CONCURRENT HIGH-LEVEL SYNTHESIS WITH FLOORPLANNING

Jen-Pin Weng

CENG Technical Report 93-29

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4476

July 1993

UNIVERSITY OF SOUTHERN CALIFORNIA
THE GRADUATE SCHOOL
UNIVERSITY PARK
LOS ANGELES, CALIFORNIA 90007

This dissertation, written by

Jen-Pin Weng

under the direction of his *Dissertation Committee, and approved by all its members, has been presented to and accepted by The Graduate School, in partial fulfillment of requirements for the degree of*

DOCTOR OF PHILOSOPHY

....
Dean of Graduate Studies

Date ....

DISSERTATION COMMITTEE

Alice C. Parker
Chairperson

Massoud Pedram

Ming-Deh Huang

Dedication

To my mother.

Acknowledgements

I would like to take this opportunity to express my sincere appreciation to my advisor, Professor Alice C. Parker, for her valuable guidance and continued encouragement. Without her, this thesis would not exist.

I would also like to thank Professors Ming-Deh Huang and Massoud Pedram for serving on my dissertation committee, and Professors Melvin A. Breuer, Sarma Sastry and Gerald G. Medioni for serving on my guidance committee.

I thank all my friends and colleagues in USC for their warm friendship and various help. In particular, I like to mention Yung-Hua Hung, Chih-Tung Chen, Pravil Gupta, Shiv Prakash, Sen-Pin Lin, Atul Ahuja, Charles Njinda, Mary Zittercob and Donnalyn Combest.

Finally, I would like to thank my mother, Kuei-Ing, for her patience, support, and sacrifices throughout my graduate study. This research was supported by Semiconductor Research Corporation (Contracts 89-DJ-075 and 92-DJ-075). I would like to thank them for the support.

Contents

Acknowledgements iii

List Of Figures vii

List Of Tables xi

Abstract xiii

1 Introduction 1
- 1.1 Behavioral Synthesis 1
  - 1.1.1 Data Path Synthesis 2
  - 1.1.2 Control Path Synthesis 2
- 1.2 Motivation and Problem Approach 3
- 1.3 Thesis Organization 5

2 Related Research 6
- 2.1 Data Path Synthesis 6
  - 2.1.1 Prediction Methods 7
  - 2.1.2 Floorplanning Approaches 8
- 2.2 Control Path Synthesis 9
- 2.3 Thermal Analysis and Synthesis 10

3 Overview of the Synthesis System Research 12
- 3.1 Limitations 12
- 3.2 Clocking Scheme 15
- 3.3 An Overview of Data Path Synthesis 17
- 3.4 An Overview of Control Path Synthesis 20
  - 3.4.1 Controller for Designs without Conditional Branches 23
  - 3.4.2 Controller for Designs with Conditional Branches 23

4 Preliminary 3D Scheduling Research 28
- 4.1 Algorithm Description 30

4.1.1 Scheduling and Module Assignment 30
4.1.2 Improvement Procedure 32
4.2 Experimental Results 32

5 Data Path Synthesis **40**
5.1 Force-Directed Scheduling 41
5.2 Prediction Technique 44
5.2.1 Operator Estimation 45
5.2.1.1 An Operator Lower-Bound Estimation for Pipelined Designs 45
5.2.1.2 An Operator Lower-Bound Estimation for Non-Pipelined Designs 46
5.2.2 Wiring Estimation 50
5.2.2.1 Wiring Area Estimation 50
5.2.2.2 Wiring Delay Estimation 51
5.2.3 Controller Estimation 52
5.2.3.1 PLA Abstract Characteristic Estimation 53
5.2.3.2 Controller Area Estimation 54
5.2.3.3 Controller Delay Estimation 55
5.3 Floorplanning Technique 57
5.3.1 Incremental Cluster Tree Generation 60
5.3.2 Shape Function Computation 63
5.4 3D Scheduling Technique 68
5.4.1 Timing Model Assumptions 68
5.4.2 Loop Model 70
5.4.3 3D Scheduling Algorithm 73
5.4.4 Complexity Analysis of the 3D Scheduling Algorithm 79
5.4.5 Discussion 80

