section 5.3.1 describes the electrical models including wiring and component delays. Section 5.3.2 describes the delay model for each block, as shown in Figure 5.7. Since macrocells and memories are usually available in a library as predesigned blocks, we assume that timing information for macrocells and memories is provided by the library. Section 5.3.3 and 5.3.4 describe the delay models for the datapath and the control unit. Section 5.3.4 describes the inter-block wiring delay model. Finally, Section 5.3.5 describes the clock-period model.

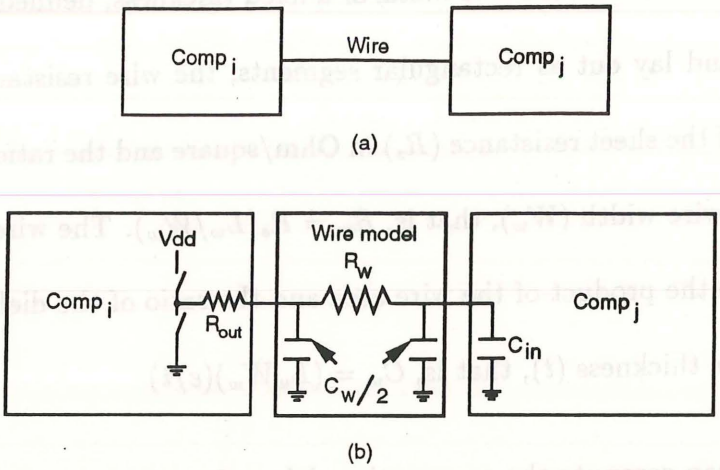


Figure 5.8: Wire: (a) RT model, (b) equivalent RC delay model.

5.3.1 Electrical Models

The lumped RC model, also called the Elmore delay model [PeRu81], is widely used for delay calculation. In this model, the propagation delay along a path from the start point to the end point (t_p(start,end)) is computed as a product of lumping all of the resistances R_j and capacitances C_k along the path, that is,

$$t_p(\text{start}, \text{end}) = \sum_j R_j \times \sum_k C_k. \tag{5.10}$$

We can use Equation 5.10 to obtain the delay of a connecting wire between two components as shown in Figure 5.8(a), or between two blocks as shown in Figure 5.7. In CMOS technology we model a component as having input capacitance (C_{in}) and output resistance (R_{out}), as shown in Figure 5.8(b). For the connecting wire, we use the well known π-model that models a wire as an input capacitance ($C_w/2$), wire resistance (R_w) and the output capacitance ($C_w/2$).

Since a wire is a thin sheet of metal of a fixed thickness, defined by the fabrication process, and lay out as rectangular segments, the wire resistance is equal to the product of the sheet resistance (R_s) in Ohm/square and the ratio of the wire length (L_w) and wire width (W_w), that is, $R_w = R_s(L_w/W_w)$. The wire capacitance (C_w) is equal to the product of the wire area and the ratio of the dielectric constant (ε) to the wire thickness (t), that is, $C_w = (L_w W_w)(\varepsilon/t)$.

We can compute the propagation delay of a wire net net_k ($t_p(net_k)$) used by a component ($comp_i$) to drive load components ($comp_j$, $1 \leq j \leq n$) as

$$t_p(net_k) = (R_{out}(comp_i) + R_w)(C_w + \sum_{j=1}^{n} C_{in}(comp_j)). \tag{5.11}$$

Thus, the delay for signals to propagate from the input of $comp_i$, through net_k, to one of $comp_i$'s driven-components $comp_j$, is

$$t_p(comp_i, net_k, comp_j) = t_p(comp_i) + t_p(net_k). \tag{5.12}$$

where $t_p(comp_i)$ is the internal delay of component $comp_i$.

5.3.2 Datapath Delay Model

To compute the propagation delay from one datapath component to another in the same datapath block requires two elements: internal delay of the component and wiring delay. Typically, the internal delays of components are provided by the targeted component library.

The actual wiring length can be determined only after the completion of computational expensive datapath placement and routing procedures. For simplicity, we assume that the average wire length of a net connecting any two units in the same datapath is equal to half of the datapath height (H_{dp}) (Figure 5.2). In the first layout architecture, H_{dp} is equal to the sum of the height of all datapath units. Whereas, in the second layout architecture, H_{dp} is proportional to the number of transistors in the bit slice and the transistor pitch as described in Section 5.2.1. Thus, the average wiring resistance and capacitance are calculated as: $R_{w(DP)} = R_s((1/2H_{dp})/W_w)$ and $C_{w(DP)} = C_s(1/2H_{dp})(W_w)$, where W_w is the width of the metal1 wire for the first layout architecture and the width of the metal2 wire for the second layout architecture. Thus, the propagation delay between two datapath components ($comp_i$ and $comp_j$) via a net (net_k) is computed using Equations 5.11 and 5.12.

5.3.3 Control Delay Model

There are two commonly used layout architectures for a control unit: random logic and PLA. Since the timing information for a PLA is usually provided by its generator, in this section I describe the random-logic timing-model for a control unit.

In the control-unit model as described in Section 5.2.2, each next-state and control signal is represented as sum of products of the present-state and conditional/status signals, as shown in Figure 5.4(b). The product term is implemented with AND gates and the sum with OR gates. However, the target component library will usually provide AND and OR gates with a limited number of inputs. Thus, to realize the impact of the technology mapping, the sum and product terms need to be decomposed into a multi-level implementation when the large AND or OR gates are not available in the target library.

The multi-level decomposition aims to produce an implementation with the minimum number of levels. This is guided by the fact that a multi-level implementation of a product term with I number of literals using AND gates with a maximum of n inputs is in the form of an n-ary tree [ChWG91]. Similarly, the same decomposition scheme can be used to obtain a multi-level OR implementation of the sum term.

The capacitive load of each control signal, C_{CUload}, that drives the datapath units is proportional to the size (bit-width) of the datapath. If C_{CUload} is high, buffers are usually inserted to reduce the delay caused by the heavy load. Let us examine the loading effect in our model. If the buffer is not inserted, the last OR gate (i.e., the gate that is represented by the root of the OR tree) has to drive C_{CUload}. Thus, the delay caused by this load equals $R_{out}(OR(m)) \times C_{CUload}$, where m is a maximum number of inputs of an OR gate available in the library. However,

if a buffer, BUF, is inserted, the delay caused by the load and the additional buffer equals $((R_{out}(BUF) \times C_{CUload}) + t_p(BUF))$. Therefore, to realize the influence of buffers insertion, we assume that each output of the control logic is driven by a buffer, BUF, if $(R_{out}(BUF) \times C_{CUload}) + t_p(BUF) - (R_{out}(OR(m)) \times C_{CUload}) < 0$.

Figure 5.9 shows an example of a multi-level implementation of a sum-of-products expression. Each product term in the sum-of-products expression, Figure 5.4(b), requires a 5-input AND gate. If the targeted library provides only AND gates with a maximum of three inputs, all product terms are decomposed into a multi-level implementation, which is represented by a trinary tree shown in Figure 5.9(a). The equivalent gates implementation of this trinary tree is shown in Figure 5.9(b).

In our model, we assume that the random-logic is laid out as strips of standard or custom cells with input ports entering at the top and output ports exiting through the bottom. The number of layout strips is predetermined by the floor-planner in such a way that the total chip area is minimized. In addition, we assume that all gates that implement an output signal are placed closely in a cluster, as shown in Figure 5.9(c). The propagation delay from any input port of the control logic to an output port O_i consists of two elements: the gate delay $(t_p(gate(O_i)))$ and the wire delay $(t_p(net(O_i)))$.

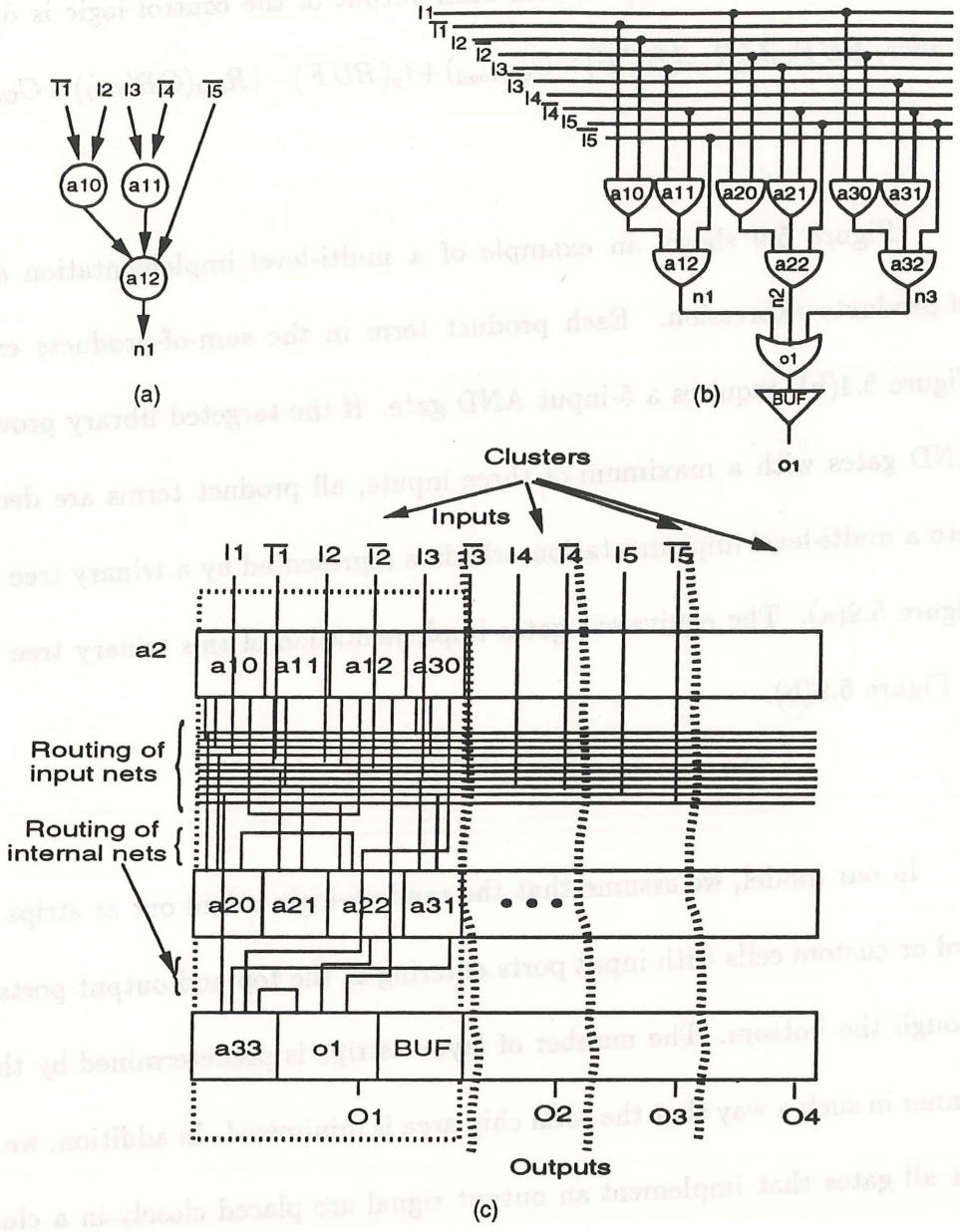


Figure 5.9: Random-logic model: (a) decomposition of a product term, (b) a multi-level implementation, (c) the layout model.

Gate delay is defined as the sum of delays of gates along the critical path. Using the decomposition scheme described earlier, the gate delay ($t_p(gate(O_i))$) can be formulated as the sum of the delay of gates in the AND tree, OR tree and the output buffer, that is,

$$t_p(CU(O_i)) = (AND_{I\text{-}node} \times t_p(AND(n))) + (OR_{I\text{-}node} \times t_p(OR(m))) + t_p(BUF) \tag{5.13}$$

where

$t_p(AND(n))$ is the propagation delay of an n-input AND gate, and

$t_p(OR(m))$ is the propagation delay of an m-input OR gate.

Wire delay, $t_p(net(O_i))$, is defined as the sum of delay of wires on the critical path. Using our layout model, wires that connect gates in the same cluster are relatively short. Thus, the wiring resistance and capacitance of these nets are negligible. Hence, the wiring delay $t_p(net(X,Y))$ of a net that connects the output of a gate of type X to an input of a gate of type Y in the random logic can be derived from Equation 5.11 with R_w and C_w equal to 0, that is, $t_p(net(X,Y)) = R_{out(X)} \times C_{in(Y)}$.

Using properties of the decomposition tree, $t_p(net(O_i))$ can be formulated as the sum of the wiring delays of nets in the AND tree, OR tree, the net that connects the AND and OR tree, and the net that connects the last OR gate (i.e., the OR gate that is represented by the root of OR tree) to the output buffer, that

is,

$$\begin{aligned} t_p(net(O_i)) &= (AND_{I\text{-}net} \times t_p(net(AND(n), AND(n)))) + \\ & (OR_{I\text{-}net} \times t_p(net(OR(m), OR(m)))) + \\ & t_p(net(AND(n), OR(m))) + t_p(net(OR(m), BUF)). \end{aligned} \quad (5.14)$$

Thus, the propagation delay from any input port to an output port O_i is

$$t_p(CU(O_i)) = t_p(gate(O_i)) + t_p(net(O_i)). \quad (5.15)$$

5.3.4 Inter-Block Wiring Delay Model

Similarly, we can use Equation 5.11 to compute the propagation delay of a net net_k that connects two blocks A and B on a chip. To obtain the wire length of the net, we use a simple cluster-growth algorithm to form the chip floorplan. In summary, the cluster is grown from the lower left corner and the algorithm iteratively adds blocks on the top and right. The main reason for using this simplied method is the considerably low computation effort. The algorithm determines the order of the block to be placed according to the cost of the resultant area and the connectivity of that block with those already placed. In another word, the objective function selects a block $B_i \in B_{unplace}$ with $max(Area(B_i)) \times \Sigma w(B_i, B_j)$, where $B_j \in B_{place}$ and w is the number of wires. Subsequently, a placement position for the selected block is determined with respect to the overall connectivity and the aspect-ratio constraints.

As a result of the floorplan, each block in the chip is centered at a coordinate (x, y). The length of a net connecting any two blocks A and B is estimated as the manhattan distance between the centers of the two blocks. Thus, the inter-block wiring resistance and capacitance are

$$R_w(net_k) = (\frac{|A_x - B_x|}{W_w} \times R_s) + (\frac{|A_y - B_y|}{W_w} \times R_s) \tag{5.16}$$

$$C_w(net_k) = (|A_x - B_x| \times (W_w) \times C_s) + (|A_y - B_y| \times (W_w) \times C_s) \tag{5.17}$$

where

W_w is the width of the routing wire,

A_x and A_y are the x and y coordinates of the block A,

B_x and B_y are the x and y coordinates of the block B.

The propagation delay of net_k is then computed using Equation 5.11.

5.3.5 Clock Cycle Model

The clock cycle is determined by the worst register-to-register delay that includes the propagation delays in the control unit, in the datapath unit and between blocks. In our implementation, the clock computation is based on the FSMD, as shown in Figure 5.10 and described below.

In Figure 5.10, the critical path (*Path1*) is from the *State register*, through the *Control logic*, *Datapath*, and *Next-state logic,* and back to the *State register.*

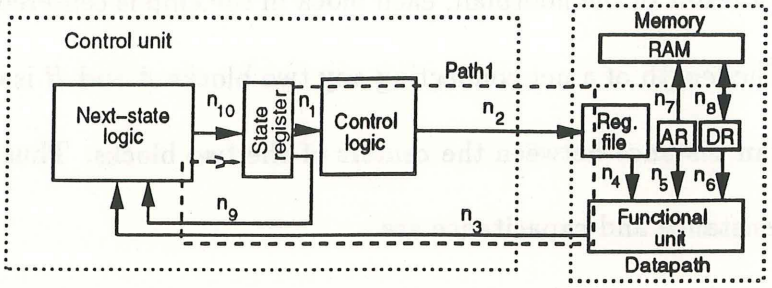


Figure 5.10: FSMD clocking model.

Thus, the clock period is the sum of the propagation delays of the *State register* ($t_p(State\ register)$), the *Control logic* ($t_p(Control\ logic)$), the *Datapath* ($t_p(DP)$), the *Next-state logic* ($t_p(Next\text{-}state\ logic)$), and the set-up delay of the *State register* ($t_{setup}(State\ register)$), that is,

$$\begin{aligned} t_{clock} &= t_p(State\ register) + t_p(Control\ logic) + t_p(DP) \\ &\quad + t_p(Next\text{-}state\ logic) + t_{setup}(State\ register). \end{aligned} \tag{5.18}$$

$t_p(Control\ logic)$ and $t_p(Next\text{-}state\ logic)$ are computed using Equation 5.15. $t_p(DP)$ is determined by the worst register-to-register delay in the datapath. For

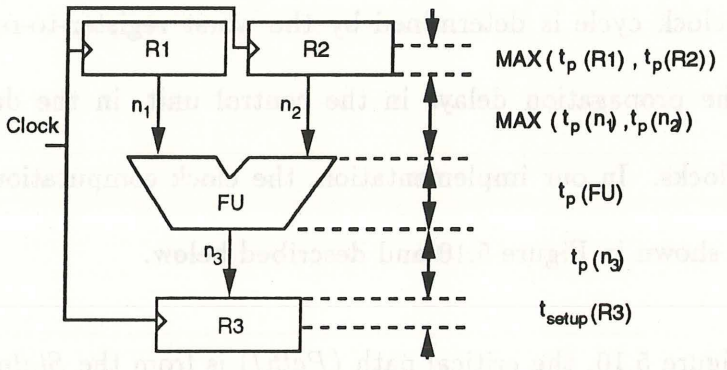


Figure 5.11: The register-transfer path.

example, a typical register-to-register delay in a datapath is shown in Figure 5.11. It includes the delay through the source storage-unit ($MAX(t_p(R1), t_p(R2))$), functional unit ($t_p(FU)$), connections ($MAX(t_p(n_1), t_p(n_2))$ and $t_p(n_3)$) and the setup time of the destination storage-unit ($t_{setup}(R3)$). Since the access time to a RAM is slow and often takes several clock cycles, we consider the storage units, *R1, R2* and *R3*, as registers or a register file. Connections n_1, n_2 and n_3 are implemented as wires or interconnect units such as muxes or buses. Thus, for a single-cycle operation op_i or a single-cycle chaining operation, the register-to-register delay of operation op_i is computed as

$$\begin{aligned} t_p(op_i) &= MAX(t_p(R1), t_p(R2)) + MAX(t_p(n_1), t_p(n_2)) \\ &\quad + t_p(FU) + t_p(n_3) + t_{setup}(R3) \end{aligned} \tag{5.19}$$

and for n-cycle operation op_i, the register-to-register delay per clock-cycle of operation op_i is

$$\begin{aligned} t_p(op_i) &= (MAX(t_p(R1), t_p(R2)) + MAX(t_p(n_1), t_p(n_2)) \\ &\quad + t_p(FU) + t_p(n_3) + t_{setup}(R3))/n. \end{aligned} \tag{5.20}$$

Thus, the worst register-to-register delay for each clock-cycle among all operations is

$$t_p(DP) = MAX(\forall_i t_p(op_i)). \tag{5.21}$$

5.4 Results

The experiments consist of two parts: area estimation and timing estimation. Section 5.4.1 describes the experiments on the area model. Section 5.4.2 describes the experiments on the timing model.

5.4.1 Area Measure

We have tested our layout models on 4 designs with 16 different implementations of the elliptic filter benchmark (Figure 5.12). Each implementation uses different number of registers and muxes. The α coefficient was calculated based on the VTI 1.5-μm datapath library [VTI88]. The final layouts were generated using Mentor Graphics GDT tools. The layout architectures used in Figure 5.12 and Figure 5.13 correspond to those described in Section 5.2.1, in which the Layout Architecture I uses 13 over-the-cell routing tracks for each bit slice. Since the multiplier is treated as a macrocell, its area remains constant throughout all the examples, and is not included in the results. Figure 5.14 show that 64 area measures using different combinations of layout architectures, muxes or buses (one tri-state buffer for each mux input). The results show that 90% of the estimates are within 90% accuracy.

** Area not including multiplier

Design	# of Reg.	# of Mux. / # Mux Inputs	#trs.	#nets	#trks. Actual (est.)	Layout Architecture I			Layout Architecture II		
						Est. Area (um^2 / bit)	Actual Area (um^2 / bit)	Est. / Actual	Est. Area (um^2 / bit)	Actual Area (um^2 / bit)	Est. / Actual
A	10	11 / 34	552	27	11(15)	129,680	136,720	0.95	213,625	193,117	1.11
	11	10 / 33	564	27	11(12)	124,080	138,080	0.90	200,216	195,038	1.03
	12	8 / 31	572	27	11(12)	125,840	138,480	0.91	200,796	195,490	1.03
	13	9 / 33	604	28	10(15)	143,653	145,040	0.98	226,625	199,430	1.14
B	10	8 / 30	472	23	10(11)	103,840	113,440	0.92	166,234	156,420	1.03
	11	6 / 28	480	22	9(10)	105,600	113,760	0.93	156,420	151,726	1.03
	12	6 / 28	500	23	9(11)	110,000	117,280	0.94	165,658	156,862	1.06
	13	6 / 29	524	24	9(10)	115,280	122,080	0.94	167,860	163,282	1.03
C	10	7 / 30	480	20	8(11)	105,600	113,536	0.93	160,369	146,464	1.09
	11	5 / 27	480	19	10(12)	105,600	111,456	0.95	161,611	151,836	1.06
	12	5 / 28	504	19	9(11)	110,880	115,296	0.96	162,855	152,934	1.06
	13	6 / 31	540	23	9(11)	118,800	126,736	0.94	179,014	168,235	1.06
D	10	10 / 36	508	23	10(11)	111,760	125,376	0.89	177,093	170,976	1.04
	11	6 / 28	482	20	9(11)	106,040	113,136	0.94	159,804	150,045	1.07
	12	6 / 26	492	21	8(10)	108,240	115,696	0.94	159,082	149,272	1.07
	13	5 / 23	490	21	8(11)	107,800	113,696	0.95	158,595	146,672	1.09

A : 17–step, 3–adder, 2–piped multipliers.
B : 19–step, 2–adder, 2–multiplier.
C : 21–step, 2–adder, 1–multiplier.
D : 19–step, 2–adder, 1–piped multiplier.

Figure 5.12: The datapath area estimates of the elliptic filter example with mux implementation.