6 Data Path Synthesis Experiments and Results **81**
6.1 Introduction 81
6.2 A Non-Pipelined FIR Filter Example 82
6.3 A Pipelined FIR Filter Example 94
6.4 A Non-Pipelined Differential Equation Solver Example 97
6.5 Non-Pipelined Elliptic Filter Examples 101
6.5.1 A 12-Time-Step Non-Pipelined Elliptic Filter Example . . . 103
6.5.2 A 10-Time-Step Non-Pipelined Elliptic Filter Example . . . 111
6.6 Robot Arm Controller Examples 111
6.7 An Inner Loop Example 117

7 Control Path Synthesis **122**
7.1 Controller Assumptions 123

7.2 Control Signal Generation 124
7.3 Designs without Conditional Branches 125
7.3.1 Controllers for Non-Pipelined Designs 125
7.3.2 Controllers for Pipelined Designs 127
7.4 Controllers Using Status Registers 129
7.4.1 Controllers for Non-Pipelined Designs 130
7.4.2 Controllers for Pipelined Designs 134
7.5 Controllers without Status Registers 138
7.5.1 Controllers for Non-Pipelined Designs 138
7.5.2 Controllers for Pipelined Designs 143
7.6 Control Path Experiments and Results 151
7.6.1 Non-Pipelined Design Experiments and Results 151
7.6.2 Pipelined Design Experiments and Results 160

8 Thermal Analysis and Simulation Results 165
8.1 Thermal Models 167
8.2 Derivation of the Analytical Solution 169
8.3 Derivation of the Numerical Solution 172
8.4 Thermal Experiments and Results 173

9 Conclusions and Future Research 180
9.1 Data Path Synthesis 181
9.2 Control Path Synthesis 181
9.3 Thermal Analysis and Synthesis 182

Appendix A
Layouts of the 10-Time-Step Non-Pipelined FIR Filter Design 190

Appendix B
Layouts of the 4-Time-Step Non-Pipelined Differential Equation Solver Example 207

Appendix C
Layouts of the 12-Time-Step Non-Pipelined Elliptic Filter Design 211

List Of Figures

3.1 A Portion of the USC High-Level Synthesis System 13
3.2 Proposed System Architecture 14
3.3 Proposed Two-Phase Clocking Scheme 15
3.4 The VHDL Description for FIR Filter Design 19
3.5 The Translated Dataflow Graph for FIR Filter Design 20
3.6 The Schedule for the 10-Time-Step Non-Pipelined FIR Filter 21
3.7 The Floorplan for the 10-Time-Step Non-Pipelined FIR Filter . . . 22
3.8 An RTL Design for the 10-Time-Step Non-Pipelined FIR Filter . . 22
3.9 A Synthesis Example without Conditional Branches 24
3.10 A Synthesis Example with a Conditional Branch 25
3.11 Two Possible Controllers for the Example Shown in Figure 3.10 . . 26

4.1 Scheduling and Module Assignment Procedure 31
4.2 Iterative Improvement Procedure 33
4.3 The Scheduling Process of a 2-Time-Step Non-pipelined FIR Filter 35
4.4 The Schedule of the 2-Time-Step Non-pipelined FIR Filter Design . 36
4.5 The Floorplan of the 2-Time-Step FIR Filter with Minimum Operators 36
4.6 The Floorplan of the 2-Time-Step FIR Filter with a Redundant Adder 37
4.7 A Non-pipelined FIR Filter Using ChipCrafter Library Set 38
4.8 Floorplan with Minimum Operators Using ChipCrafter Library Set 39
4.9 Floorplan with a Redundant Adder Using ChipCrafter Library Set . 39