** Area not including multiplier

Design	# of Reg.	# of Mux. / # Mux Inputs	#trs.	#nets	#trks. Actual (est.)	Layout Architecture I: Est. Area (um² / bit)	Layout Architecture I: Actual Area (um² / bit)	Layout Architecture I: Est. / Actual	Layout Architecture II: Est. Area (um² / bit)	Layout Architecture II: Actual Area (um² / bit)	Layout Architecture II: Est. / Actual
A	10	11 / 34	776	26	11(15)	182,888	162,240	1.13	259,600	229,164	1.13
	11	10 / 33	784	28	10(14)	174,526	163,680	1.07	255,750	225,060	1.14
	12	8 / 31	780	28	9(13)	171,600	162,720	1.05	242,063	217,638	1.11
	13	9 / 33	824	29	8(12)	181,280	168,960	1.07	244,976	219,648	1.10
B	10	8 / 30	672	21	8(10)	147,840	136,960	1.08	199,176	172,912	1.15
	11	6 / 28	688	22	7(9)	151,360	136,000	1.11	197,260	171,700	1.15
	12	6 / 28	688	23	9(10)	151,360	139,840	1.08	203,800	187,036	1.09
	13	6 / 29	720	24	8(10)	158,400	146,080	1.08	200,860	189,904	1.06
C	10	7 / 30	672	20	10(10)	147,840	136,896	1.08	199,176	186,816	1.07
	11	5 / 27	656	19	8(8)	144,320	133,536	1.08	184,380	172,464	1.07
	12	5 / 28	688	20	7(8)	151,360	137,376	1.10	192,572	172,446	1.10
	13	6 / 31	744	23	9(8)	163,680	153,216	1.07	209,644	203,652	1.03
D	10	10 / 36	744	21	10(13)	163,680	148,896	1.10	225,099	203,316	1.10
	11	6 / 28	668	19	7(9)	146,960	133,536	1.10	191.706	167,598	1.14
	12	6 / 26	664	20	8(9)	146,080	132,576	1.10	196,824	171,216	1.11
	13	5 / 23	648	20	8(8)	142,560	129,216	1.10	181,324	166,848	1.09

A : 17–step, 3–adder, 2–piped multipliers.
B : 19–step, 2–adder, 2–multiplier.
C : 21–step, 2–adder, 1–multiplier.
D : 19–step, 2–adder, 1–piped multiplier.

Figure 5.13: The datapath area estimates of the elliptic filter example with bus implementation.

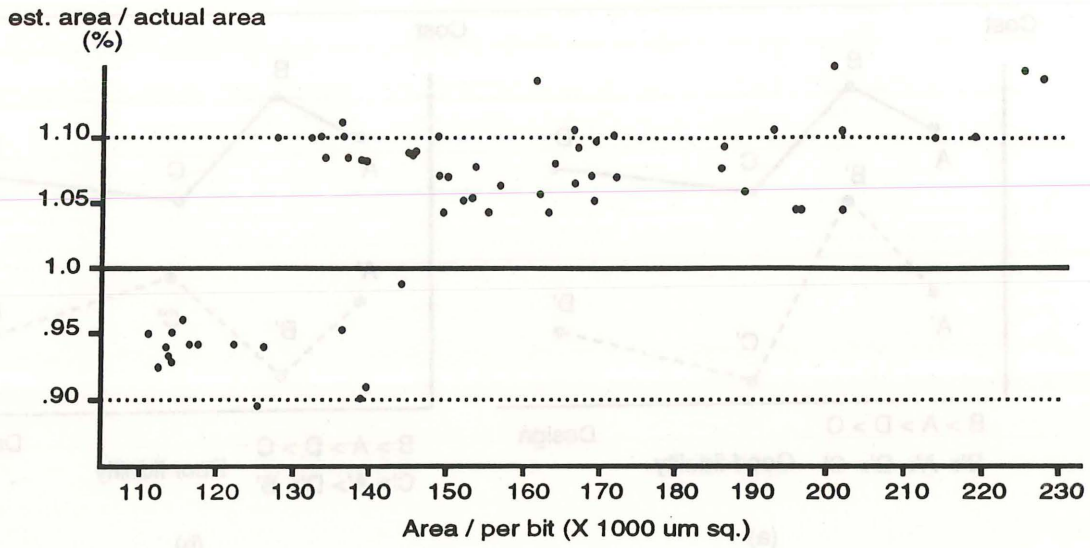


Figure 5.14: The accuracy analysis of the datapath area estimates.

We have investigated the "fidelity" of our area estimates. Fidelity is another crucial factor in the quality measure that indicates the degree of the estimated results correspond to the actual results. In the other words, fidelity is the deviation from the average error over all design points. If the error over all design points is always of the same magnitude then fidelity is high. For instance, Figure 5.15 shows two examples, in which solid line represents the actual results while dash line represents the the estimated results. Figure 5.15(a) shows that the estimates well predict the actual results; that is, if we have to select the minimum-cost design then design C will be selected since the estimate C' predicts the minimum cost. Thus, the estimates in Figure 5.15(a) show high fidelity. On the other hand, the estimates in Figure 5.15(b) show poor fidelity since design B will be selected as

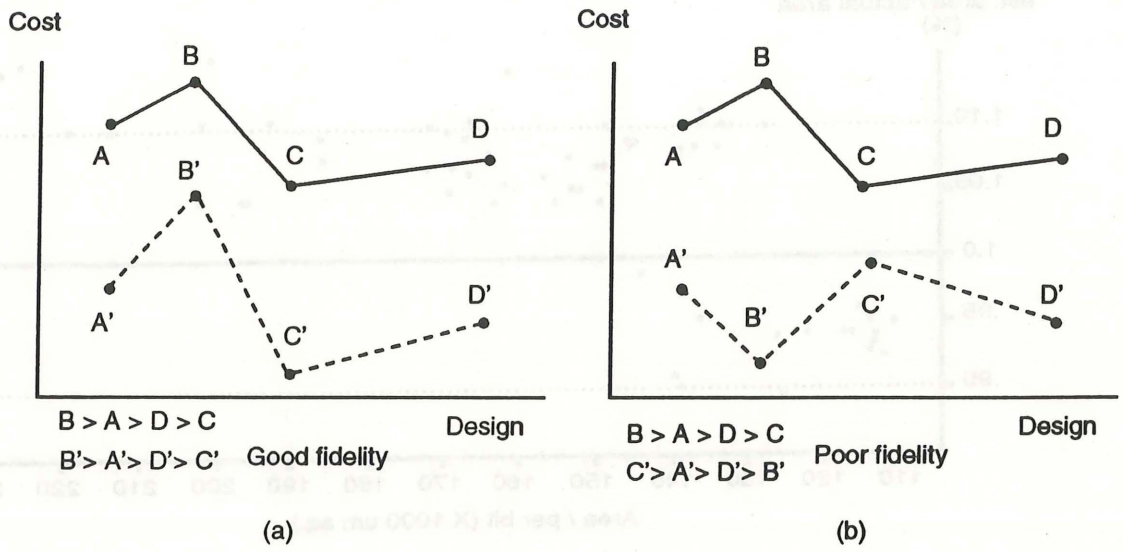


Figure 5.15: The fidelity analysis: (a) good fidelity, (b) poor fidelity.

the minimum-cost design according to the estimate B'. However, design B has the highest cost.

We compare the "fidelity" of 7 different metrics, namely metric #1, 2, 3, 4, 5, 6, and 7 (Figure 5.16). The numbers in Figure 5.16 represent the percent difference between the area of predicted minimal area implementation and actual minimal area of the design.

For each design, we first choose the minimum cost implementation according to different metrics. For example, based on metric #5, we choose the implementation with the minimum number of transistors as the best implementation (Figure 5.12). For design D, the implementation with 11 registers, 6 muxes, 28

Metrics	Quality mesures	Percent difference between the predicted best implementation and the actual minimum area of the design							
		Layout architecture I				Layout architecture II			
		A	B	C	D	A	B	C	D
1	# Register	0	0	1.87	10.82	0	3.09	0	16.60
2	# Mux input	1.27	0.28	0	0.50	1.22	0	3.67	0
3	# Equivalent 2:1mux	0	0	0	0.50	0	3.09	3.67	0
4	# Register + # Equivalent 2:1mux	0	0	0	0.50	0	3.09	3.67	0
5	# transistor	0	0	0	0	0	3.09	3.67	2.30
6	# Register+# Mux input+ # Unique net	1.29	0.28	0	0.50	1.23	0	3.67	0
7	Our layout model	1.0	0	0	0	1.0	0	0	0

(a)

Metrics	Quality mesures	Percent difference between the predicted best implementation and the actual minimum area of the design							
		Layout architecture I				Layout architecture II			
		A	B	C	D	A	B	C	D
1	# Register	0	0.7	2.5	15.2	5.3	0.7	8.3	21.8
2	# Mux input	0.3	0	0	0	0	0	0.01	0
3	# Equivalent 2:1mux	0.8	0	0	0	5.3	8.9	0.01	0
4	# Register + # Equivalent 2:1mux	0	2.5	2.5	0	5.3	0.7	0.01	0
5	# transistor	0	0	0	0	5.3	0	0.01	0
6	# Register+# Mux input+ # Unique net	0	0	0	0	5.3	0.7	0.01	0
7	Our layout model	0.3	0	0	0	0	0	0.01	0

(b)

Figure 5.16: Comparative study of the elliptic filter example with different design quality measures: (a) mux implementation, (b) bus implementation.

mux inputs (482 transistors) is chosen as the best implementation in which the areas are 113,136μm^2 and 150,045μm^2 using layout architectures I and II, respectively. On the other hand, the actual minimal areas for design D are 113,136μm^2 and 146,672μm^2. For design D with layout architecture I, the transistor metric (metric #5) accurately predicts the minimum area. Since the percent difference between the area of predicted best implementation and the actual minimum area is 0, the entry for metric #5 and design D with layout architecture I in Figure 5.16(a) is 0. On the other hand, for design D with layout architecture II, the area of predicted best implementation is 2.3% (150,045μm^2 vs. 146,672μm^2) larger than the actual minimum area of the design. Hence, the number in Figure 5.16(a) is 2.3.

Since all implementations using layout architecture I use less than 13 actual routing tracks, they do not require any extra routing tracks. Hence, the area of the datapath is solely dependent on the number of transistors. This is the reason why metric #5 can predict the minimum area implementations on all designs using layout architecture I. Metrics #1, 2, 3 and 4 give poor predictions because register and mux counts alone will not accurately predict total number of transistors in the datapath. Metric #6 also gives poor predictions because this metric considers wiring area in terms of number of unique nets, which is absent in this case. Our layout model (metric #7) shows accurate predictions except for the design A due to over-estimation in the number of routing tracks caused by our simple linear placement method.

Using layout architecture II, both transistors and routing tracks make equal contribution to the total area. Hence, design quality measures which do not consider routing tracks, for example metrics #1, 2, 3, 4 and 5, do not predict layout area well. Metric #6 does not do well because the number of unique nets does not directly indicate the number of routing tracks. As for our layout model, the results show consistent fidelity.

In addition, we have estimated the total area (including datapath, control, and multiplier) of the elliptic filter benchmark with a 16-bit, 19-step, 2-adder and 1-piped multiplier. We have implemented two control-logic models, PLA and random logic. The result shows that our layout models can predict: the datapath area with 10% error, the PLA area with 18% error and the random logic area with 16% error.

5.4.2 Timing Measure

We have tested our timing models on the elliptical filter benchmark. The experiment is divided into three parts. First, we compare our timing models for clock-period estimation against traditional performance measures by comparing estimates with the actual timing. The main distinction between different performance-estimation schemes is the granularity of the underlying model. A realistic timing model should consider all delay constituents of a chip. In Section

5.3, we have provided timing models for each of these constituents. Second, we determine the percentage contributed by these constituents to show that each of the constituent does in fact contribute delay to the clock period. The amount of delays contributed by each constituent of a chip varies across designs. Fourth, we show that estimates from timing models can be used to guide behavioral synthesis tools in the selection of design styles.

In the experiments, the clock period is computed using Equation 5.18. For simplicity, we divided the delay of the clock period into three parts: *Datapath* delay, *Control unit* delay and *Wire/load* delay. *Datapath* delay includes the delays of wiring, functional, interconnect and storage units as described in Equation 5.19. *Control unit* delay includes the delays of control logic, next-state logic and state register as described in Equation 5.15. Wire/load delay takes into account the global wiring delay and the overall driven-load. The first and second experiments are carried out in 3μm CMOS technology [GDT89], while the third experiment uses an 1.5μm CMOS technology [VTI88].

In the first and second experiments, we have tested our control timing models on four synthesized designs of the elliptical filter benchmark with 2 adders and a 2-stage pipeline multiplier. The delay calculation is based on a 16-bit datapath and a 3μm CMOS technology. All four designs are scheduled in 19 control steps but with different utilization of registers and muxes (Figure 5.17): (1) design A contains 10 registers and 36 mux-inputs, (2) design B contains 11 registers and 28

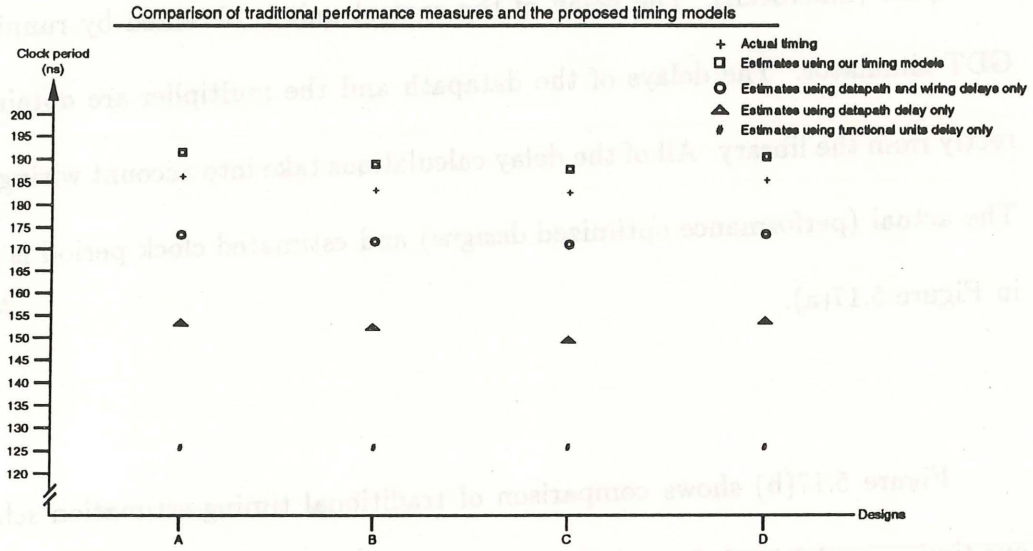
mux-inputs, (3) design C contains 12 registers and 26 mux-inputs, and (4) design D contains 13 registers and 23 mux-inputs.

Using the layout area model described in Section 5.2, the elliptic-filter benchmark is laid out in three blocks: a control unit, a datapath and a 2-stage pipelined multiplier (macrocell). The delay of the control unit is obtained by running the GDT simulator. The delays of the datapath and the multiplier are obtained directly from the library. All of the delay calculations take into account wiring delay. The actual (performance optimized designs) and estimated clock period is shown in Figure 5.17(a).

Figure 5.17(b) shows comparison of traditional timing-estimation schemes, our timing models and the actual clock period. And from results in Figure 5.17(c) we can draw the following observations. Estimators that use only delay of functional units provide estimates with an average of 31.9% error (Figure 5.17(c)). Estimators that use only unit delays in the datapath (i.e., registers, functional units, muxes, etc.) in the clock period estimation provide estimates with an average of 18.2% error. Estimators that obtain clock period estimation by considering datapath and wiring delays give result with an average of 7.5% error. However, using our timing models that consider all constituents of a chip and technology factors giving the results with an average of 2.7% error.

Elliptical filter designs with 19 control steps, 2 adders and a 2-st pipelined mult.	Datapath delay (ns)	Control delay (ns)		Wire/Load delay (ns)		Clock period (ns)	
		Estimate	Actual	Estimate	Actual	Estimate	Actual
A (10reg,36mux-i/p)	152.0	19.4	14.3	20.2	20.6	191.6	186.9
B (11reg, 28mux-i/p)	151.1	18.8	11.4	19.6	20.8	189.5	183.3
C (12reg, 26mux-i/p)	149.3	18.5	14.1	19.9	21.2	187.7	184.6
D (13reg, 23mux-i/p)	152.5	18.5	12.4	19.9	20.2	190.9	185.1

(a)



(b)

Elliptical filter designs with 19 control steps, 2 adders and a 2-st pipelined mult.	Percent error of clock-period estimates using various performance measures			
	Using only function units delay	Using only datapath delay	Using datapath and wiring delays	Using our timing models
A (10reg,36mux-i/p)	32.6%	18.7%	7.9%	2.5%
B (11reg, 28mux-i/p)	31.3%	17.4%	6.9%	3.4%
C (12reg, 26mux-i/p)	31.2%	19.1%	8.3%	1.7%
D (13reg, 23mux-i/p)	31.9%	17.6%	6.9%	3.1%

(c)

Figure 5.17: The clock period for four designs of the elliptical filter benchmark: (a) table of data, (b) comparison of different timing estimation schemes, (c) percentage error of each estimation scheme.

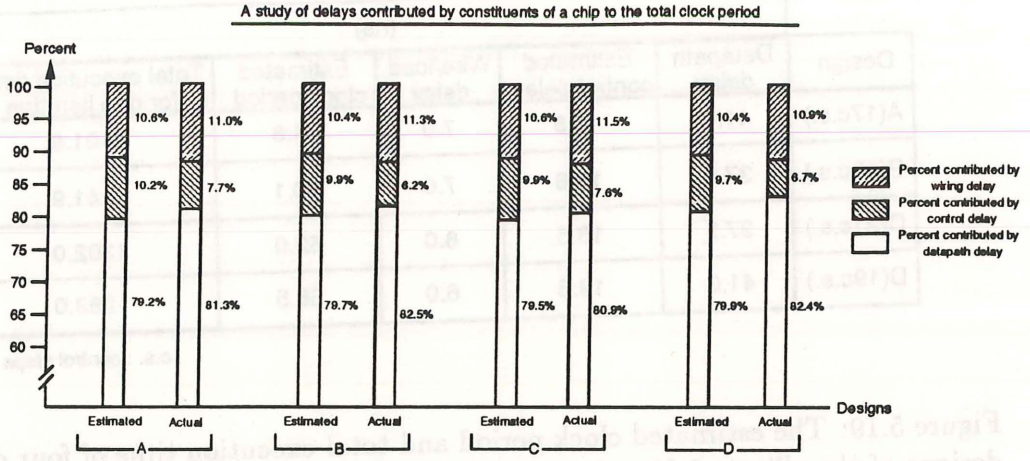


Figure 5.18: Delay distribution of constituents of a chip.

Using data obtained in the first experiment, we derive a distribution bar-chart shown in Figure 5.18. The chart shows that the clock period comprises of delay contributed by each constituent of the chip, as follows:

- an average of 80% of the clock period is contributed by the delay in the datapath units,
- an average of 10% of the clock period is contributed by the wiring and its driving load, and
- an average of 10% of the clock period is contributed by the control-unit delay.

Because the elliptic-filter benchmark is a datapath-dominated design, the main contributor of the the clock period is the datapath delay. However, the amount of contribution by each constituent to the clock may vary from design to design.

	(ns)				
Design	Datapath delay	Estimated control delay	Wire/load delay	Estimated clock period	Total execution time for one iteration
A(17c.s.)	41.0	16.8	7.0	64.8	1101.6
B(19c.s.)	37.5	15.6	7.0	60.1	1141.9
C(21c.s.)	37.5	18.5	6.0	62.0	1302.0
D(19c.s.)	41.0	19.5	6.0	66.5	1263.0

c.s. : control steps

Figure 5.19: The estimated clock period and total execution time of four different designs of the elliptical filter benchmark.

In the third experiment, we have tested our timing models on four synthesized designs of a 16-bit elliptical filter benchmark with four different schedules and design styles (Figure 5.19): (1) design A with 17 control steps, 3 adders, 2 multipliers, 10 registers and 34 mux-inputs, (2) design B with 19 control steps, 2 adders, 2 multipliers, 11 registers and 28 mux-inputs, (3) design C with 21 control steps, 2 adders, 1 multiplier, 10 registers and 25 mux-inputs, and (4) design D with 19 control steps, 2 adders, one 2-stage pipelined multiplier, 10 registers and 28 mux-inputs. The delay computation in this experiment is based on an 1.5μm technology [VTI88].

The results in Figure 5.19 show the total execution time (for one iteration) of four designs. We can use these estimates to guide the selection of designs that will satisfy a given performance constraint. For instance, if the performance constraint is *1000ns* there are two designs, *Design A* and *Design B*, that can satisfy the

constraint. On the other hand, if the performance constraint is *1600ns* all four designs satisfy the constraint.

5.5 Conclusions

In this chapter, we have presented layout models for area and performance measures for behavioral synthesis. Since the datapath area-model has taken into account most of technology factors, including layout architecture, component library, placement and routing, our preliminary results show high fidelity and accuracy for datapath area estimates. Similarly, the datapath timing model also can well predict the datapath delay.

In the control-unit area and timing models, since we ignored the impacts of logic optimization, the simple-minded bounds may not be adequate when more accurate estimates are needed. Better estimates can be obtained by generating more accurate netlists for the control logic and by modeling placement and routing algorithms more accurately.

One approach is to generate a large netlist for the control-state table and perform a limited technology mapping by decomposing each AND or OR gate into series of gates from the given library. Placement can be modeled by probabilistic distributions of pin positions and wire lengths. For example, Kurdahi

and Parker [KuPa89] assume a uniform distribution of pins across R rows of cells and a geometric distribution of the wire length with average wire length obtained experimentally by running many examples. Routing algorithms are approximated by computing routing density across each channel.

To obtain even better estimates, more accurate models of the placement and routing algorithms must be used. Pedram and Preas [PePr89] model a placement algorithm that minimizes the sum over all nets of the half-perimeter length of the rectangle enclosing pins of each net. They also model global routing by approximating a minimal rectangular Steiner tree for connecting pins on each net and channel routing by approximating the left-edge algorithm. Instead of modeling placement and routing algorithms, some linear algorithms can be used to obtain even more accurate estimates. Zimmermann [Zimm88] uses the well-known min-cut algorithm [FiMa82] to quickly generate an acceptable floorplan from netlist of components with known area aspect ratios. In addition, Kurdahi and Ramachandran [KuRa91] combine the analytical and constructive methods to provide fast and accurate area estimates for standard cell layouts.

The main drawback of obtaining high-accuracy estimates from RT schematics is that it will greatly increase the computational complexity. However, long estimation times are not desirable in behavioral synthesis. If the estimate turnaround time is the main concern, then a simple and fast area measure should be used and the fidelity of the estimates are more important than the accuracy.

In essence, "accuracy" and "fidelity" are two main factors in quality measures. In order to improve accuracy, the impacts of control-logic optimization and floorplanning need to be study further. Furthermore, high-fidelity and fast quality measures are needed to support behavioral synthesis, and extensive empirical study is necessary before any model can be objectively established.

Chapter 6

A Unified Model for Behavioral Synthesis

In order to incorporate layout information into behavioral synthesis, a unified model is needed to noticeably reflect the behavior and the structure of the design. Using such a model, synthesis tools can retrieve layout information in any design stage and use this information to guide the design process.

This chapter presents a unified model that is a structural representation by grouping nodes and edges of a control/data flow graph (CDFG). This model encapsulates both the behavior (CDFG) and the structure of the design (Figure 6.1); that is, the alternatives of the design are represented using different graph configurations (e.g., different structural designs) that still encapsulate the same behavior. Two graph formations for two different target architectures, a point-to-point (random topology) datapath with a one-phase clock, and a multi-bus (linear topology) datapath with a two-phase clock, are presented.

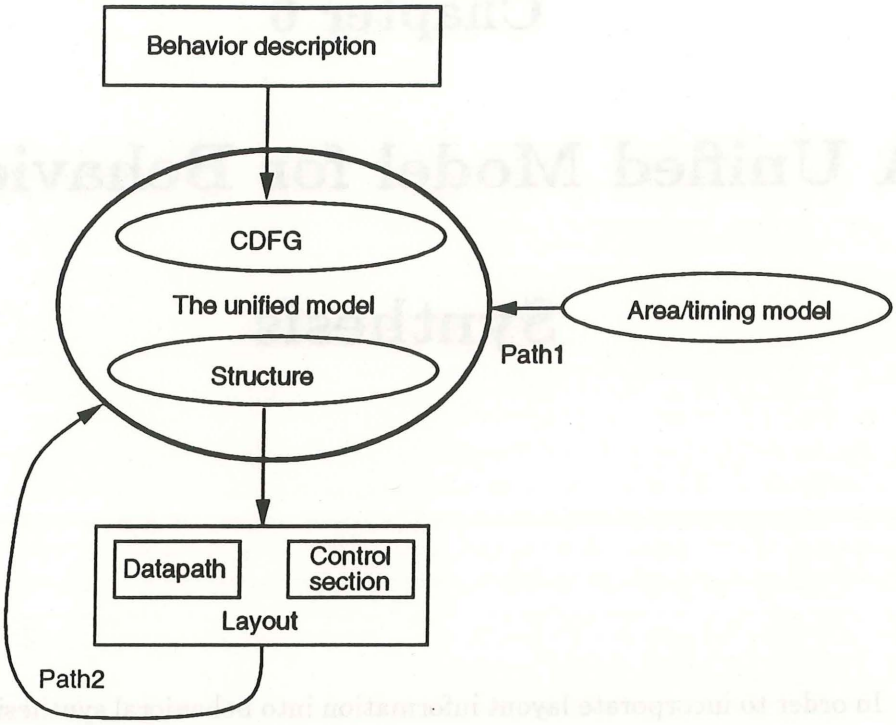


Figure 6.1: A unified model for behavioral synthesis.

Using the proposed unified model and the layout model discussed in Chapter 5, the synthesis tool can evaluate physical information during the behavioral synthesis process (*Path1*). A layout-driven unit-binding approach using the proposed unified model and the layout model is discussed in Chapter 7. Furthermore, using this model synthesis tools can feed back the physical information to guide the design process (*Path2*). A feedback-driven approach for clock-cycle estimation is also discussed in Chapter 7.

The remainder of this chapter is organized in the following manner. Section 6.1 describes the control/data flow graph (CDFG) representation. Section 6.2

presents the structural graph model. Sections 6.3 and 6.4 describe the relationship between the graph and the structure including datapath formation, control-unit formation and chip formation for two different datapath architectures (random and linear topologies) and clocking schema (one-phase and two-phase). Section 6.5 presents a unified view from system to module using the proposed unified model. Finally, Section 6.6 summarizes the proposed unified model.

6.1 CDFG: Control/Data Flow Graph

A behavioral description is usually converted into a hierarchical CDFG $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. $\mathcal{V} = \{V_i \mid i = 1..n\}$ denotes a set of behavioral supernodes. $type(V_i) \in \{op, if, case, for, while, join\}$ denotes the type of the supernode, an operational or a control supernode. Let G=(V,E) denote a data-flow graph (DFG), where $V = \{v_j \mid j = 1..m\}$ represents a set of operational nodes and $E = \{e_{jm} \mid \{v_j, v_m\} \in V\}$ represents a set of data-dependency edges. $type(v_j) \in \{mult, add, sub, shift, comp, ..\}$ denotes the type of the operational nodes.

Each control-supernode such as $if, case, for, while, join$, consists of a DFG that determines the conditional branch decisions, while each operational-supernode consists of a DFG performing data computations. Figure 6.2(b) shows a CDFG that corresponds to the VHDL program in Figure 6.2(a). This CDFG consists of 10

control-supernodes in which $type(V_1)$={for}, $type(V_3)$={if}, $type(V_4)$={$case$}, $type(V_5)$={$while$}, $type(V_9)$={$join$} and $\{V_i \mid i = 2, 6, 7, 8, 9\}$={$op$}.

$\mathcal{E}$ ={e_{ik} | $\{V_i, V_k\} \in \mathcal{V}$} $\cup$ {e_{jm} | $\{v_j, v_m\} \in V$} denotes a set of edges that contain three types of edges, control edge, data edge and timing edge; that is, $type(e_{ik}) = \{control, data, timing\}$. The control-edge indicates the mutually exclusive branch conditions of control supernodes. The data-edge denotes the data dependency between nodes. Each edge e_{jm} corresponds to a variable $var(e_{jm})$. Finally, the timing-edge indicates the timing constraint between two supernodes or two operational nodes. For example, e_1 in Figure 6.2 represents the timing constraint between supernodes V_4 and V_{10}. Similarly, e_2 represents the timing constraint between two operational nodes in V_6.

6.2 The Supergraph Model

Let $\mathbf{H} = (\mathbf{V}, \mathbf{E})$ denote a supergraph. $\mathbf{V}$={$V_i \mid i = 1..n$} denotes a set of structural supernodes. Each structural supernode represents a particular structural component such as a functional unit, a storage unit, an I/O port or a control unit. There are four types of structural supernodes, I/O port, storage, functional-unit and control, that are specified by $type(V_i)$={I/O,MEM,FU,CNT}. Let $f(V_i)$ denote the function of the supernode V_i. For instance, if V_i is a storage supernode then V_i can be a latch, a flip-flop, a register, a register file, a RAM or a

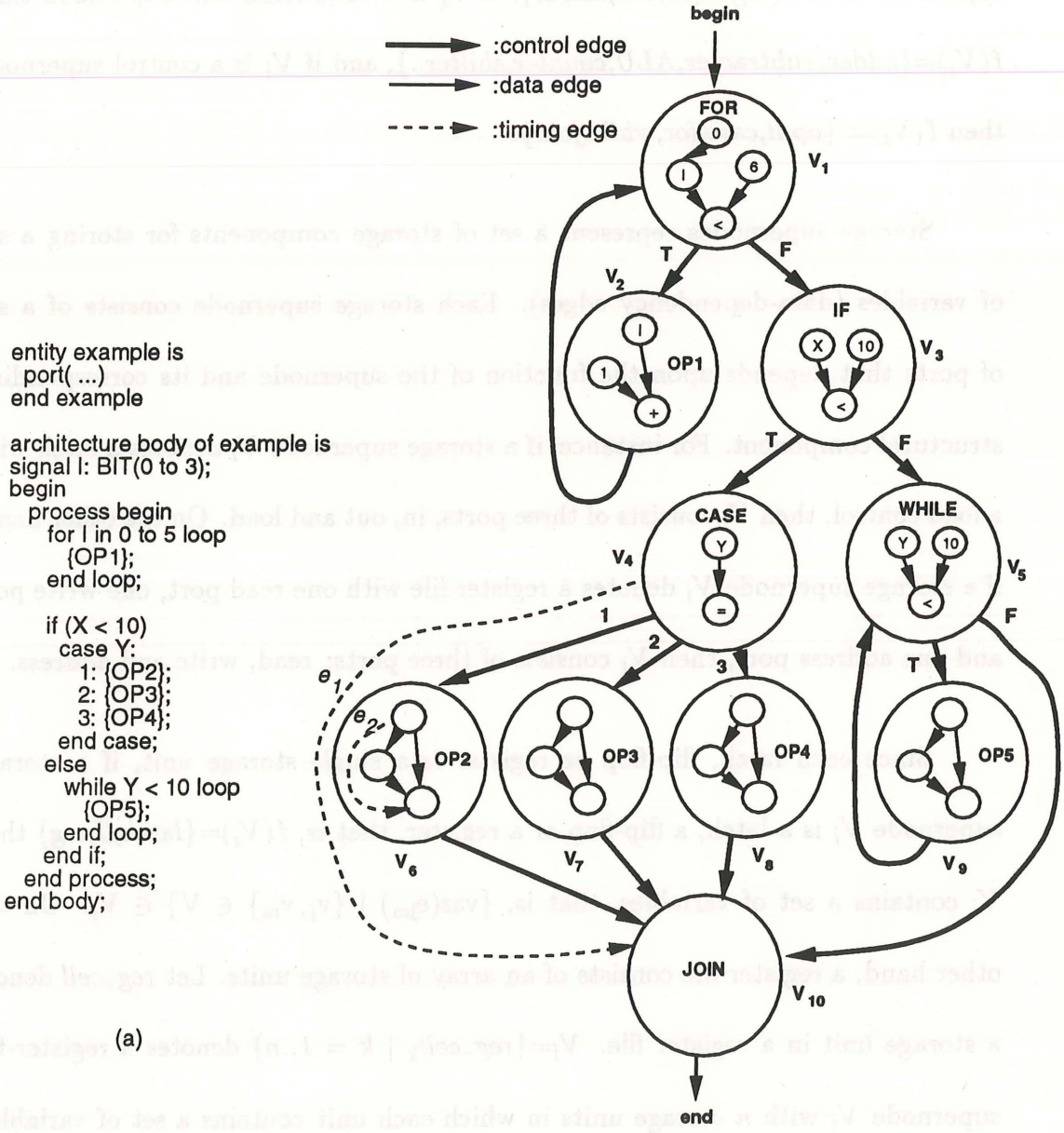


Figure 6.2: A hierarchical control/data flow graph representation: (a) a VHDL program, (b) the corresponding CDFG.

ROM, that is, $f(V_i)$={*latch, ff,reg,reg_file,RAM,ROM*}. Similarly, if V_i is an I/O supernode then $f(V_i)$={*in,out,in_out*}, if V_i is a functional-unit supernode then $f(V_i)$={*adder/subtracter,ALU,counter,shifter..*}, and if V_i is a control supernode then $f(V_i)$= {*op,if,case,for,while,join*}.

Storage supernodes represent a set of storage components for storing a set of variables (data-dependency edges). Each storage supernode consists of a set of ports that depends upon the function of the supernode and its corresponding structural component. For instance, if a storage supernode V_i denotes a latch with a load control, then V_i consists of three ports, in, out and load. On the other hand, if a storage supernode V_i denotes a register file with one read port, one write port and one address port, then V_i consists of three ports: read, write and address.

Since each latch, flip-flop or register is a single storage unit, if a storage supernode V_i is a latch, a flip-flop or a register, that is, $f(V_i)$={*latch,ff,reg*} then V_i contains a set of variables, that is, $\{var(e_{jm}) \mid \{v_j, v_m\} \in V\} \in V_i$. On the other hand, a register file consists of an array of storage units. Let *reg_cell* denote a storage unit in a register file. V_i={$reg\_cell_k \mid k = 1..n$} denotes a register-file supernode V_i with n storage units in which each unit contains a set of variables. Further, a RAM or a ROM consists of a two-dimensional array of storage units. Let *mem_cell* denote a storage unit in a RAM or a ROM. V_i={$mem\_cell_{jk} \mid j = 1..n, k = 1..m$} denotes a RAM or a ROM V_i with $n \times m$ storage units in which each unit contains a set of variables.

Each functional-unit supernode represents a particular functional unit containing a set of operational nodes that can be executed by this functional unit. Each functional-unit supernode also consists of a set of ports that depends upon the function of the supernode and its corresponding structural component. For instance, if a functional-unit supernode V_i denotes an ALU with eight functions, then V_i consists of four ports, in1, in2, out and select.

As discussed in Section 6.1, each supernode in the CDFG contains a DFG. Control supernodes correspond to the behavioral supernodes of the CDFG. Each control supernode contains the schedule of the DFG in a behavioral supernode. Let t_j denote a time-step j. V_i=$\{t_j \mid j = 1..n\}$ denotes a control supernode V_i with a n-time-step schedule. Each time-step node consists of a set of operational nodes that are executed in this time step, that is, t_j=$\{v_i \in V\}$.

E=$\{e_{ij} \mid \{V_i, V_j\} \in V\}$ denotes a set of superedges. There are two types superedges, control and data, that are specified by $e_{ij} = \{cnt, data\}$. In addition, $w(e_{ij})$ represents the weight of e_{ij}. Each data superedge represents the physical connection between two supernodes, while control superedges represent the control flows and branch conditions. The weight of a data superedge is the number of variables communicating between the two supernodes. In addition, the superedge direction depends on the direction of control/data flow between the supernodes. For example, a superedge e_{12} is connected from V_1 to V_2 so that e_{12} is an outgoing superedge of V_1 while e_{12} is an incoming superedge of V_2. Since certain supernode

(functional unit) inputs are non-commutable, each superedge uses a flag to indicate the input position of the connecting supernode.

6.2.1 Supergraph Formation

We are given a CDFG, a set of functional units and storage units, the schedule and the variable/operation assignments (the operation/variable binding algorithm is discussed in the next chapter). The supergraph-formation algorithm folds the CDFG into a supergraph in two steps: supernode formation and superedge formation. Using the FSMD model described in Chapter 3, we divide the graph into two parts, datapath and control unit.

For the datapath, the supergraph-formation algorithm maps the input/output ports, functional and storage units to a set of structural supernodes. Then, the algorithm maps variables and operators to their corresponding supernodes. For example, Figures 6.3(a) and (b) show a VHDL program and its corresponding CDFG. Using an adder, a multiplier and a comparator, a scheduler partitions this CDFG into 6 time steps. In addition, the allocator assigns three registers for storing variables as: variables $\{a, d, h\}$, $\{e, b, g\}$ and $\{c, f, k\}$ are stored in registers R_1, R_2 and R_3, respectively.

In the first step, the supergraph-formation algorithm first maps the comparator, multiplier and adder to supernodes V_7, V_8, and V_9, registers R_1, R_2

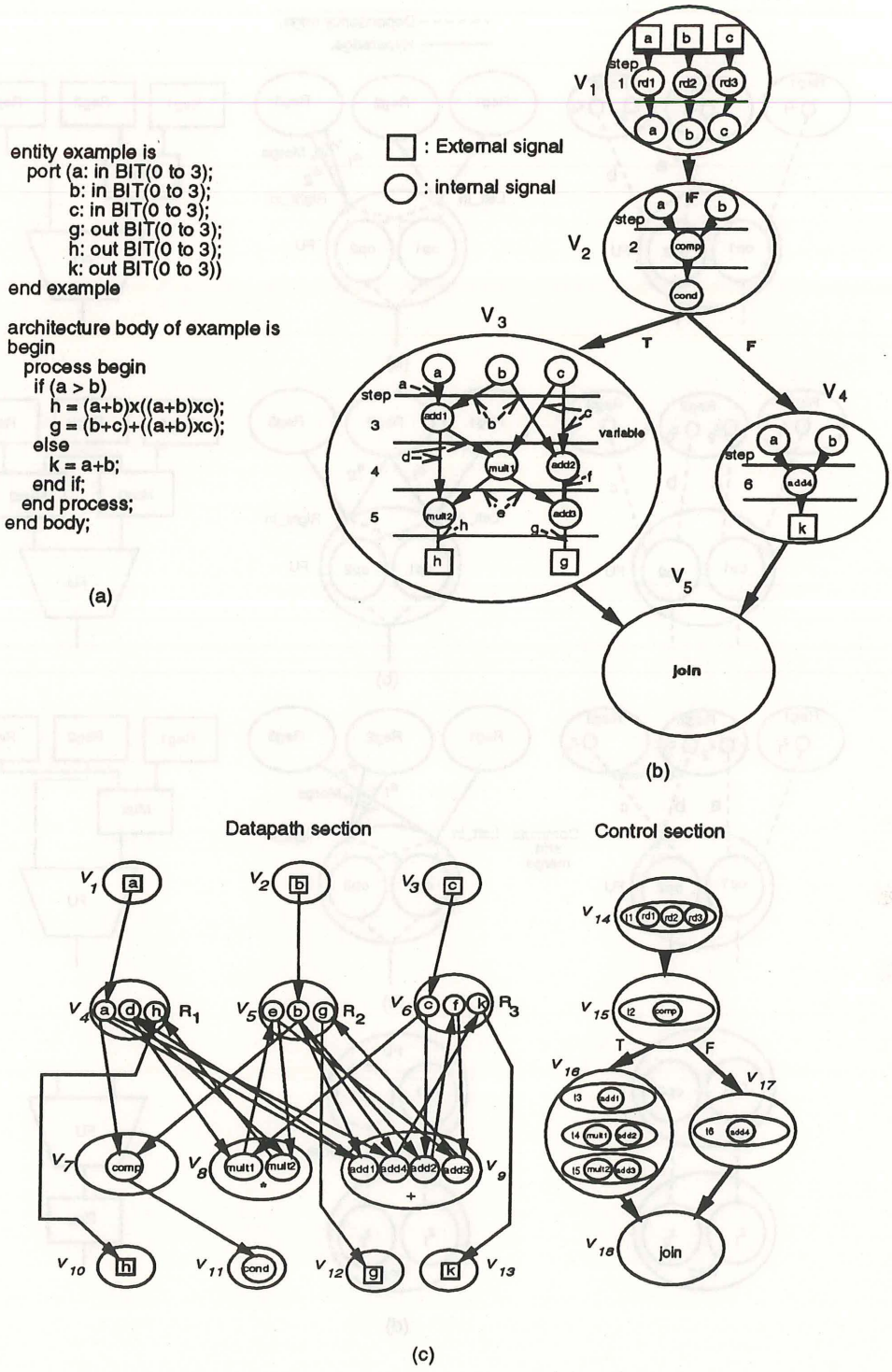


Figure 6.3: The graph formation: (a) a VHDL program, (b) the CDFG, (c) the supernode formation.

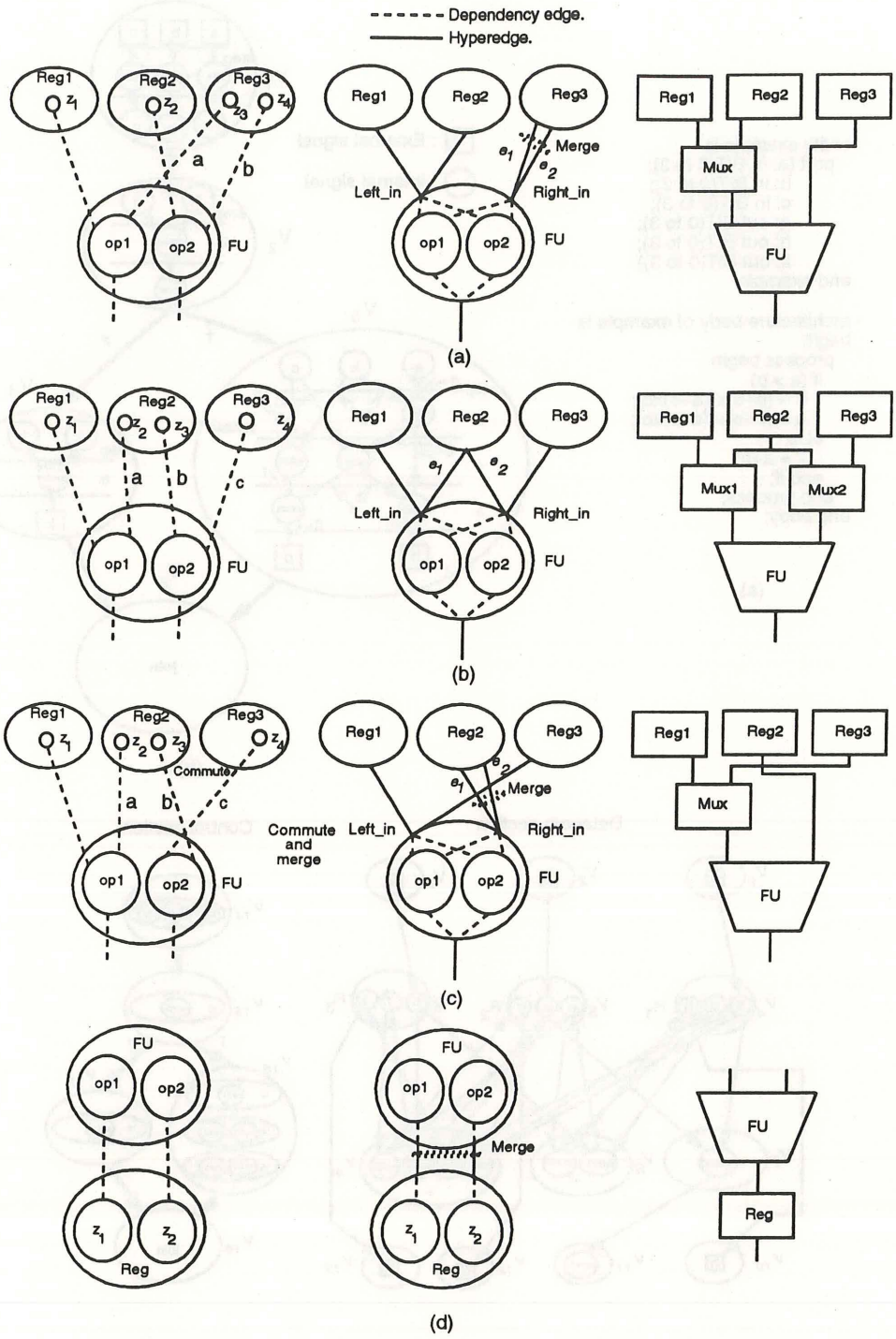


Figure 6.4: Superedge merging

and R_3 to supernodes V_4, V_5 and V_6, and input/output ports a, b, c, h, g, k to supernodes $V_1, V_2, V_3, V_{10}, V_{12}$ and V_{13}, respectively. In the second step, the supergraph-formation algorithm folds variables and operations into their corresponding supernodes. Figure 6.3(c) shows the supergraph generated from the CDFG shown in Figure 6.3(b). The set of operational nodes $\{comp\}$ is assigned to V_7, $\{mult1, mult2\}$ to V_8, and $\{add1, add2, add3, add4\}$ to V_9. In addition, the variables are assigned to supernodes V_4, V_5 and V_6, such that V_4=$\{a, d, h\}$, V_5=$\{b, e, g\}$ and V_6=$\{c, f, k\}$.

In the superedge-formation step, the algorithm maps dependency edges to superedges one at a time. If a dependency edge can share the same path with another superedge, then the edges can be merged into a single superedge; otherwise a new superedge has to be created. Consider Figure 6.4(a). If edges a and b are connected from variables z_3 and z_4 in $Reg3$ to the right input of $op1$ and $op2$, then edge a is mapped to the superedge e_1 and edge b is mapped to the superedge e_2. Since e_1 and e_2 are connected to the right input of FU (i.e., sharing the same signal path), they can be merged. Figure 6.4(b) shows that edge a connects to the right input of $op1$ and edge b connects to the left input of $op2$. Thus, edge a is mapped to the superedge e_2 connected to the right input of the supernode (FU), while edge b is mapped to the superedge e_1 connected to the left input of the supernode (FU). If the inputs of this supernode are not commutable, then these two superedges cannot be merged. Otherwise, these two superedges, e_1 and

e_2, (Figure 6.4(c)) can be merged by commuting edges b and c. Figure 6.4(d) shows that a register supernode has only one input. This means that incoming superedges from the same functional-unit supernode can be merged. Applying the superedge formation on the example of Figure 6.3(c) results in the final supergraph shown in Figure 6.5(a).

For the control unit, the supernodes in the CDFG correspond one-to-one with the control supernodes. Each control supernode consists of a set of time steps, and each time step consists of a set of operations that can be executed in the same time step. For example, the operational node *comp* in V_2 is assigned to the time step t_2. Thus, the control supernode V_{15} consists of a time step t_2 which contains the operational node *comp*. In addition, the control superedges correspond to the control edges in the CDFG.