5.1 A 4-time-step FIR Filter ASAP and ALAP Schedule 42
5.2 Time Frames for the 4-time-step FIR Filter Example 42
5.3 Distribution Graph for 4-time-step FIR Filter Example 43
5.4 The Estimation of Wiring Area 51
5.5 A Finite State Machine Implementation Using PLA 53
5.6 Internal Construction of PLA (Taken from [Mli91]) 55
5.7 The Flow Chart of the Floorplanning Process 58
5.8 FIR Filter Floorplan with Overall Aspect Ratio Goal 2:1 58
5.9 FIR Filter Floorplan with Overall Aspect Ratio Goal 3:1 59
5.10 3-Room Floorplan Patterns, Orientations and Some Possible Labelings 59
5.11 The Construction of a Cluster Tree 61

5.12 Shape Function Addition for Horizontal and Vertical Cuts 64
5.13 Merging of Shape Functions 66
5.14 Calculation of the Shape Function for a Cluster Node with 3 Children 67
5.15 The Loop Model 71
5.16 The Proposed Approach for Loop Designs 72
5.17 The Flow Chart of 3D Scheduling Algorithm 74
5.18 The Allocation Table for the 4-Time-Step FIR Filter Example . . . 75

6.1 The 10-Time-Step FIR Filter Schedule Produced by LADS 83
6.2 The RTL 10-Time-Step FIR Filter Design using the LADS Schedule 84
6.3 The Floorplan created by LADS 85
6.4 The Final Implementation Floorplan using the LADS Schedule . . . 86
6.5 The 10-Time-Step FIR Filter Design Produced by FDS 88
6.6 The RTL 10-Time-Step FIR Filter Design using the FDS Schedule . 89
6.7 A Manually Created Timing-Driven Floorplan using the FDS Schedule 89
6.8 The 10-Time-Step FIR Filter Schedule Produced by MAHA 90
6.9 The RTL 10-Time-Step FIR Filter Design using the MAHA Schedule 91
6.10 A Manually Created Timing-Driven Floorplan using the MAHA Schedule 91
6.11 Comparisons with Different ChipCrafter Placements 95
6.12 Comparisons with Different Placement Strategies 96
6.13 The Pipelined FIR Filter Schedule by LADS (Init.I. = 4, P.Len. = 9) 98
6.14 The Floorplan for Pipelined FIR Filter Design Created by LADS . 99
6.15 The Pipelined FIR Filter Schedule by Sehwa (Init.I.=4, P.Len.=6) . 100
6.16 The 4-Time-Step Non-Pipelined Differential Equation Solver Schedule 100
6.17 The RTL Design for the 4-Time-Step Differential Equation Solver . 101
6.18 The Floorplan for the 4-Time-Step Differential Equation Solver Produced by LADS 102
6.19 The 12-Time-Step Non-Pipelined Elliptic Filter Schedule by LADS 104
6.20 The RTL Design of 12-Time-Step Non-Pipelined Elliptic Filter by MABAL 105
6.21 The Floorplan for 12-Time-Step Non-Pipelined Elliptic Filter Design by LADS 106
6.22 The Final Implementation Floorplan *LADS Placement I* for 12-Time-Step Non-Pipelined Elliptic Filter Design 107
6.23 The Final Implementation Floorplan *LADS Placement II* for 12-Time-Step Non-Pipelined Elliptic Filter Design 108
6.24 The 17-Time-Step Non-Pipelined Elliptic Filter Schedule by MAHA 112
6.25 The 10-Time-Step Non-Pipelined Elliptic Filter Schedule by LADS 113
6.26 The Floorplan for 10-Time-Step Elliptic Filter Design Created by LADS 114
6.27 The 14-Time-Step Non-Pipelined Elliptic Filter Schedule by MAHA 115

6.28 The Control/Data Flow Graph for the Robot Arm Controller 116
6.29 The VHDL Description of the Robot Arm Controller Example . . . 118
6.30 The Schedule and Data Flow Graph of the Robot Arm Controller . 119
6.31 The Timing Graph of the Robot Arm Controller Example 120