Algorithm 6.1 describes the supergraph formation. The input to the algorithm includes a CDFG, a set of resources, a schedule and the operation/variable binding result (the binding procedure is discussed in the Chapter 7). The procedure *supernode_mapping()* maps the given resources to a set of supernodes. The procedure *var_op_mapping()* assigns the operations and variables to their corresponding supernodes. The function *superedge_merge_check(e)* returns "true" if the given edge e can be merged with the existing superedge, otherwise returns "false". The procedure *insert_new_superedge(e)* creates a new superedge for edge e. The procedure *control_step_assignment()* assigns operations to the time steps of the

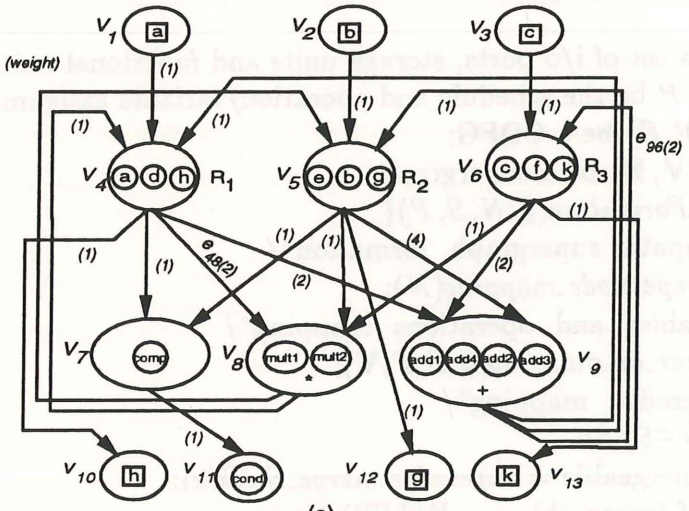


Figure 6.5: Supergraph formation (cont.): (a) the superedge formation, (b) the structural netlist.

control supernodes according to the given schedule. The algorithm first forms the datapath section of the supergraph. Then, the algorithm forms the control section of the supergraph.

Algorithm 6.1. Supergraph Formation.
Let
N be a set of i/o ports, storage units and functional units;
S and P be the schedule and operation/variable assignments;
$\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$ be a CDFG;
$\mathbf{H} = \{\mathbf{V}, \mathbf{E}\}$ be a supergraph;

```
Supergraph_Formation(G,N, S, P){
    /*datapath supergraph formation*/
    V = supernode_mapping(N);
    /*variables and operations mapping*/
    H = var_op_mapping(G,S, P,V);
    /*superedge mapping*/
    for (∀e∈E) do
        mergeable = superedge_merge_check(e);
        if (mergeable == FALSE) then
            insert_new_superedge(e);
        endif
    endfor
    /*control-unit supergraph formation*/
    for (∀V∈V) do
        V' = supernode_mapping(V);
        V = V ∪ V';
    endfor
    /*control-step assignment*/
    H = control_step_assignment(G,H,S);
    return(H);
}
```

Complexity analysis. The complexity analysis of the supergraph formation is described as follows:

1. Supernode mapping takes $O(\mathbf{V})$ time.

2. Variable and operation mapping takes $O(V+\mathcal{E})$ time.

3. Superedge mapping takes $O(\mathcal{E})$ time.

4. Control-step assignment takes $O(V)$ time.

6.3 Supergraph and Structure: I

This section describes the relationship between the supergraph and the structure of the design based on a point-to-point datapath with a one-phase clock target architecture. First, Section 6.3.1 describes the target architecture. Then, Sections 6.3.2, 6.3.3, and 6.3.4 present the datapath formation, control-unit formation and chip formation from the supergraph.

6.3.1 Point-to-Point Datapath with A One-Phase Clock Scheme

The target architecture is defined as follows:

1. Datapath. The datapath uses a point-to-point architecture, as shown in Figure 6.6(a). Both storage and functional units are connected in a point-to-point topology based on the directions of data flows.

2. Control path. The controller consists of three parts: the *state register*, the *control logic* and the *next-state logic*, as shown in Figure 6.6(a). The state

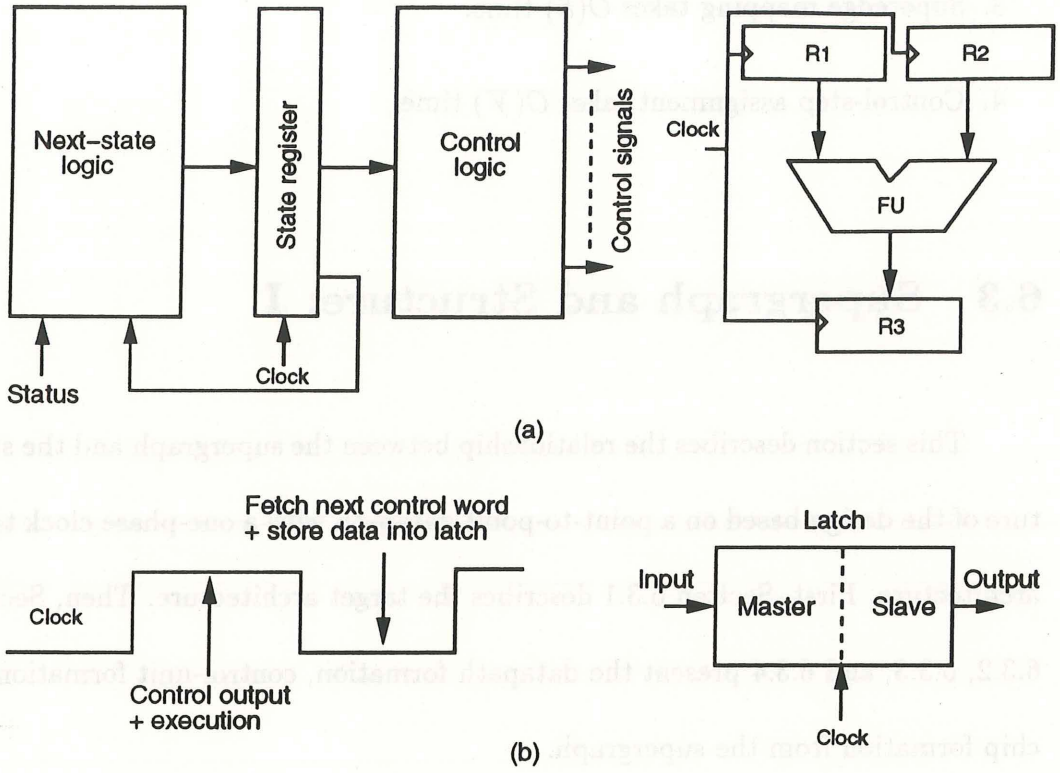


Figure 6.6: The point-to-point datapath with one-phase clock architecture: (a) control/data paths, (b) one-phase clocking scheme.

register stores the currently executing control-word. The control logic emits the control signals to direct the execution of the datapath. The next-state logic determines the next control-word.

3. Clock scheme. An one-phase clock is used and all storage units are implemented using level-sensitive master-slave (M/S) latchs. When the clock is "low" the M/S-latch loads the input signal into the master-latch. When the clock goes "high" the slave-latch fetches the data stored in the master-latch. Using the one-phase clock, the register transfer (RT) operation consists of two steps: (1) when the clock is "high" the control logic emits the control

signals and the datapath executes the RT operation, and (2) when the clock is "low" the state register latches the next control word and the resulting data is stored into the destination latch, as shown in Figure 6.6(b).

6.3.2 Datapath Formation

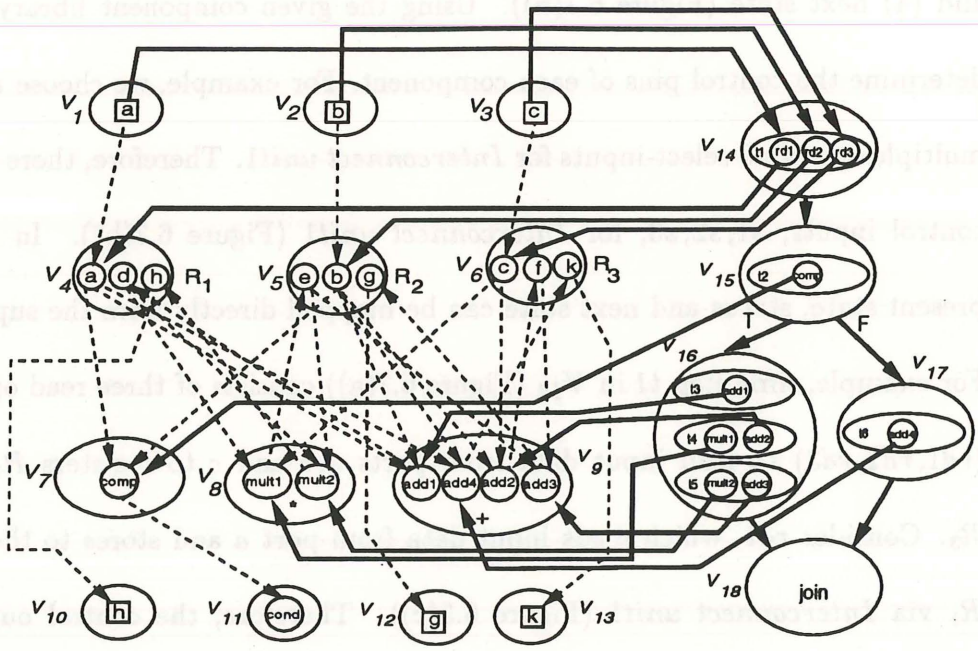
In the datapath section of the supergraph, each supernode denotes a functional unit, a storage component or an input/output port. Each superedge denotes a physical connection between two supernodes. We assume that a single-level interconnect model is used. If a supernode has more than one incoming superedge entering one of its inputs, then a selector (e.g., a multiplexer) is needed to select the data input from different sources. For example, Figure 6.5(b) shows the structural netlist of the supergraph in Figure 6.5(a). The register supernodes V_4, V_5 and V_6 are mapped to $Reg1$, $Reg2$ and $Reg3$, respectively. The functional-unit supernodes V_7, V_8 and V_9 are mapped to a comparator, a multiplier and an adder.

In addition, input supernodes V_1, V_2 and V_3 are mapped to input ports a, b and c, while output supernodes V_{10}, V_{11}, V_{12} and V_{13} are mapped to output ports h, $cond$, g and k. Since all the register supernodes have more than one incoming superedge (3 for V_4 and V_5, and 2 for V_6), each register needs an interconnect unit to select data inputs from different sources ($Interconnect\ unit1, 2, 3$ are connected to $Reg1$, $Reg2$ and $Reg3$). For the functional-unit supernode V_8,

the incoming superedge e_{48} is shared by all of the operational nodes (*mult*1 and *mult*2) in V_8 (i.e., the left input of the multiplier has only one data input source so that an interconnect unit is not needed). On the other hand, an interconnect unit (*Interconnect unit*4) is required for the right input of the multiplier to select inputs from two sources. Similarly, the adder needs an interconnect unit (*Interconnect unit*5) for its left input. Since the interconnect units are represented implicitly in the supergraph, a supergraph can be viewed as a structural representation.

6.3.3 Control-Unit Formation

The control section of the supergraph denotes the control sequence of the design. If a control supernode has more than one outgoing control superedge, then this supernode is a conditional branch node. For example, the control supernode V_{15} in Figure 6.7(a) is an *if* supernode which has two outgoing superedges. The branch decision depends on the condition status as: (1) branch to V_{16} when *cond*=true, and (2) branch to V_{17} when *cond*=false. Each control supernode consists of a set of time steps, in which each time step consists of a set of operational nodes which are executed in this time step. These nodes are linked to the operational nodes resided in the datapath section of the supergraph as shown in Figure 6.7(a).



(a)

Present state	Control ouput															Status	Next step
	Register			Interconnect													
Step	R1		R2		R3		Unit1			Unit2			Unit3		Unit4		Unit5			
	load		load		load		s1	s2	s3	s1	s2	s3	s1	s2	s1	s2	s1	s2	cond.	
t1	1		1		1		0	1	0	0	1	0	1	0	0	0	0	0	0	t2
t2	0		0		0		0	0	0	0	0	0	0	0	0	0	0	0	0/1	t6/t3
t3	1		0		0		1	0	0	0	0	0	0	0	0	0	1	0	0	t4
t4	0		1		1		0	0	0	1	0	0	0	1	0	1	0	1	0	t5
t5	1		1		0		1	0	0	0	0	1	0	0	1	0	0	1	0	stop
t6	0		0		1		0	0	0	0	0	0	0	1	0	0	1	0	0	stop

(b)

Figure 6.7: Control-unit formation: (a) the control-section of the supergraph, (b) the control-state table.

We formulate the control section of the supergraph to a control-state table, which includes four parts: (1) present state, (2) control output, (3) status input and (4) next state (Figure 6.7(b)). Using the given component library, we can determine the control pins of each component. For example, we choose a 3-input multiplexer with 3 select-inputs for *Interconnect unit*1. Therefore, there are three control inputs, $s1, s2, s3$, for *Interconnect unit*1 (Figure 6.7(b)). In addition, present state, status and next state can be mapped directly from the supergraph. For example, time step *t1* in V_{14} (Figure 6.7(a)) consists of three read operations $(rd1, rd2, rd3)$ to load input data from ports a, b and c to registers R_1, R_2 and R_3. Consider $rd1$, which reads input data from port a and stores to the register R_1 via *Interconnect unit*1 (Figure 6.5(c)). Therefore, the control outputs for the load input of R_1 and the select input $s1$ of *Interconnect unit*1 are set to one. Similarly, the load inputs of ports b and c and the select inputs $s2$ and $s1$ of *Interconnect unit*2 and *Interconnect unit*3 are set to one for $rd2$ and $rd3$ operations. Since this control supernode (V_{14}) is not a branch node, the next state is the first time step (*t2*) of V_{15} (V_{14}'s successor node). On the other hand, for a conditional branch node V_{15}, the next state depends on the conditional status (row *t2* of Figure 6.7(b)).

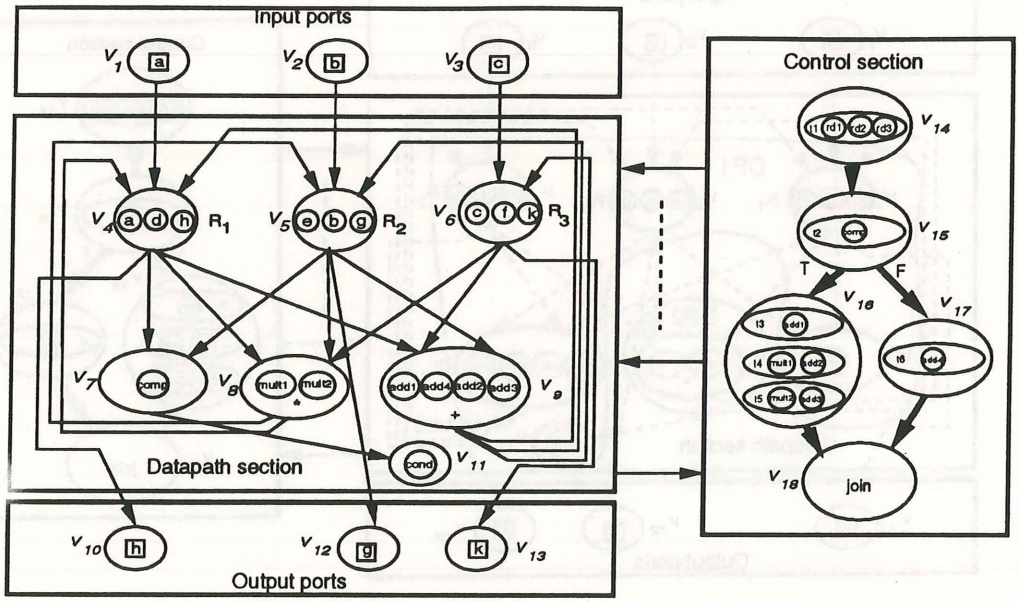


Figure 6.8: Chip formation: (a) the supergraph, (b) the chip structure.

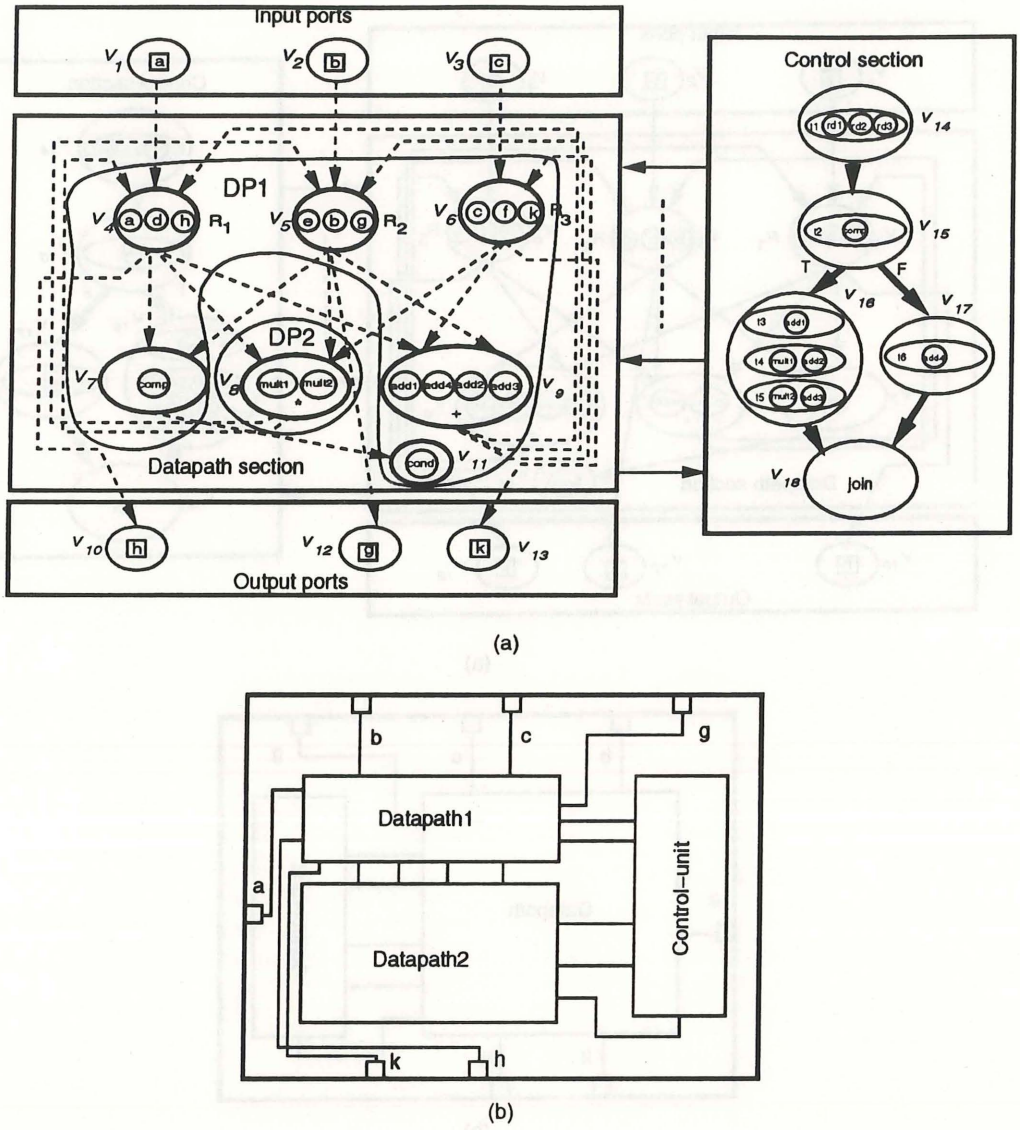


Figure 6.9: Chip formation with multiple datapaths: (a) the supergraph, (b) the chip structure.

6.3.4 Chip Formation

Using the datapath and control-unit formation techniques described in the previous two sections, we can directly map the supergraph into the FSMD chip architecture. Using the Figure 6.3 example, the final supergraph in Figure 6.8(a) is divided into four parts: *Datapath section*, *Control section*, *Input ports* and *Output ports*. Figure 6.8(b) shows the chip formation of the supergraph in Figure 6.8(a). Each section of the supergraph is mapped to a particular section of the chip. The *Datapath section* is mapped to a *Datapath* that can be implemented using a bit-sliced stack or standard cells, while the *Control section* is mapped to a *Control unit* that can be implemented using a PLA or standard cells. Each port supernode is mapped to a chip-pin that consists of an I/O pad and a pad driver. In addition, the superedges across the boundaries of the datapath, the control unit and the input/output ports are mapped to the routing area of the chip.

Similarly, the chip formation can be applied to the FSMD model with multiple datapaths. Figure 6.9(a) shows the supergraph in which the datapath section is divided into two subsections, *DP1* and *DP2*. In the chip formation, each datapath subsection of the supergraph is mapped to a datapath on the chip, e.g., *DP1* is mapped to *Datapath1* while *DP2* is mapped to *Datapath2*, as shown in Figure 6.9(b).

6.4 Supergraph and Structure: II

This section describes the relationship between the supergraph and the structure of the design based on a multi-bus datapath with a two-phase clock target architecture. First, Section 6.4.1 describes the target architecture. Then, Section 6.4.2 presents the datapath, control-unit and chip formations from the supergraph.

6.4.1 Multi-bus Datapath with A Two-Phase Clock Scheme

The target architecture is defined as follows:

1. Datapath. The datapath uses a multi-bus architecture, as shown in Figure 6.10(a). Both storage and functional units are connected to the buses. Tri-state buffers are inserted between units and buses. Storage units, such as registers, latches and flip-flops, are grouped into multi-port register files.

2. Control path. The controller consists of three parts: *the state register*, *the control logic* and *the next-state logic*, as shown in Figure 6.10(a). The state register stores the currently executing control-word. The control logic emits the control signals to direct the execution of the datapath. The next-state logic determines the next control-word.

3. Clock scheme. A two-phase nonoverlapping clock is used and all storage units are implemented using level-sensitive latches. Typically, the RT operation

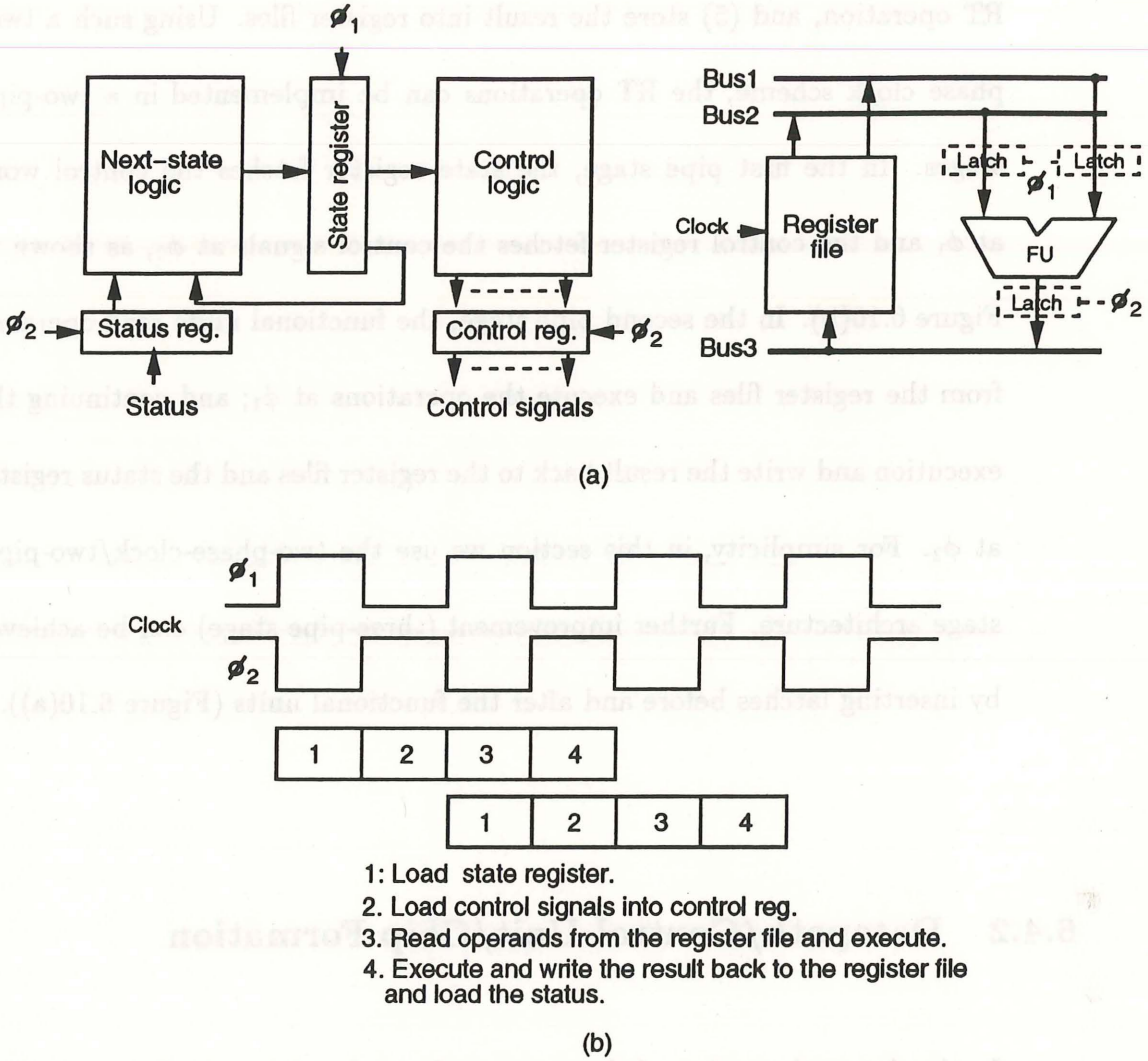


Figure 6.10: The multi-bus datapath with a two-phase clock architecture: (a) control/data paths with two-pipe stage and three-pipe stage with latch insertion (dash boxes), (b)two-phase-clock/two-pipe-stage scheme.

is divided into five steps: (1) load the control word into state register, (2) decode the control signals, (3) read data from register files, (4) execute the RT operation, and (5) store the result into register files. Using such a two-phase clock scheme, the RT operations can be implemented in a two-pipe stages. In the first pipe stage, the state register fetches the control word at ϕ_1 and the control register fetches the control signals at ϕ_2, as shown in Figure 6.10(b). In the second pipe stage, the functional units read operands from the register files and execute the operations at ϕ_1; and continuing the execution and write the result back to the register files and the status register at ϕ_2. For simplicity, in this section we use the two-phase-clock/two-pipe-stage architecture. Further improvement (three-pipe stage) can be achieved by inserting latches before and after the functional units (Figure 6.10(a)).

6.4.2 Datapath/Control-Unit/Chip Formation

In the datapath section of the supergraph, each supernode denotes a functional unit, a storage component or an input/output port. Each superedge represents a physical connection between a port of the supernode and a bus. All the superedges that share a common port can share the same bus. In the following of this section, I will use the Figure 6.3 example to describe the datapath/control-unit/chip formation from the supergraph.

Cycle	Phase	Operation1	Operation2	Operation3
1	1	Load state register		
	2	Load control signals		
2	1	Read external signals a, b and c	Load state register	
	2	Store signals a, b and c into RF	Load control signals	
3	1	Read variables a and b from RF	Comp(a,b) => "cond"	
	2	Store "cond" into status register		
4	1	"cond" = true Load state register	"cond" = false Load state register	
	2	Load control signals	Load control signals	
5	1	Read variables a and b from RF Add(a,b) => d	Read variables a and b from RF Add(a,b) => k	Load state register
	2	Store d into RF	Store k into RF	Load control signals
6	1	Read variables b, c and d from RF Mult(c,d) => e, Add(b,c)=>f		Load state register
	2	Store e and f into RF		Load control signals
7	1	Read variables d, e and f from RF Mult(d,e) => h, Add(e,f) => g		
	2	Store g and h into RF		

Figure 6.11: The schedule of the CDFG example in Figure 6.3(b).

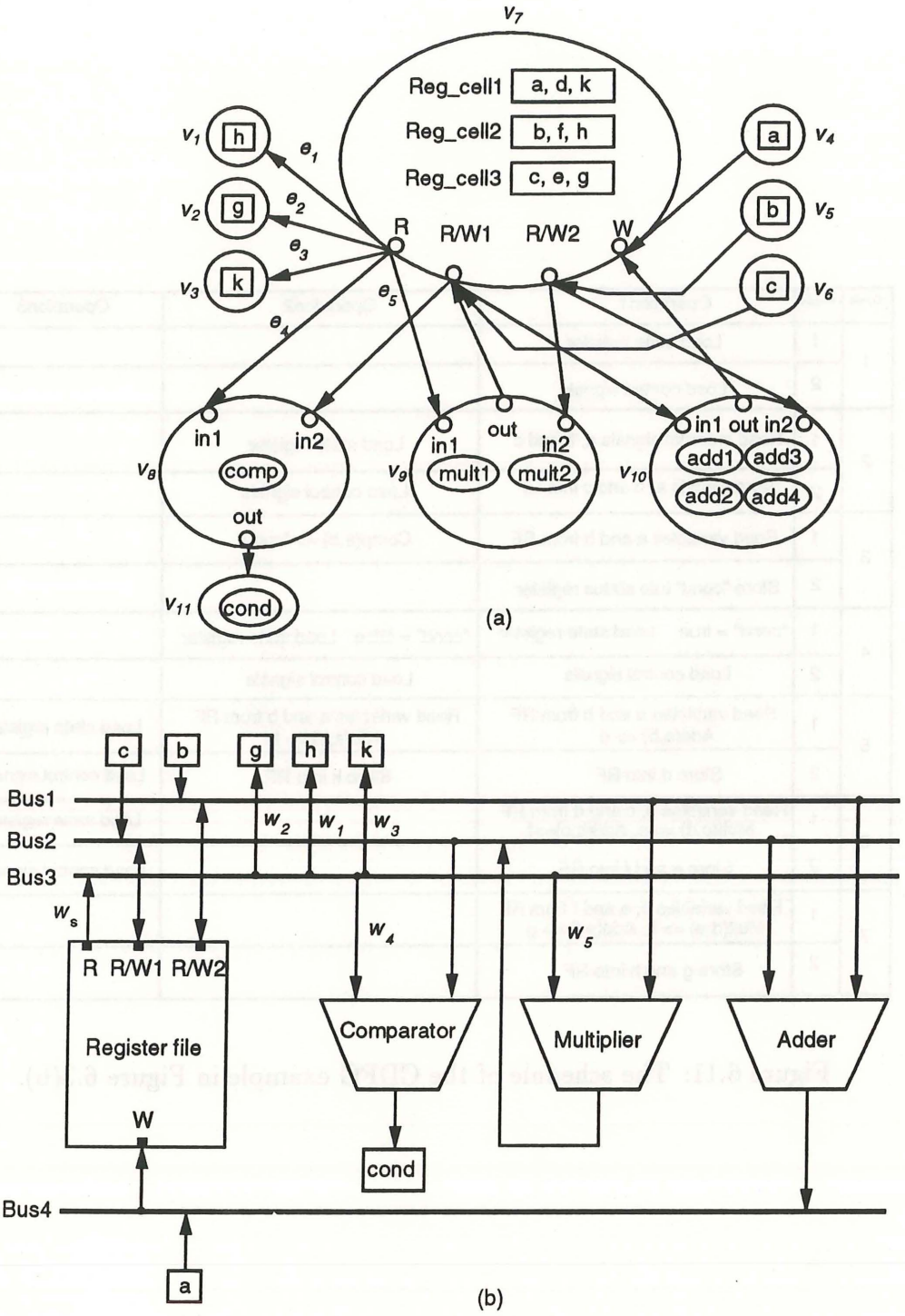


Figure 6.12: Datapath formation: (a) the supergraph, (b) the structural netlist.

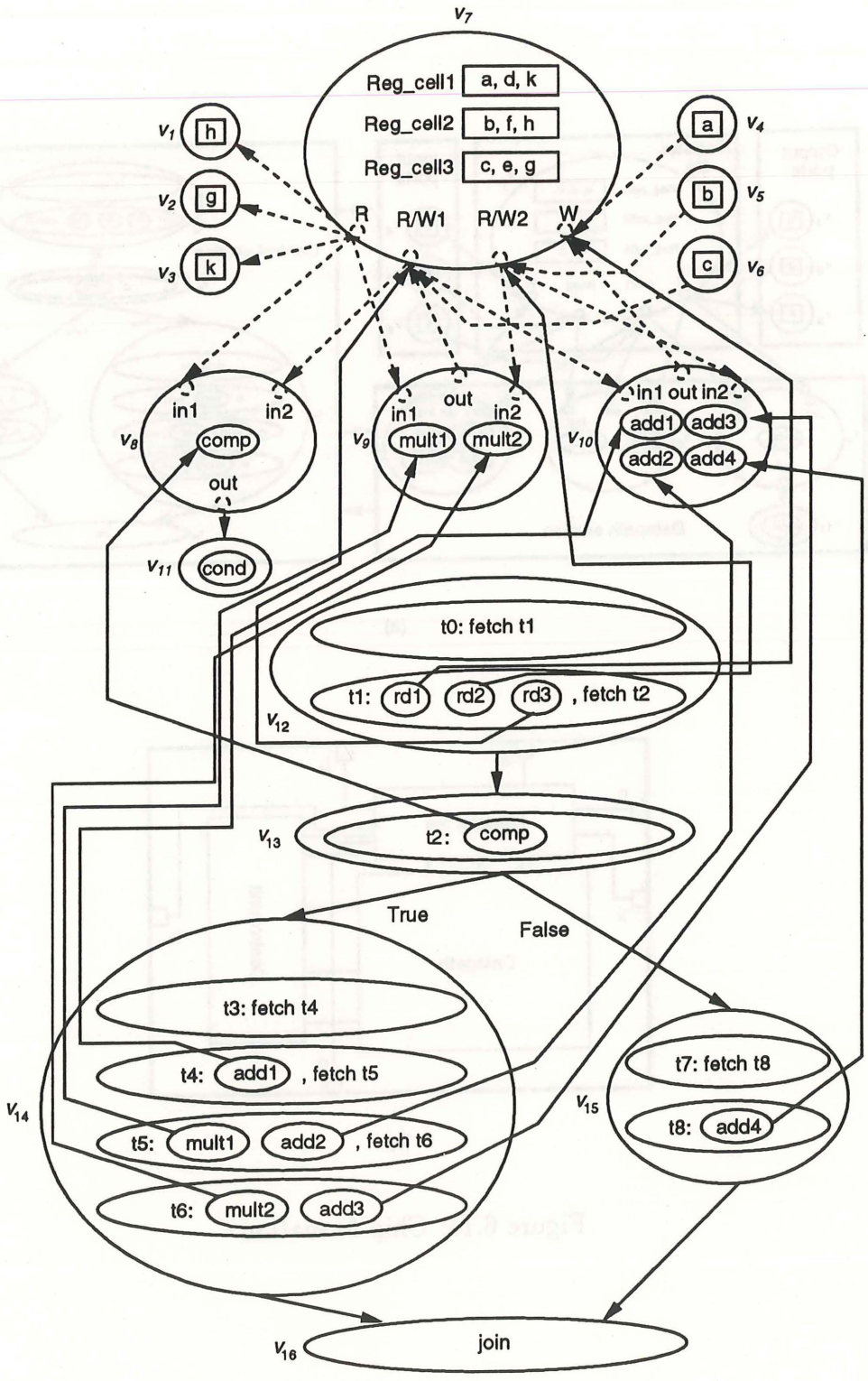
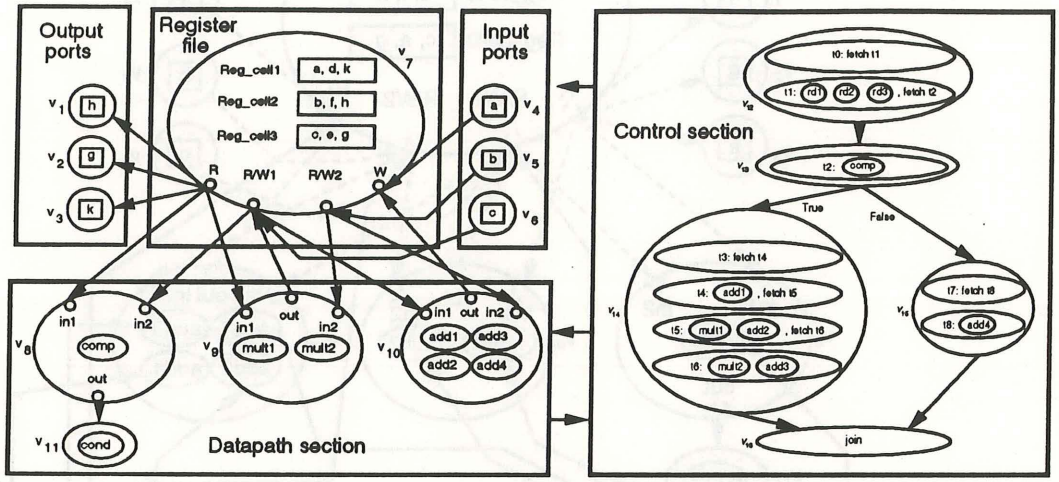
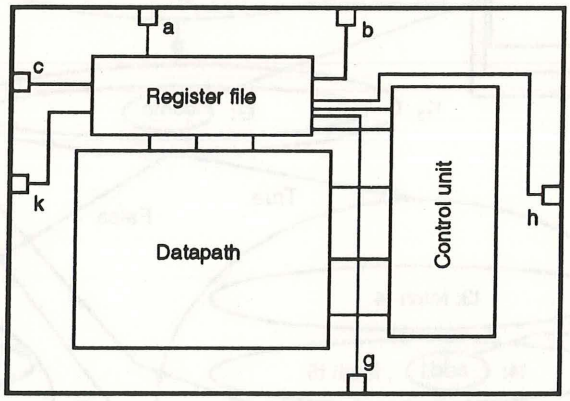


Figure 6.13: The final supergraph.



(a)



(b)

Figure 6.14: Chip formation.

Using the proposed target architecture and clocking scheme, the schedule of the CDFG example in Figure 6.3(b) is shown in Figure 6.11. The unit/storage binding result is the same as described in section 6.3 (Figure 6.5(a)) except the registers are grouped into a register file. This register file consists of four ports, one read-only, one write only and two read/write ports, and contains three register cells, as shown in Figure 6.12(a). Figure 6.12(b) shows the structural netlist of the supergraph in Figure 6.12(a). The supernode V_7 is mapped into a register file that consists of three register cells, *Reg_cell*1, *Reg_cell*2 and *Reg_cell*3 and four ports, R, R/W_1, R/W_2 and W. The functional-unit supernodes V_8, V_9 and V_{10} are mapped to a comparator, a multiplier and an adder. In addition, input supernodes V_4, V_5 and V_6 are mapped to input ports a, b and c, while output supernodes V_1, V_2, V_3 and V_{11} are mapped to output ports h, g, k and *cond*. Using the multi-bus architecture, all the superedges that share the same connection are grouped into a bus. For instance, superedges e_1, e_2, e_3, e_4 and e_5 are mapped to wires w_1, w_2, w_3, w_4 and w_5 that are connected to the bus *Bus*3 sharing the common source (R port of the register file) via the wire w_s.

Using the control-unit formation described in Section 6.3.2, we can form the final supergraph, as shown in Figure 6.13. Similarly, a control-state table can be derived directly from the supergraph as described in Section 6.3.2.

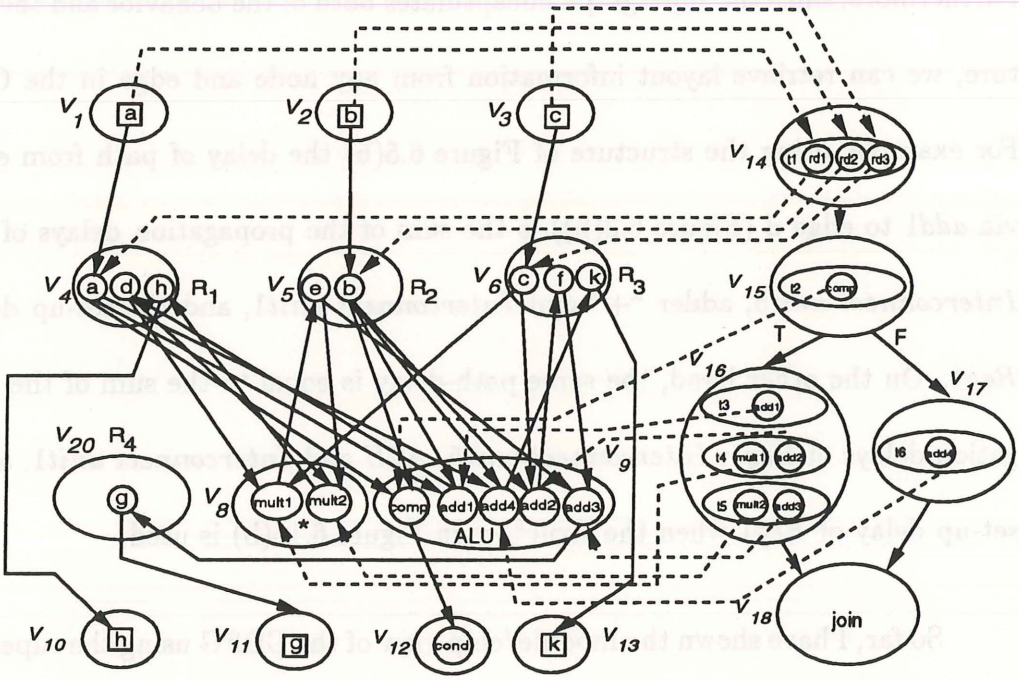
Using the datapath and control-unit formation methods, we can directly map the supergraph into the FSMD chip architecture, as described in Section 6.3.3. For

instance, the final supergraph in Figure 6.14(a) is divided into five parts: *Datapath section*, *Control section*, *Register file*, *Input ports* and *Output ports*. Figure 6.14(b) shows the chip formation of the supergraph in Figure 6.14(a).

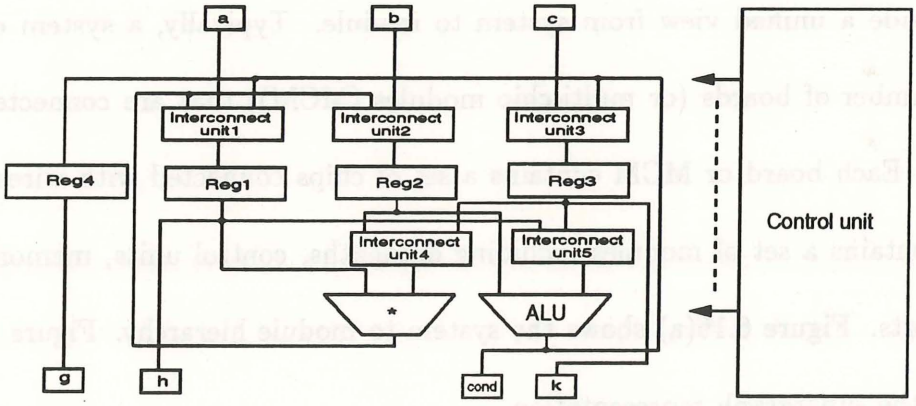
6.5 Extension: A Unified View From System To Module

In the previous two sections, we have shown that the supergraph is a structural representation by grouping nodes and edges of a CDFG. In the supergraph, a functional-unit/storage/port supernode represents a supernode (e.g., a register, a functional unit or a port) containing a cluster of operational nodes or dependency edges in a CDFG. A control supernode is a supernode containing a cluster of operational nodes that can be executed in the same control step. The alternatives of the design are represented using different supergraph configurations (e.g., different structural designs) that still encapsulate the same CDFG (i.e., behavior).

For instance, consider the Figure 6.3(c) example, we can group operation *comp* with other operations $\{add1, add2, add3, add4\}$ in supernode V_9 if we replace V_9 with an ALU, as shown in Figure 6.15(a). We can also partition variables e, b and g in supernode V_5 into two groups, e and b to V_5, and g to V_{20}.



(a)



(b)

Figure 6.15: Hypergraph modification: (a) the supergraph, (b) the structure.

This regrouping results in an additional register ($R4$) but the number of inputs of *Interconnect unit*2 is reduced from 3 to 2, as shown in Figure 6.15(b). Furthermore, since the supergraph encapsulates both of the behavior and the structure, we can retrieve layout information from any node and edge in the CDFG. For example, using the structure of Figure 6.5(b) the delay of path from edge a, via *add*1 to edge d (Figure 6.3(b)) is the sum of the propagation delays of *Reg*1, *Interconnect unit*5, adder "+" and *Interconnect unit*1, and the set-up delay of *Reg*1. On the other hand, the same path-delay is equal to the sum of the propagation delays of *Reg*1, *Interconnect unit*5, *ALU* and *Interconnect unit*1, and the set-up delay of *Reg*1 when the structure in Figure 6.15(b) is used.

So far, I have shown the module/chip view of the CDFG using the supergraph model. In the following section, I present the extension of the supergraph model to provide a unified view from system to module. Typically, a system consists of a number of boards (or multi-chip modules (MCM)) that are connected with cables. Each board or MCM contains a set of chips connected with wires. Each chip contains a set of modules including datapaths, control units, memories and I/O ports. Figure 6.16(a) shows the system to module hierarchy. Figure 6.16(b) shows the supergraph representation.