7.1 A Ring State Transition Diagram 126
7.2 A Simple Pipelined Design Synthesis Example 128
7.3 Control Specifications Using Status Registers 131
7.4 A Pipelined Design Synthesis Example with Conditional Branches . 136
7.5 The Controller for the Pipelined Example Shown in Figure 7.4 . . . 147
7.6 A PLA-Based Controller Synthesized by Finesse Program 153
7.7 PLA Area Comparison of the Non-Pipelined Designs 155
7.8 Controller Area Comparison of the Non-Pipelined Designs 156
7.9 *MAHA.12* State Transition Diagram without Status Registers . . . 158
7.10 *MAHA.12* State Transition Diagram Using Status Registers 159
7.11 PLA Area Comparison of the Pipelined Designs 164
7.12 Controller Area Comparison of the Pipelined Designs 164

8.1 A Typical Single-Chip Heat Transfer Model 167
8.2 A Four-Layer Chip Thermal Model 168
8.3 A Simplified Layered Chip Thermal Model 169
8.4 Finite Difference Approximation and The Corresponding Nodal Network 172
8.5 A 2-time-step Non-pipelined FIR Filter Design 174
8.6 The Floorplan for the FIR Filter Design with Minimum Operators . 174
8.7 The Floorplan for the FIR Filter Design with a Redundant Adder . 175
8.8 The Thermograms of the FIR Filter Design 177
8.9 The Floorplan for the Heat-Balanced FIR with a Redundant Adder 178
8.10 The Floorplan for the Heat-Balanced FIR with Redundant Adders . 178
8.11 The Thermogram of the Heat-Balanced FIR with Redundant Adders 179

A.1 The Layout of *ChipCrafter Placement I* using LADS Schedule . . . 192
A.2 The Layout of *ChipCrafter Placement II* using LADS Schedule . . . 193
A.3 The Layout of *ChipCrafter Placement III* using LADS Schedule . . 194
A.4 The Layout using LADS Schedule and Floorplan without Pin Assignment 195
A.5 The Layout using LADS Schedule and Floorplan with Pin Assignment 196
A.6 The Layout of *ChipCrafter Placement I* using FDS Schedule 197
A.7 The Layout of *ChipCrafter Placement II* using FDS Schedule 198
A.8 The Layout of *ChipCrafter Placement III* using FDS Schedule . . . 199
A.9 The Performance-Driven Layout using FDS Schedule without Pin Assignment 200

A.10 The Performance-Driven Layout using FDS Schedule with Pin Assignment . . . 201
A.11 The Layout of *ChipCrafter Placement I* using MAHA Schedule . . . 202
A.12 The Layout of *ChipCrafter Placement II* using MAHA Schedule . . 203
A.13 The Layout of *ChipCrafter Placement III* using MAHA Schedule . . 204
A.14 The Performance-Driven Layout using MAHA Schedule without Pin Assignment . . . 205
A.15 The Performance-Driven Layout using MAHA Schedule with Pin Assignment . . . 206

B.1 The Layout of *ChipCrafter I* using LADS Schedule . . . 208
B.2 The Layout of *ChipCrafter II* using LADS Schedule . . . 209
B.3 The Layout using LADS Schedule and Floorplan . . . 210

C.1 The Layout of *ChipCrafter Placement I* before Buffer Resizing . . . 213
C.2 The Layout of *ChipCrafter Placement II* before Buffer Resizing . . 214
C.3 The Layout of *ChipCrafter Placement III* before Buffer Resizing . . 215
C.4 The Layout of *ChipCrafter Placement IV* before Buffer Resizing . . 216
C.5 The Layout of *ChipCrafter Placement V* before Buffer Resizing . . 217
C.6 The Layout using *LADS Placement I* before Buffer Resizing 218
C.7 The Layout using *LADS Placement II* before Buffer Resizing 219
C.8 The Layout of *ChipCrafter Placement I* after Buffer Resizing 220
C.9 The Layout of *ChipCrafter Placement II* after Buffer Resizing . . . 221
C.10 The Layout of *ChipCrafter Placement III* after Buffer Resizing . . . 222
C.11 The Layout of *ChipCrafter Placement IV* after Buffer Resizing . . . 223
C.12 The Layout of *ChipCrafter Placement V* after Buffer Resizing . . . 224
C.13 The Layout using *LADS Placement I* after Buffer Resizing 225
C.14 The Layout using *LADS Placement II* after Buffer Resizing 226