We can easily extend the supergraph model by adding: chip supernodes that consists of a set of modules (i.e., a set of functional-unit, control, storage and port supernodes) and board supernodes that consists of a set of chip supernodes. The

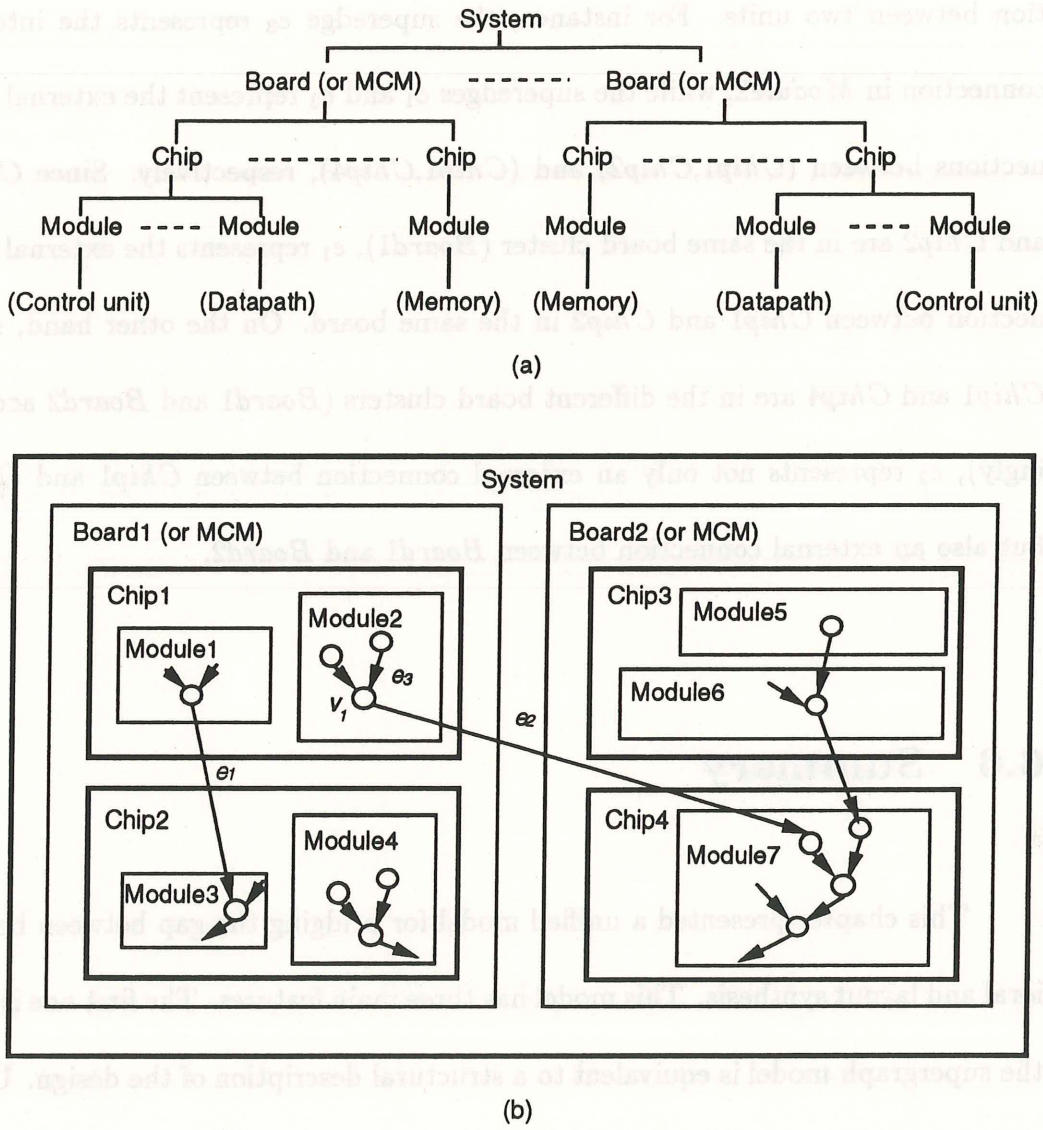


Figure 6.16: A unified view: (a) the system hierarchy, (b) a system to module view.

superedge connected two supernodes in the same cluster represents the internal connection in the same unit (e.g., module, chip or board), while the superedge connected two supernodes in the different clusters represents the external connection between two units. For instance, the superedge e_3 represents the internal connection in $Module2$, while the superedges e_1 and e_2 represent the external connections between ($Chip1$,$Chip2$) and ($Chip1$,$Chip4$), respectively. Since $Chip1$ and $Chip2$ are in the same board cluster ($Board1$), e_1 represents the external connection between $Chip1$ and $Chip2$ in the same board. On the other hand, since $Chip1$ and $Chip4$ are in the different board clusters ($Board1$ and $Board2$ accordingly), e_2 represents not only an external connection between $Chip1$ and $Chip4$ but also an external connection between $Board1$ and $Board2$.

6.6 Summary

This chapter presented a unified model for bridging the gap between behavioral and layout synthesis. This model has three main features. The first one is that the supergraph model is equivalent to a structural description of the design. Using the area and timing models described in Chapter 5, we can estimate the layout area and delay of the design from the supergraph representation. In addition, the supergraph model also provides a chip structure hierarchy of the design. The second feature is that the supergraph encapsulates the behavior (CDFG) of the design.

Using this supergraph model, we can retrieve layout information during the design process to support design decision making and design tradeoffs. The last and most important feature is that this model provides a unified behavior/structure view of the design that is well suited for interactive synthesis.

Chapter 7

Binding Using Layout Information

This chapter presents two layout-driven approaches, using the layout model and using the feedback layout information, for behavioral synthesis. To provide a fast area/timing measure in datapath design process, this chapter describes a new approach that combines our proposed area/timing model discussed in Chapter 5 and the unified model discussed in Chapter 6 for datapath optimization. We model the datapath as a graph representation that noticeably reflects the datapath floorplan and we also formulate datapath binding as a graph partitioning problem. Contrary to the other datapath optimization algorithms that minimize the number and size of registers and muxes, our algorithm evaluates layout-area quality during datapath optimization. Our approach provides faster and more accurate

area quality measures for datapath optimization than previous approaches. In addition, this chapter also presents an approach that uses back-annotation of layout information to estimate the clock cycle of the design.

The remainder of this chapter is organized in the following manner. Section 7.1 presents our unit-binding approach for datapath optimization. First, Section 7.1.1 defines the unit-binding problem. Then, Section 7.1.2 describes the area-cost function. Finally, Section 7.1.3 presents the unit-binding algorithm. Section 7.2 describes the back-annotation approach for clock estimation. Further, Section 7.3 presents the experimental results. Finally, Section 7.4 concludes our approach.

7.1 Unit Binding

Datapath synthesis consists of three interdependent binding tasks: functional-unit binding, storage-unit binding and interconnect-unit binding. Functional-unit binding determines the exact mapping of the operations into the functional units. Storage-unit binding maps data carriers, such as variables and constants, in the behavioral description to storage components. Interconnect-unit binding assigns interconnect units and wires between functional/storage units to connect data-transfer paths.

In the past, the number and size of functional units, registers, muxes (or equivalent 2-to-1 muxes), mux inputs and connections (or wires) were the commonly-used area measures in datapath synthesis. Consequently, a great deal of effort has been devoted to minimize the number and size of registers and muxes in datapath synthesis [ClTh90, DeNe89, DiTh89, HCLH90, LyEG90, Pang88, PaGa87, PaPM86, PaKG86, TsSi86]. However, these area measures assume that the layout area is directly proportional to the number and size of RT components and do not take into account layout technology factors, such as layout architectures or styles, component libraries, and the impact of floorplanning, placement and routing. These factors often greatly affect the final layout of the design.

In this section, we describe a unit binding algorithm that combines the layout model described in Chapter 5 and a graph representation for datapath optimization. Contrary to the other datapath binding algorithms which minimize the number and size of registers and muxes, our algorithm uses the combination of the layout model and unified representation to evaluate area quality during the datapath optimization.

7.1.1 Problem Definition

The objective of unit binding is to assign operations to functional units and to assign variables to storage units so that the total layout area is minimized.

We can formulate the unit-binding problem to a graph-partitioning problem, as follows: given a data-flow graph, its corresponding schedule, and a set of functional units, partition operations and variables into a set of supernodes, such that:

1. no two operations in the same control step can be assigned to the same functional-unit supernode,
2. no variables with overlapping lifetime can be assigned to the same storage-unit supernode, and
3. the total area of the supergraph is minimized.

7.1.2 Area Cost Function

Using this graph model, we can calculate the datapath area and control-unit area using the layout model described in Chapter 5. Using this model, we need two elements to compute the area: the number of transistors and the number of routing tracks.

The number of transistors of each RT component can be obtained directly from the target component library. To obtain the number of routing tracks required to completely connect all nets in one bit slice, we first implement stack placement using the KLFM [FiMa82, KeLi70] algorithm. Then we implement routing track assignments using the left-edge algorithm. Since the stack placement takes pseudo linear time and the routing-track estimation takes $O(n \log n)$

time where n is the number of nets in the RT-netlist, the complexity of the area calculation is $O(n \log n)$.

7.1.3 The Algorithm

The algorithm consists of two phases: initial assignment and interchange optimization. Initial assignment consists of two steps: supernode formation and superedge formation. In the first step, the algorithm determines the minimum number of registers required to store all variables using the left-edge algorithm [KuPa87] and assigns variables to corresponding registers. The algorithm then assigns operations to the given functional units arbitrarily, such that no more than one operation in the same control step will be assigned to the same functional unit. Finally, the algorithm forms the graph as described in Algorithm 6.1.

In the interchange optimization phase, the algorithm performs superedge merging by interchanging operations and variables that reside in the supernodes. superedge merging is important because it contributes to interconnect sharing. In the following section, we first describe two possible ways to merge superedges by interchanging variables or operations: node relocation and node swapping. Then, we describe the interchange technique by taking into account the interdependent relationship between operation and variable assignments.

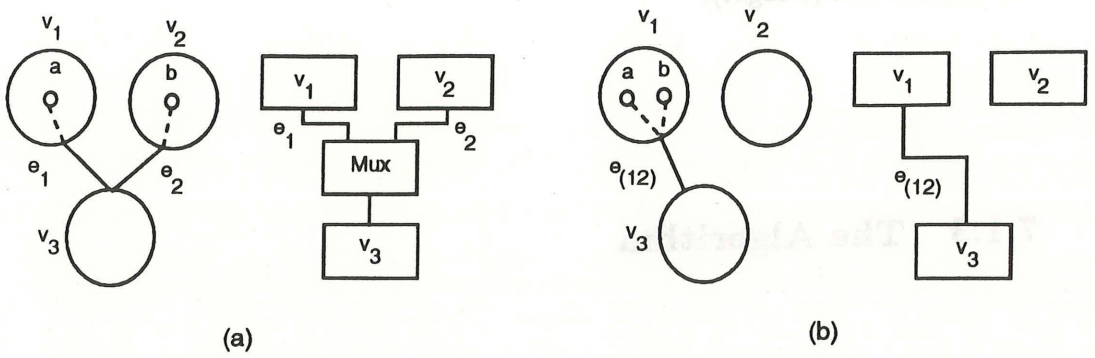


Figure 7.1: Superedge merging by node relocation: (a) before, (b) after.

The first possible way to merge superedges is to relocate variables between register supernodes or to relocate operations between operation supernodes. A variable can be relocated from a source register supernode to a destination register supernode if and only if: the destination supernode is free during the lifetime of that variable. An operation can be relocated from a source supernode to a destination supernode if and only if: (1) the destination supernode can performs the function of that operation, and (2) there does not exist another operation in the destination supernode such that this operation is assigned to the same control step as the relocating operation's. We term the above conditions the "relocation preconditions". The node relocation can be performed if and only if the relocation preconditions are satisfied that is termed as a "feasible relocation". Node relocation allows us to relocate one node, as well as a group of nodes at one time.

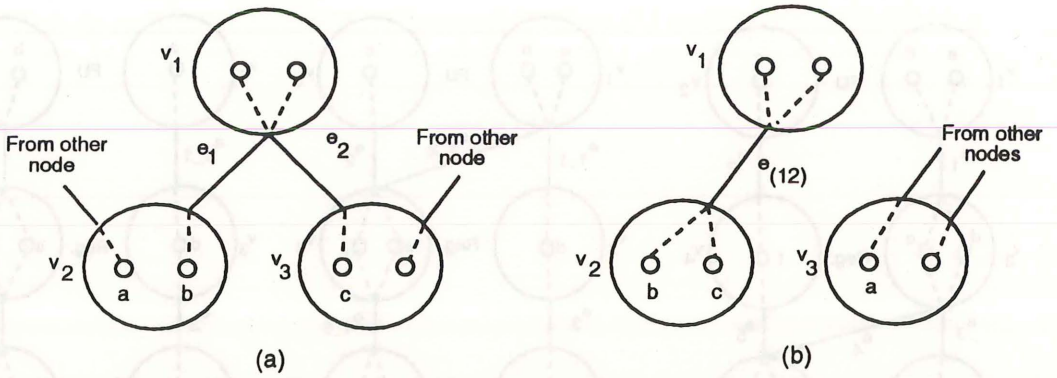


Figure 7.2: Node swapping.

As an example, consider the register supernode V_3 in Figure 7.1(a). V_3 is connected to V_1 and V_2 with the superedges e_1 and e_2, respectively. Since V_3 has to select one input from two sources V_1 and V_2, a 2-input Mux is required. If node b in V_2 can be moved to V_1, then e_1 and e_2 can be merged into e_{12}, as shown in Figure 7.1(b). As a result, V_3 does not need a mux for its input.

The second possible way for superedge merging is to swap the variables between register supernodes or to swap the operations between operation supernodes. Node swapping can be viewed as a two-way node relocation problem. Node relocation is performed as relocating nodes from the same source supernode to one or more destination supernodes if there exists a "feasible relocation". On the other hand, node swapping is performed when a one-way feasible relocation from a source supernode to a destination supernode can not be found, but a feasible relocation can be created by rearranging the nodes in the destination supernode.

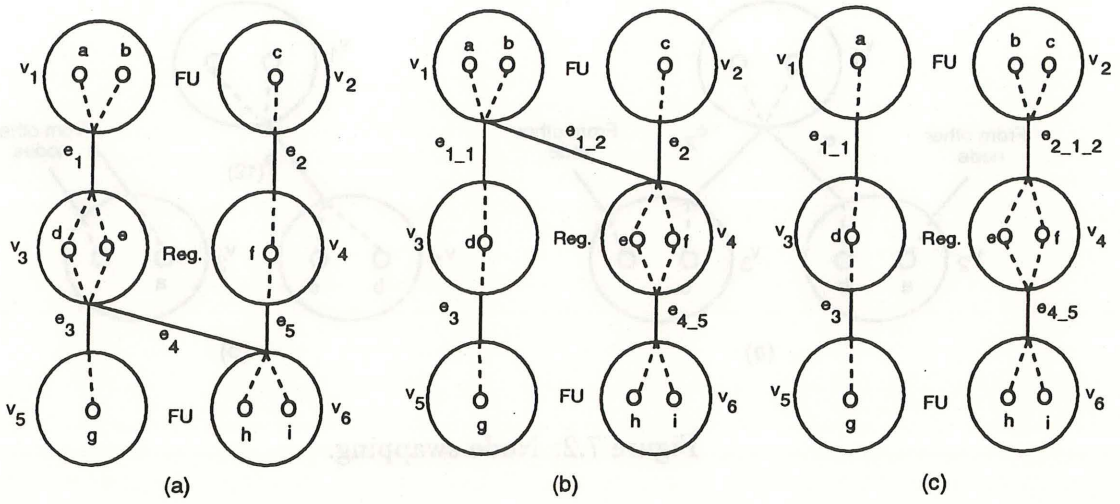


Figure 7.3: Interchange by considering interdependent relationship between operation and variable assignments.

In Figure 7.2(a), assume V_2 and V_3 are two operation supernodes; e_1 and e_2 can be merged by relocating node c from V_3 to V_2. If node a in V_2 is assigned to the same control step as node c's, then node c can not be relocated from V_3 to V_2. However, e_1 and e_2 can be merged by swapping node a and node c as shown in Figure 7.2(b).

In order to take into account the interdependent relationship between operation and variable assignments, the algorithm determines a group of "feasible relocation" nodes by rearranging variables in the registers and operations in the functional units simultaneously. Consider Figure 7.3(a), where e_4 and e_5 can be merged by relocating variable e from V_3 to V_4, so that the mux in front of V_6

can be eliminated. However, after relocating variable e, e_1 has to be split into two superedges e_{1_1} and e_{1_2} as shown in Figure 7.3(b) so that an additional mux is needed in front of V_4. However, if there is a feasible relocation of operation b from V_1 to V_2, then the algorithm finds a solution to achieve overall interconnect reduction. As a result, the algorithm relocates variable e and operation b to V_4 and V_2, respectively, as shown in Figure 7.3(c).

Algorithm 7.1 describes the unit binding for datapath synthesis. The input to the algorithm includes a given CDFG, the schedule and a set of given functional units. The algorithm first uses the left-edge-algorithm to assign variables into a set of registers (Procedure *left_edge_alg()*). Then, the algorithm assigns operations and variables into registers and functional units arbitrarily (Procedure *init_op_var_assignment()*). The procedure *Supergraph_Formation()* forms the supergraph, while the procedure *layout_estimation()* returns the area-cost of the given supergraph. Further, the algorithm locates the supernodes such that their inputs connect to more than one supernode (Procedure *locate_feasible_merging_superedge()*). The superedges connected to the inputs of these supernodes are called "feasibly-mergeable superedges". For a feasibly mergeable superedge, the algorithm locates a set of variables or operations associated with this superedge, rearranges the nodes (Procedure *relocate_node()*), and calculates the layout area as described in the previous section. If a smaller layout area is obtained, then a merging solution has

been found. If there is only one superedge connected to an input pin of a supernode, then a mux is not needed for this input pin. Thus, this superedge achieves the maximum "sharing". In this case, the algorithm will "lock" this superedge so that the algorithm will not consider this superedge as a feasibly mergeable superedge. The algorithm begins with the minimum number of registers and performs the allocation iteratively by incrementing the number of registers to explore the design space. For each iteration, the algorithm runs repeatedly until no more improvement can be found.

Algorithm 7.1. Unit Binding.

```
Let
    G = {V, E} be a CDFG;
    H={V,E} be a supergraph;
    "count" be a given arbitrary number;
    M be a set of feasible merging superedges;
    F be a set of given functional units;
    T be a set of schedule;
    P be the operation/variable assignments;
    R and R_x be a set of registers;
Unit_Binding(G,F,T,count){
    R_x = φ;
    while (count > 0) do
        R = left_edge_alg(G);
        P = init_op_var_assignment(G,S, R, S);
        /*See Algorithm 6.1*/
        H = Supergraph_Formation(G,F ∪ R ∪ R_x,T,P);
        old_area = layout_estimation(H);
        /*Interchange optimization*/
        no_more_improve = FALSE;
        while (no_more_improve = FALSE) do
            M = locate_feasible_merging_superedge(H);
            a_gain_merging = FALSE;
            for (∀ feasible_merging_superedge e ∈ M) do
                /*relocate nodes associated with superedge e*/
                H' = relocate_node(e);
```

```
                    new_area = layout_estimation(H');
                    if (new_area < old_area) do
                        H = H';
                        old_area = new_area;
                        a_gain_merging = TRUE;
                    endif
                endfor
                if (a_gain_merging = FALSE) then
                    no_more_improve = TRUE;
                endif
            endwhile
            /*incrementing one more register for next allocation iteration*/
            count = count - 1;
            if (count > 0) then
                R_x = R_x ∪ register;
            endif
        endwhile
}
```

Complexity analysis. Since the algorithm performs registers and selectors tradeoffs in several runs (outer **while** loop), we consider only one allocation run, which consists of three parts:

1. Using the left-edge algorithm, It takes $O(m \log m)$ time to determine the minimum number of registers and initial variable assignment, where m is the number of variables in the CDFG.

2. The complexity of *Supergraph_Formation* procedure is described in Algorithm 6.1.

3. The complexity of *layout_estimation* procedure is $O(n \log n)$ where n is the number of nets in the netlist.

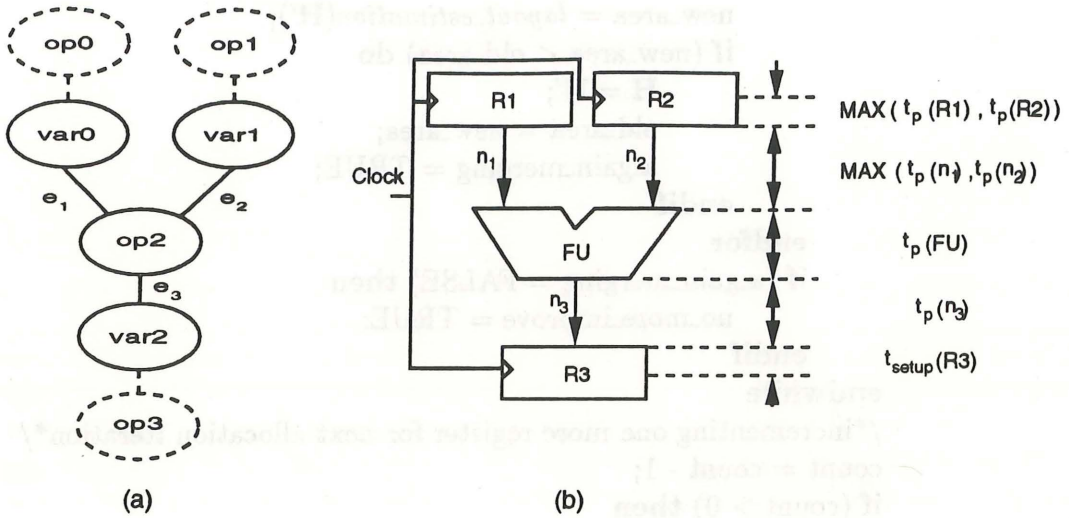


Figure 7.4: The register-to-register delay path: (a) *var* node insertion, (b) the structure.

4. In the interchange optimization procedure, it takes $O(pq)$ time to locate feasible merging superedges, where p is the number of superedges and q is the number of supernodes. For each feasible merging superedge, it takes $O(r + n \log n)$ to relocate nodes and estimate area, where r is the average number of variables or operations associated with the feasible merging superedge. Thus, each interchange optimization loop takes $O(pq + s(r + n \log n))$ time, where s is the average number of feasible merging superedge. In our experience, the local optimal (no_more_improve) state can be achieved in less than 20 iterations (interchange optimization **while** loop).

7.2 Back Annotation for Clock Estimation

Typically, the clock period is determined by the most critical register-to-register delay over all register-to-register paths. Figures 7.4(a) and (b) show a register-to-register path of a CDFG example and its corresponding structure. To realize the delay for each segment of a register-to-register path in a CDFG, we insert an internal node *var* on each edge of the CDFG, as shown in Figure 7.4(a). Hence, we can use Equation 5.19 (the register-to-register delay of an operation) to compute the datapath delay.

In the following of this section, we describe a walk through example (Figure 7.5). Figure 7.5(a) shows a data-flow graph example that is scheduled into five steps. Given an adder and a multiplier, Figure 7.5(b) shows the operation and variable assignments. Operations *add1, add2* and *add3* are assigned to the adder (*Adder*), and operations *sub1* and *sub2* are assigned to the subtracter (*Sub*). For the eight variables *a, b, c, d, e, f, g* and *h*, and we need three registers to store them as shown in Figure 7.5(b). Figure 7.5(c) the DFG after inserting *var* nodes, and the resulting structural graph and structure are shown in Figures 7.5(c) and (d).

To obtain layout information, we can use the area and timing models described in Chapter 5 or retrieve the layout information from the real layout. In this section, we describe how to back-annotate the propagation-delay of the datapath from the real layout. We assume the bit-width of the datapath is eight with

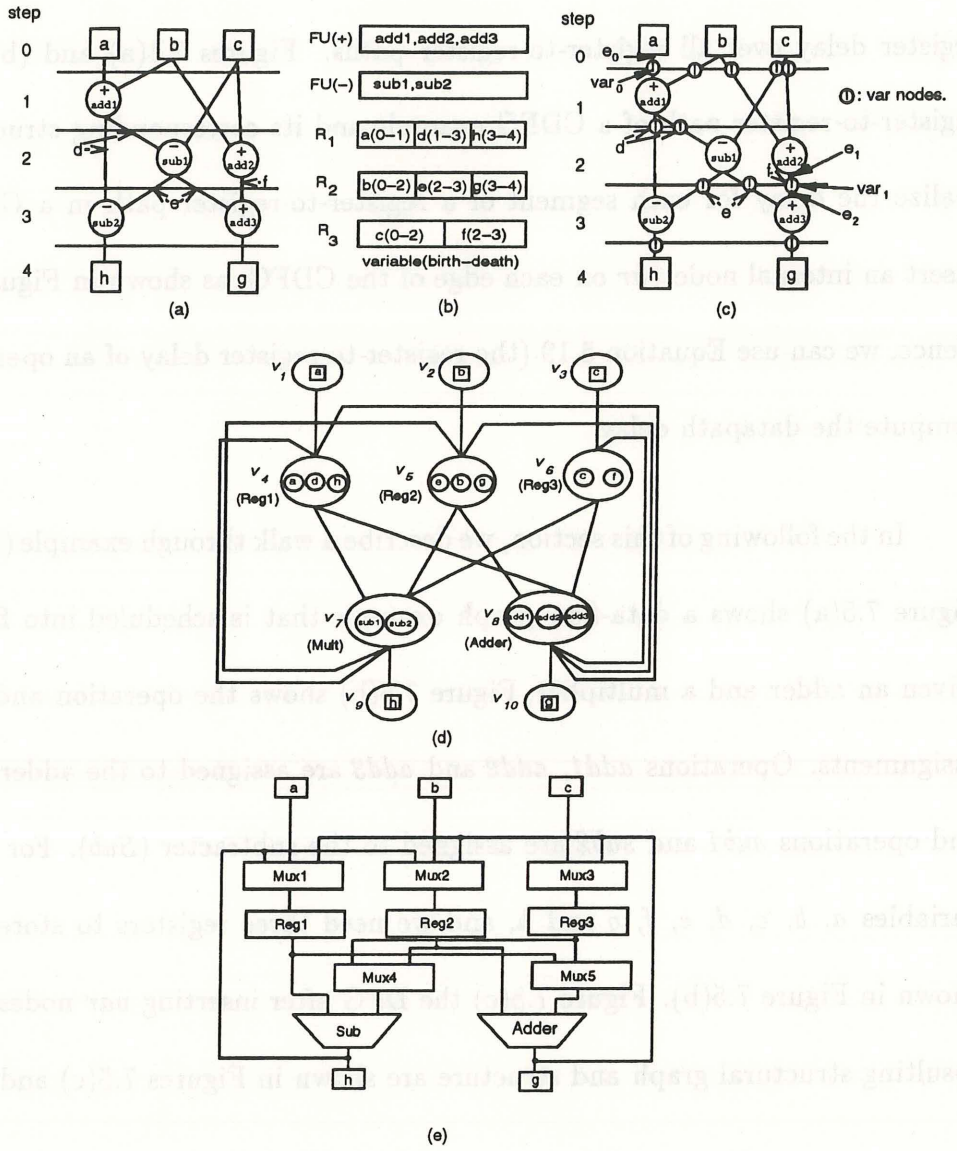


Figure 7.5: The back-annotation example: (a) DFG and schedule, (b) operation/variable assignments, (c) *var* node insertion, (d) the graph, (e) the structure.

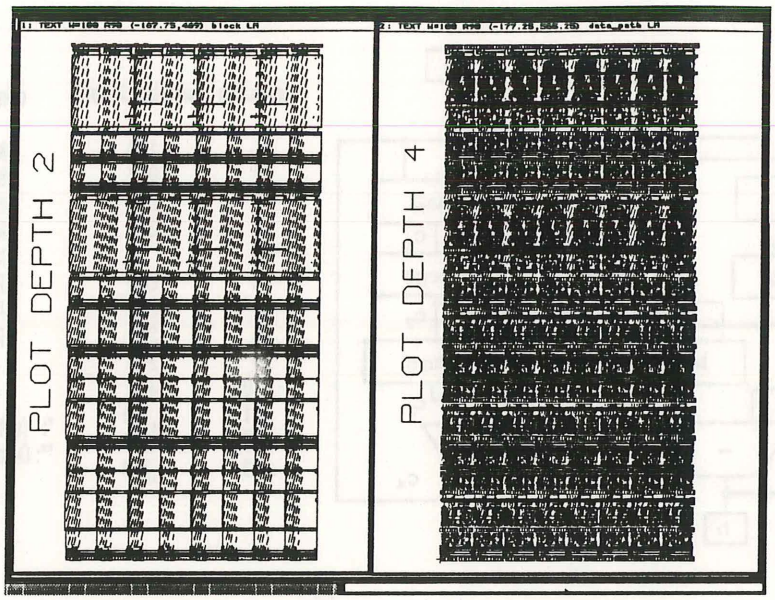
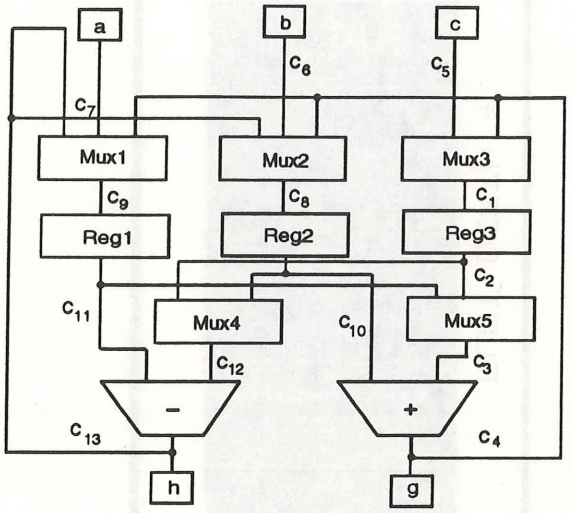


Figure 7.6: The layout of the back-annotation example (a) routing track assignments, (b) the final layout.

the bit-sliced stack layout architecture, as described in Chapter 3. The layout of the Figure 7.5 example is shown in Figure 7.6. Figure 7.7(a) shows the actual wire length and the delay of each component retrieved from the layout. Each edge delay is then computed. For example, the delay of edge e_1 Figure 7.7(b) is 5ns that is equal to the delay of $Mux1$. Based on the delay information shown in Figures 7.7(a) and (b), we can compute the register-to-register propagation delay of each operation using Equation 5.19. For instance, the delay of operation $add1$ is the sum of the maximum propagation delay of *var-node* 1 and *var-node* 2 (4.8ns), the maximum delay of e_2 and e_{10} (4.8ns), the delay of the adder (22ns), the maximum delay of e_3 and e_4 (5.3ns) and the set-up time of *var-node* 6 and

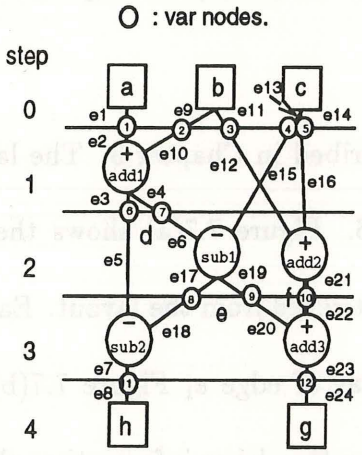


connection	wire length(um)
c1	130
c2	1,040
c3	260
c4	2,040
c5	10
c6	660
c7	280
c8	230
c9	230
c10	410
c11	1,160
c12	500
c13	1,760

component	delay(ns)
Mux1	5.0
Mux2	5.3
Mux3	5.2
Mux4	5.0
Mux5	4.8
Reg1	1/4.8 *
Reg2	1/4.2 *
Reg3	1/4.7 *
–	21.0 **
+	22.0 **

*: (set-up/propagation) delay.
**: Delay with 8-bit component.

(a)



connection	delay(ns)
e1	5.0
e2	4.8
e3	5.3
e4	5.3
e5	0.0
e6	0.0
e7	5.0
e8	0.0
e9	5.3
e10	0.0
e11	5.3
e12	5.3
e13	5.2
e14	5.2
e15	5.0
e16	4.8
e17	5.3
e18	5.0
e19	5.3
e20	0.0
e21	5.2
e22	4.8
e23	5.3
e24	0.0

var-node	delay(ns) (set-up/propagation)
1	1/4.8
2	1/4.2
3	1/4.2
4	1/4.7
5	1/4.7
6	1/4.8
7	1/4.8
8	1/4.2
9	1/4.2
10	1/4.7
11	1/4.8
12	1/4.2

(b)

Figure 7.7: (a) Back-annotation of wire lengths and component delays, (b) Back-annotation of delay information to the DFG.

var-node 7 (1ns), that is, the total datapath propagation-delay of operation *add*1 is 37.9ns. Using the same procedure, we can compute the delay for each operation and determine the worst register-to-register delay using Equation 5.21. Similarly, we can compute the control-unit delay, and thus compute the clock period using Equation 5.18.

One interesting observation from this example is that using this supergraph model we can back-annotate delay information to each node and edge in the CDFG. This is very useful in the interactive synthesis process because this detailed delay information can pinpoint the critical delay point (e.g., a particular wire or component) or delay path in the CDFG as well as the structure.

7.3 Experiments

We have implemented the previously described algorithm using C programming language on SUN4 workstations under UNIX. We have tested the binding algorithm on the elliptic filter benchmark with different schedules, including 17-step with 3-adder and 2-piped multiplier, 19-step with 2-adder and 2-multiplier, 21-step with 2-adder and 1-multiplier, and 19-step with 2-adder and 1-piped multiplier. Figure 7.8 shows the schedule of the 19-step with 2-adder and 1-piped

multiplier example. Figures 7.9, 7.10, 7.11 and 7.12 show the four different implementations of the same design as shown in Figure 7.8. The other examples can be found in [WuGa91].

We use the single-level multiplexer model. The transistor-pitch and wire-pitch coefficients (α and β) were calculated from the VTI 1.5-μm datapath library [VTI88]. The final layouts were generated using Mentor Graphics GDT tools. Figure 7.13 shows the datapath of a 16-bit elliptic filter using layout architectures I and II described in Chapter 5, in which architecture I uses 13 over-the-cell routing tracks for each bit slice. Figures 7.14 and 7.15 show the results of four different designs. Since the areas of multipliers for each design are the same, the areas shown in Figures 7.14 and 7.15 do not include the multiplier. Furthermore, the results only show the area of 1-bit datapath.

The results show that all implementations using architecture I require less than 13 actual routing tracks so that an extra routing area is not needed. Hence, the datapath area using architecture I is solely dependent on the number of transistors in the datapath. On the other hand, using layout architecture II both transistors and routing tracks contribute equally to the total area. The results also show that neither the design with the minimum number of registers nor the design with the minimum number of muxes can guarantee the minimum area. For instance,

(1) The designs with the minimum number of registers do not always produce the minimum area, such as:

(i) 21-step and architecture I (Figure 7.15(e)).

(ii) 19-step and 19-step with 2-adder and 1-piped multiplier, and architecture II (Figure 7.15(d)(h)).

(2) The designs with the minimum number of mux inputs do not always produce the minimum area, such as:

(i) 19-step with 2-adder and 1-piped multiplier, and architecture I (Figure 7.15(g)).

(ii) 21-step and architecture II (Figure 7.15(f)).

(iii) 17-step, architecture I and II (Figure 7.15(a)(b)).

(3) The design that produces the minimum area using layout architecture I does not guarantee the minimum area using layout architecture II. For example, the 21-step design (Figure 7.15(e)) with 11 registers and 27 mux inputs produces the minimum area using layout architecture I but not layout architecture II (Figure 7.15(f)).

To calculate the total area of the design including datapath, control unit and multiplier, we have experimented with a 16-bit, 19-step, 2-adder, and 1-piped multiplier elliptic filter example. Using datapath architecture I, we implemented two control-logic models, PLA and random logic, along with mux interconnect models. Figures 7.16 and 7.17 show a 16-bit elliptic filter example with PLA and random-logic implementations, respectively. Since the multiplier is treated as

a macrocell, the area of the multiplier is obtained directly from the component library. Figures 7.18 shows that using random-logic implementation the design with 13 register produces the minimum total area, while using PLA implementation the design with 11 registers produces the minimum total area. We have also used the described area and delay models to explore the design space of the elliptic filter benchmark. The results in Figure 7.19 shows that the 17-step design has fastest speed and largest area, while the 21-step design has the slowest speed and smallest area.

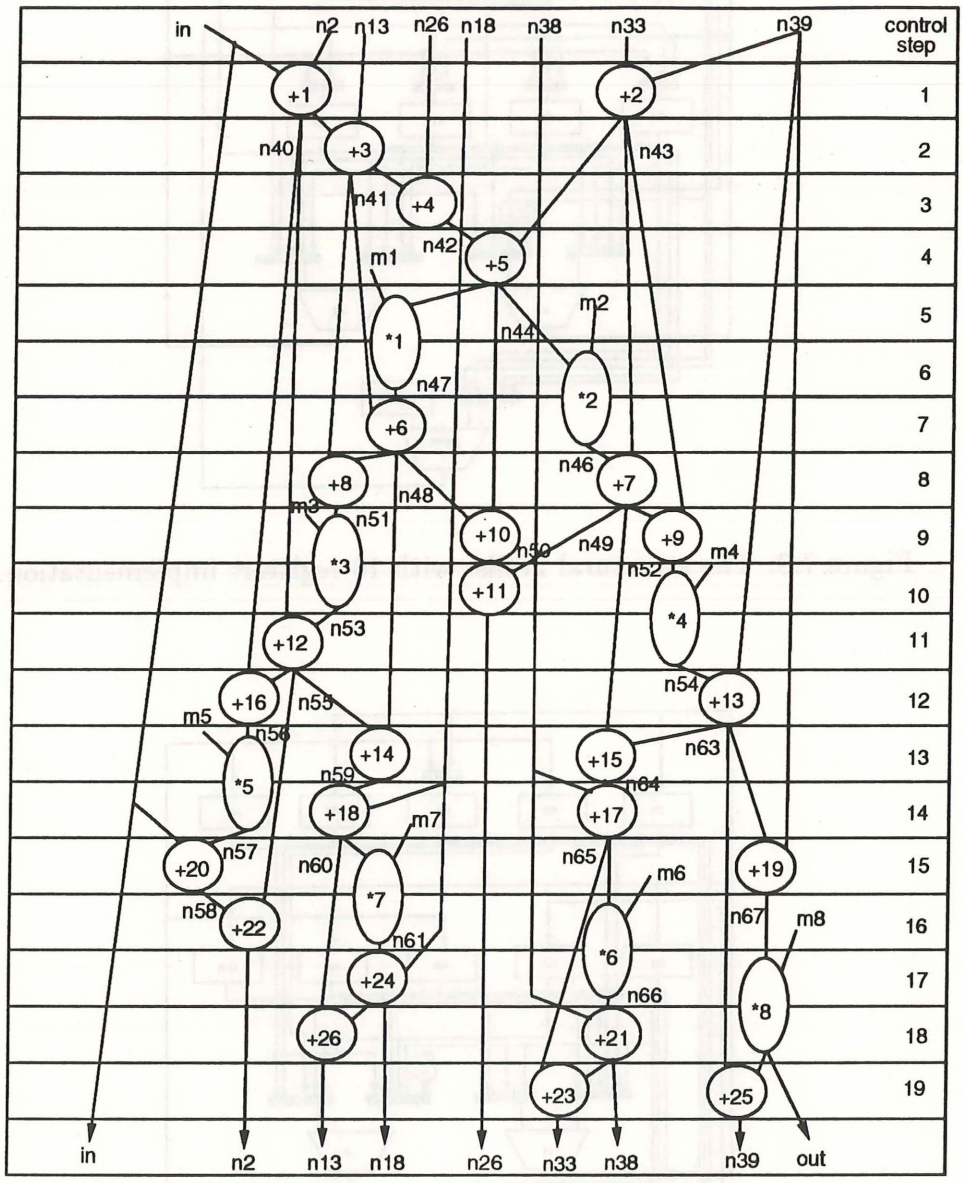


Figure 7.8: The schedule of the 19-step Elliptic Filter benchmark.

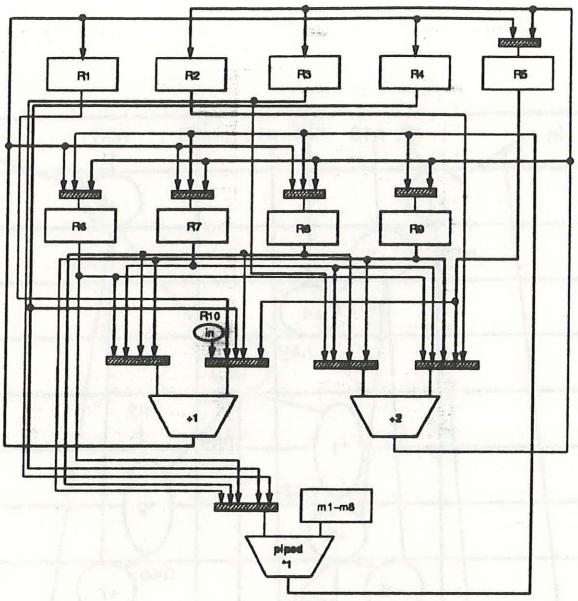


Figure 7.9: The structural netlist with 10 registers implementation.

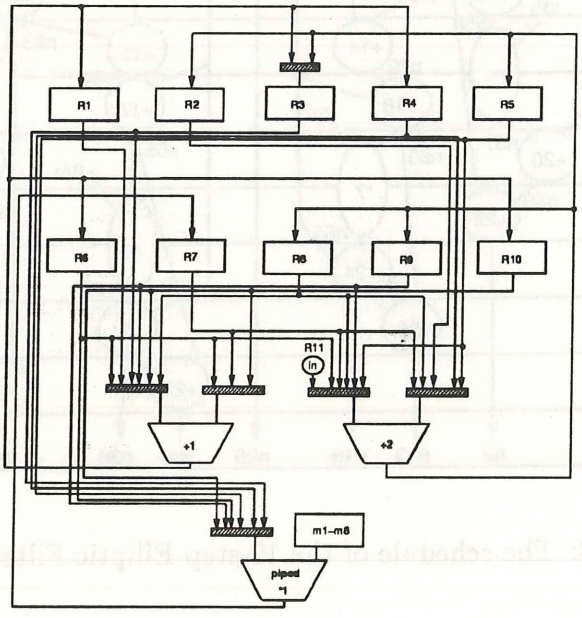


Figure 7.10: The structural netlist with 11 registers implementation.

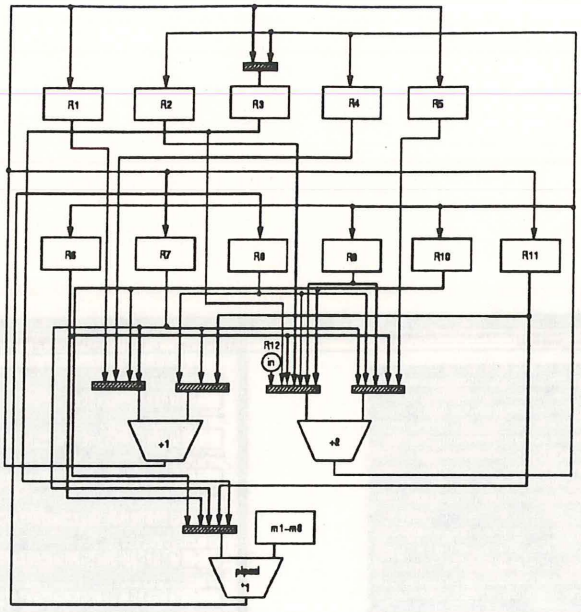


Figure 7.11: The structural netlist with 12 registers implementation.

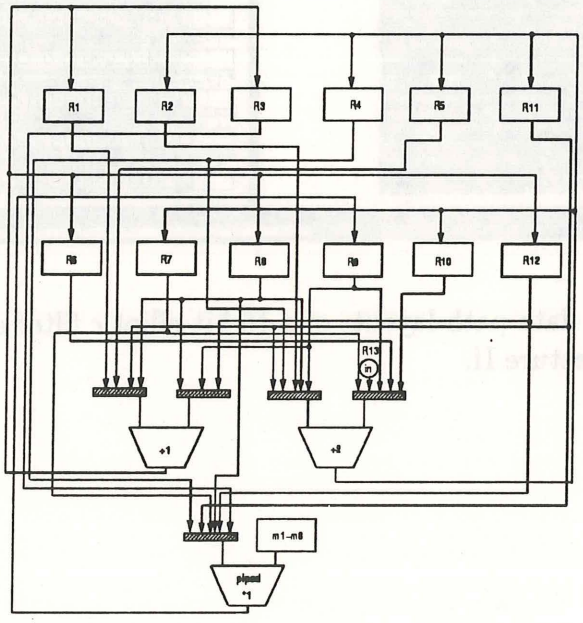


Figure 7.12: The structural netlist with 13 registers implementation.

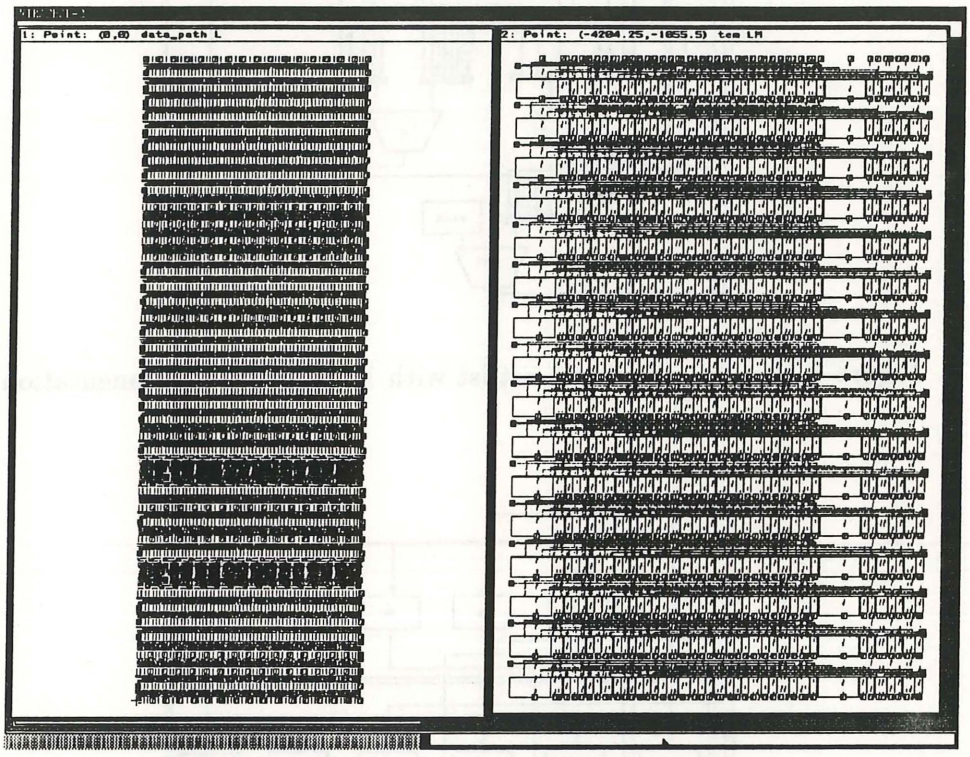


Figure 7.13: The data path layouts of a 16-bit elliptic filter example: (a) architecture I, (b) architecture II.

** Area not including multiplier

Control Steps	# of +	# of *	# of Reg.	# of Sel. / # Sel. Inputs	#trs.	#nets	#trks.	Layout Architecture I Actual Area (um^2 bit)	Layout Architecture II Actual Area (um^2 bit)
17	3	2-piped	10	11 / 34	552	27	11	136,720	193,117
17	3	2-piped	11	10 / 33	564	27	11	138,080	195,038
17	3	2-piped	12	8 / 31	572	27	11	138,480	195,490
17	3	2-piped	13	9 / 33	604	28	10	145,040	199,430

(a)

Control Steps	# of +	# of *	# of Reg.	# of Sel. / # Sel. Inputs	#trs.	#nets	#trks.	Layout Architecture I Actual Area (um^2 bit)	Layout Architecture II Actual Area (um^2 bit)
19	2	2	10	8 / 30	472	23	10	113,440	156,420
19	2	2	11	6 / 28	480	22	9	113,760	151,726
19	2	2	12	6 / 28	500	23	9	117,280	156,862
19	2	2	13	6 / 29	524	24	9	122,080	163,282

(b)

Control Steps	# of +	# of *	# of Reg.	# of Sel. / # Sel. Inputs	#trs.	#nets	#trks.	Layout Architecture I Actual Area (um^2 bit)	Layout Architecture II Actual Area (um^2 bit)
21	2	1	10	7 / 30	480	20	8	113,536	146,464
21	2	1	11	5 / 27	480	19	10	111,456	151,836
21	2	1	12	5 / 28	504	19	9	115,296	152,934
21	2	1	13	6 / 31	540	23	9	126,736	168,235

(c)

Control Steps	# of +	# of *	# of Reg.	# of Sel. / # Sel. Inputs	#trs.	#nets	#trks.	Layout Architecture I Actual Area (um^2 bit)	Layout Architecture II Actual Area (um^2 bit)
19	2	1-piped	10	10 / 36	508	23	10	125,376	170,976
19	2	1-piped	11	6 / 28	482	20	9	113,136	150,045
19	2	1-piped	12	6 / 26	492	21	8	115,696	149,272
19	2	1-piped	13	5 / 23	490	21	8	113,696	146,672

(d)

Figure 7.14: The results of the Elliptic Filter example: part 1.

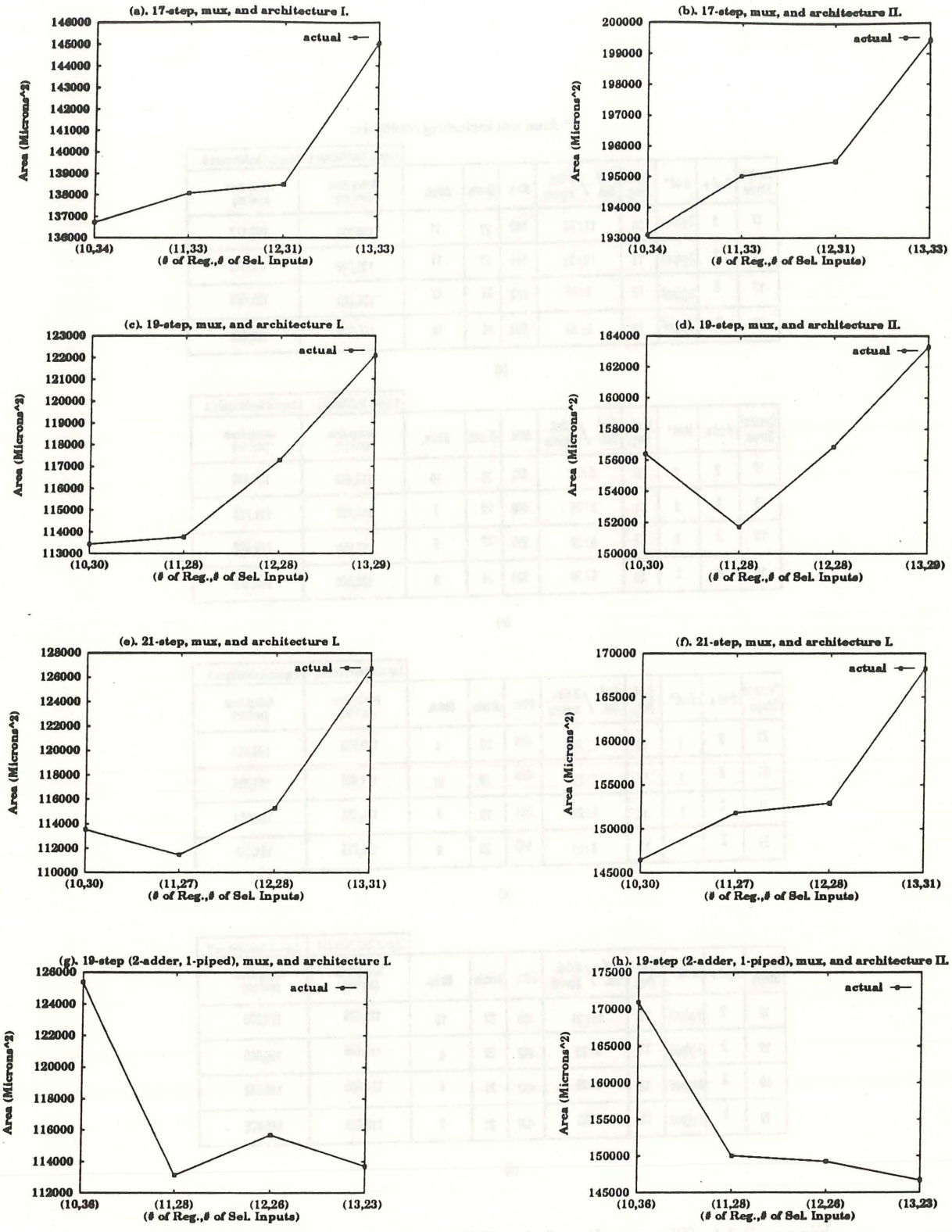


Figure 7.15: The results of the Elliptic Filter example: part 2.

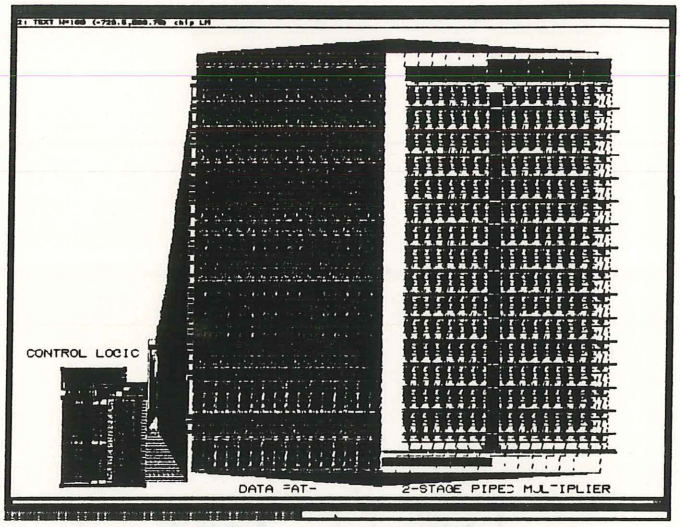


Figure 7.16: The final layout of a 16-bit elliptic filter example with PLA implementation.

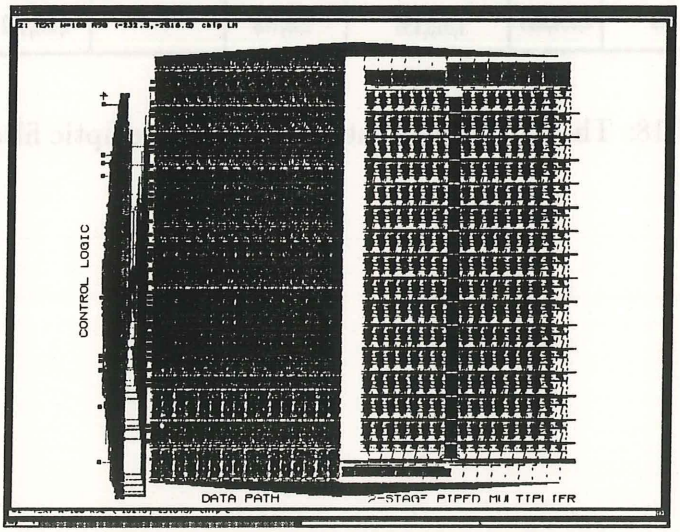


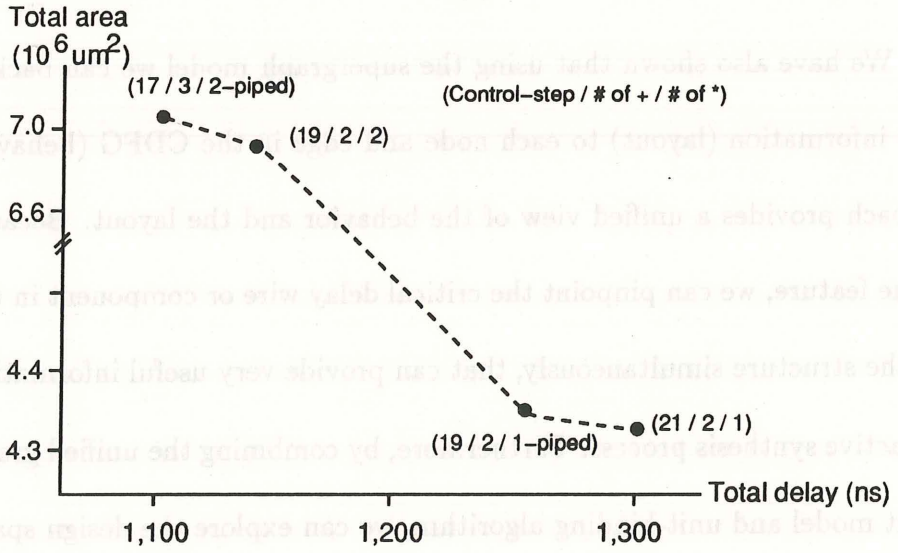
Figure 7.17: The final layout of a 16-bit elliptic filter example with random-logic implementation.

# of Reg.	# of Sel. / # Sel. Inputs	Multiplier	Architecture I	Control Logic		Total Area	
			Datapath	PLA	Random Logic	PLA	Random Logic
		Area (um^2)	Area (um^2)	Area (um^2)	Area (um^2)	Area (um^2)	Area (um^2)
10	10 / 36	2,330,880	2,006,016	312,256	255,352	4,649,152	4,592,248
11	6 / 28	2,330,880	1,810,176	267,540	230,082	4,408,596	4,371,138
12	6 / 26	2,330,880	1,851,136	266,228	196,616	4,448,244	4,378,632
13	5 / 23	2,330,880	1,819,136	259,116	200,889	4,409,132	4,350,905

Figure 7.18: The overall area estimation of the elliptic filter example.

# of steps	# of + / # of *	# of regs / # of sel. / # Sel. inputs	Clock (ns)	Total execution time (ns)	Total area (um²)
17	3 / 2–piped	12 / 8 / 31	64.8	1,102	7,074,632
19	2 / 2	11 / 6 / 28	60.1	1,142	6,682,152
21	2 / 1	11 / 5 / 27	62.0	1,302	4,329,526
19	2 / 1–piped	13 / 5 / 23	66.5	1,263	4,350,905

(a)



(b)

Figure 7.19: The area-time curve of four different design of the elliptic filter benchmark: (a) table, (b) AT-curve.

7.4 Conclusions

This chapter presented a new unit-binding approach that uses the supergraph and layout-area model for design-quality evaluation during datapath optimization. We have shown that datapath optimization by minimizing the number and size of registers or muxes do not always guarantee the minimum area. Since our proposed approach is technology independent, our algorithm can evaluate design quality and select the minimum area design using different component libraries and layout architectures.

We have also shown that using the supergraph model we can back-annotate delay information (layout) to each node and edge in the CDFG (behavior). This approach provides a unified view of the behavior and the layout. Because of this unique feature, we can pinpoint the critical delay wire or component in the CDFG and the structure simultaneously, that can provide very useful information for the interactive synthesis process. Furthermore, by combining the unified graph model, layout model and unit-binding algorithm, we can explore the design space.

Chapter 8

Conclusions

8.1 Summary of Contributions

This dissertation presented an approach to chip synthesis. The essential issues involved in chip synthesis, including target architecture, layout-synthesis method, design model and integration techniques between behavioral and layout synthesis, were presented.

First, a sliced-layout architecture was presented for generalized register-transfer (RT) netlists. Using the sliced-layout architecture, a layout-synthesis system was developed for layout generation from RT netlists. The system introduced a new partitioning approach that considers the component layout-style, floorplan and critical paths simultaneously to improve the overall area utilization and to minimize the critical wire length.

Second, to obtain more realistic quality measures for behavioral synthesis, a layout model, including area and timing models, was presented. This layout model considered most technology factors and thus produced more accurate estimates than previously proposed models. This research also presented two different views of quality measures, "accuracy" and "fidelity". When the accuracy of estimates is the main concern, the layout model should be formulated using the actually implemented algorithms or should be evaluated by running the actually implemented algorithms. On the other hand, when the estimation turn-around time is the main concern, a simple and fast estimate with high fidelity is needed. In the chip synthesis, we can mix these two types of quality measures for design evaluation, such that we use the fast and high-fidelity estimates for design tradeoffs and use the slow but accurate estimates to evaluate the design quality.

Third, a unified model was developed to bridge the gap between behavioral and structural descriptions. This model encapsulates both behavior and structure of the design that provides a unified behavior/structure view of the design. Hence, using this unified model, synthesis tools can retrieve layout information at any design level to support design decision making and design tradeoffs.

Finally, two methods, layout-model driven and feedback driven, were presented to incorporate layout information into behavioral synthesis. A unit-binding approach combining the supergraph model and layout model was presented for datapath optimization. The experiments showed that the previous algorithms which

minimize the number and size of registers and muxes do not always guarantee the minimum area. Contrary to previous algorithms that minimize the number and size of units, our approach can evaluate the area quality of the design and select the minimum area design. In addition, a back-annotation method was presented that can pinpoint the critical delay wire or component in the behavioral description. This feature is very useful for interactive synthesis.

8.2 Future Work

While the essential issues of integration of behavioral and layout synthesis for chip design are addressed in this research, a number of issues and improvements need to be studied further. First, at the layout-synthesis level, more sophisticated bit-sliced stack partitioning techniques, such as interleaved folding, are needed to improve stack area utilization. An I/O-pad placement and routing algorithm is needed in order to generate a complete chip.

Second, there are many open quality-measure problems that need to be explored. The area measures for specific layout architectures, such as gate array, sea of gates and FPGA, should be studied further. In order to improve the accuracy of area/performance measures, the impact of control-logic optimization needs to be taken into account. Furthermore, more extensive empirical study is needed to objectively establish our proposed layout model.

Third, using the unified representation and layout model, different behavioral-synthesis tasks, including module selection, scheduling and allocation, need to be developed. In addition, system-level partitioning scheme using the proposed unified representation and layout model should be studied further.

Bibliography

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

[BrGa90] F. Brewer and D.D. Gajski, "Chippe: A System for Constraint Driven Behavioral Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 7, pp. 681–695, July, 1990.

[ChGa90] G. D. Chen and D. D. Gajski, "An Intelligent Component Database System for Behavioral Synthesis," *Proceedings of the 27th Design Automation Conference*, pp.150–155, 1990.

[ChWG91] V. Chaiyakul, A. C-H Wu and D. D. Gajski, "Timing Models for High-Level Synthesis," Info. & Computer Science Dept., UCI, Tech. Rep. 91–70, 1991.

[ClTh90] R. J. Cloutier and D. G. Thomas, "The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm," *Proceedings of the 27th Design Automation Conference*, pp. 71–76, 1990.

[CNSD90] H. Cai, S. Note, P. Six and H. De Man, "A Data Path Layout Assembler for High Performance DSP Circuits," *Proceedings of the 27th Design Automation Conference*, pp.306–311, 1990.

[DeNe89] S. Devadas and A. R. Newton, "Algorithms for Hardware Allocation in Data Path Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-8, no. 7, pp. 768–781, 1989.

[DiTh89] E. Dirkes Lagnese and D. E. Thomas, "Architectural Partitioning for System Level Design," *Proceedings of the 26th Design Automation Conference*, pp. 62–67, 1989.

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

[DRSC86] H, De Man, J. Rabaey, P. Six and L. Claesen, "CATHEDRAL-II: A Silicon Compiler for Digital Signal Processing," *IEEE Design and Test*, 1986.

[FiMa82] C.M. Fiduccia and R.M. Mattheyses, "A Linear-Time Heuristic for Improving Network Partitions," *Proceedings of the 19th Design Automation Conference*, pp. 175–181, 1982.

[GDT89] "GDT Database and Language Tools," Silicon Compiler System, Sec. 7, V. 4.0, 1989.

[GDWL92] D. D. Gajski, N. Dutt, A C-H Wu and Y-L Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

[HaSt71] A. Hashimoto and J. Stevens, "Wire Routing by Optimizing Channel Assignment within Large Apertures," *The 8th Design Automation Conference Workshop*, pp. 155–169, 1971.

[HCLH90] C. Y. Huang, Y. S. Chen, Y. L. Lin and Y. C. Hsu, "Data Path Allocation Based on Bipartite Weighted Matching", *Proceedings of the 27th Design Automation Conference*, pp. 499–504, 1990.

[Hilf85] P. N. Hilfinger, "A High-Level Language and Silicon Compiler for Digital Signal Processing," *Proceedings of Custom Integrated Circuit Conference*, 1985.

[JaJe85] R. Jamier and A. Jeraya, "APOLLON: A Datapath Compiler," *Proceedings of the International Conference on Computer Design*, 1985.

[Joha79] D. L. Johannsen, "Bristle Blocks: A Silicon Compiler," *Proceedings of the 16th Design Automation Conference*, pp.310–313, 1979.

[John67] S.C. Johnson, "Hierarchical Clustering Schemes," *Psychometrika*, pp. 241–254, September, 1967.

[KeLi70] K.H. Kernighan and S. Lin, "An Efficient Heuristic Procedure for Partitioning Graph," *Bell System Technical Journal,* vol. 49, no. 2, pp. 291–307, February, 1970.

[Knap89] D. W. Knapp, "Feekback-Driven Datapath Optimization in Fasolt," *Proceedings of the International Conference on Computer-Aided Design*, pp. 300–303, 1989.

[KuPa87] F.J. Kurdahi and A.C. Parker, "REAL: A Program for Register Allocation," *Proceedings of the 24th Design Automation Conference*, pp. 210–215, 1987.

[KuPa89] F.J. Kurdahi and A.C. Parker, "Techniques for Area Estimation of VLSI Layouts," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 8, No.1, pp. 81–92, January 1989.

[KuRa91] F.J. Kurdahi and C. ramachandran, "LAST: A LAyout Area and Shape function esTimator for High Level Applications," *Proceedings od The European Conference on Design Automation*, pp. 351–355, 1991.

[LaGW91] Lawrence L. Larmore, D. D. Gajski and Allen C-H Wu, "Layout Placement for Sliced Architecture," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 11, no. 1, pp. 102–114, 1992.

[LiGa87] Y. L. Lin and D. D. Gajski, "LES: A Layout Expert System," *Proceeding of the 24th Design Automation Conference*, pp.672–678, 1987.

[LiGa88] J. S. Lis and D. D. Gajski, "Synthesis from VHDL," *Proceedings of the International Conference on Computer Design*, pp.378–381, 1988.

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

[LyEG90] T. A. Ly, W. L. Elwood, and E. F. Girczyc, "A Generalized Interconnect Model for Data Path Synthesis," *Proceedings of the 27th Design Automation Conference*, pp.168–173, 1990.

[McFa86] M.C. McFarland, "Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions," *Proceedings of the 23rd Design Automation Conference*, pp. 474–480, 1986.

[McKo90] M.C. McFarland and T.J. Kowalski, Incorporating Bottom-Up Design into Hardware Synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 9, no. 9, pp. 938–950, 1990.

[NGCD91] S. Note, W. Geurts, F. Catthoor and H. De Man, "Cathedral-III: Architecture-Driven High-Level Synthesis for High Throughput DSP Applications," *Proceedings of the 28th Design Automation Conference*, pp. 597–602, 1991.

[Pang88] B. M. Pangrle, "Splicer: A heuristic Approach to Connectivity Binding," *Proceedings of the 25th Design Automation Conference*, pp. 536–541, 1988.

[PaGa87] B. M. Pangrle, and D. D. Gajski, "Design Tools for Intelligent Silicon Compilation", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD–6 no. 6, pp. 1098–1112, 1987.

[PaKG86] P. G. Paulin, J. P. Knight and E. F. Girczyc, "HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis," *Proceedings of the 23rd Design Automation Conference*, pp. 263–270, 1986.

[PaPM86] A. C. Parker, J. Pizarro and M. Mlinar, "MAHA: A Program for Datapath Synthesis," *Proceedings of the 23rd Design Automation Conference*, pp. 461–466, 1986.

[PePr89] M. Pedram and B. Preas, "Interconnection Length Estimation for Optimized Standard Cell Layouts," *Proceedings of the International Conference on Computer-Aided Design*, pp. 390–393, 1989.

[PeRu81] P. Penfield Jr. and J. Rubenstein, "Signal Delay in RC Tree Networks," *Proceedings of the 18th Design Automation Conference*, pp. 613–617, 1981.

[PWSE86] B. R. Petersen, B. A. White, D. J. Salomon and M. I. Elmasry, "SPIL: A Silicon Compiler with Performance Evaluation," *Proceedings of the International Conference on Computer-Aided Design*, pp. 500–503, 1986.

[RaPB85] J. Rabaey, S. Pope and R. Brodersen, "An integrated Automatic Layout Generation System," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD–4, pp. 285–296, 1985.

[RiHi88] K. Rimey and P. N. Hilfinger, "A Compiler for Application-Specific Signal Processors," *VLSI Signal Processing*, pp. 341–351, 1988.

[RDVG88] J. Rabaey, H. De Man, J. Vanhoof, G. Goossens and F. Catthoor, "Cathedral-II: A Synthesis System for Multiprocessor DSP Systems," in *Silicon Compilation*, (Gajski, D.D. editor) Addison-Wesley Publishing Co., pp. 311–360, 1988.

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

[Sout83] J. R. Southard, "MacPitts: An Approach to Silicon Compilation," *Computer*, vol. 16, no. 12, pp. 74–82, 1983.

[Shun91] C. B. Shung et al., "An Integrated CAD System for Algorithm-Specific IC Design," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 10, no. 4, pp. 447–463, 1991.

[TrDi89] M. T. Trick and S. W. Director, "LASSIE: Structure to Layout for Behavioral Synthesis Tool," *Proceedings of the 26th Design Automation Conference*, pp. 104–109, 1989.

[TsSi86] C. J. Tseng and D. P. Siewiorek, "Automated Synthesis of Data Path in Digital Systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD–5, no.3, pp. 379–395, 1986.

[TLWN90] D.E. Thomas, E.D. Lagnese, R.A. Walker, J.A. Nestor, J.V. Rajan and R.L. Blackburn, *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench*, Kluwer Academic Publishers, Boston, 1990.

[VTI88] "Data path Library," VLSI Technology, INC., 1988.

[WuGa89] Allen C-H Wu and D. D. Gajski, "SLAM: An Automated Structure to Layout Synthesis System," Info. & Computer Science Dept., UCI, Tech. Rep. 89–40, 1990.

[WuCG90] Allen C-H Wu, G. D. Chen and D. D. Gajski, "Silicon Compilation from Register-Transfer Schematics," *Proceeding of International Symposium on Circuits and Systems*, pp.2576–2579, 1990.

[WuGa90] Allen C-H. Wu and D.D. Gajski, "Partitioning Algorithms for Layout Synthesis from Register-Transfer Netlists," *Proceedings of the International Conference on Computer-Aided Design*, pp. 144–147, 1990.

[WuCG91] A. C-H Wu, V. Chaiyakul and D. D. Gajski, " Layout Area Models for High-Level Synthesis," *Proceedings of the International Conference on Computer-Aided Design*, 1991.

[WuGa91] Allen C-H Wu and D. D. Gajski, "Layout-Driven Allocation for High Level Synthesis," Info. & Computer Science Dept., UCI, Tech. Rep. 91–30, 1991.

[Zimm88] G. Zimmermann, "A New Area and Shape Function Estimation Technique for VLSI Layouts," *Proceedings of the 25th Design Automation Conference*, pp. 60–65, 1988.