List Of Tables

4.1 First Library Set Used for FIR Filter 34
4.2 Minimum-Operator Designs versus Redundant-Operator Designs . . 37
4.3 Library Set Created by ChipCrafter 38
4.4 A FIR Filter 2-Time-Step Design Using ChipCrafter Library Set . . 38

6.1 Design Library Set Used by LADS (Obtained from ChipCrafter) . . 82
6.2 The 10-Time-Step Non-Pipelined FIR Filter Designs using the LADS Schedule 92
6.3 The 10-Time-Step Non-Pipelined FIR Filter Designs using the FDS Schedule 93
6.4 10-Time-Step Non-Pipelined FIR Filter Designs using the MAHA Schedule 93
6.5 The 4-Time-Step Non-Pipelined Differential Equation Solver Example 102
6.6 The 12-Time-Step Elliptic Filter Designs before Buffer Resizing . . 109
6.7 The 12-Time-Step Elliptic Filter Designs after Buffer Resizing . . . 110
6.8 Non-Pipelined Robot Arm Controller Examples by LADS 117
6.9 Pipelined Robot Arm Controller Examples by LADS 121

7.1 Design Library Set Used by CSG (Obtained from ChipCrafter) . . . 151
7.2 Non-Pipelined Robot Arm Controller Examples by MAHA 152
7.3 Non-Pipelined Robot Arm Controller RTL Examples by MABAL . 152
7.4 Controllers Using Status Registers for Non-Pipelined Examples . . . 154
7.5 Controllers without Status Registers for Non-Pipelined Examples . 157
7.6 Pipelined Robot Arm Controller Examples by Sehwa 160
7.7 Pipelined Robot Arm Controller RTL Examples by MABAL 161
7.8 Controllers Using Status Registers for Pipelined Examples 162
7.9 Controllers without Status Registers for Pipelined Examples 163

8.1 Library Set Created by ChipCrafter Silicon Compiler 173

A.1 The Bindings for the 10-Time-Step Non-Pipelined FIR Filter Design Produced by LADS 191

B.1 The Bindings for the 4-Time-Step Non-Pipelined Differential Equation Solver Example Produced by LADS 208

C.1 The Bindings for the 12-Time-Step Non-Pipelined Elliptic Filter Design Produced by LADS 212

Abstract

In modern integrated circuits, submicron feature sizes result in delays of interconnections being comparable to delays of functional units. This thesis describes a new approach to high-level synthesis which simultaneously considers the area and delays of interconnections as well as functional units during the scheduling process using floorplanning. The floorplan concurrently constructed during the scheduling process provides an accurate estimate of the area and delays caused by interconnections to the scheduler. Our experiments shows that this new scheduling technique produces satisfactory results for various practical-size examples.

The control path synthesis program *CSG* described in this thesis automatically synthesizes controllers with conditional branches for pipelined as well as non-pipelined datapath dominated designs. *CSG* can introduce status registers into a controller design to store the condition state in a design with conditional branches. A multiple-code state encoding may be obtained using status registers, which in turn enables logic synthesis programs to produce a better minimization. The experimental results show that controllers using status registers result in smaller PLAs in many design examples.

A result of reduced feature sizes is the increase in power density. Design strategies are described to relieve potential thermal problems during high-level synthesis. Operators are placed as close as possible to their data predecessors in order to minimize the interconnection cost while ensuring that the thermal constraints are not violated. Spreading out overused functional units around the problem area is often at the cost of performance degradation. Introducing redundant operators is suggested to reduce the module utilization among problem modules when system performance is important. Our experimental results produce quite satisfactory results for a power-dominated example.

Chapter 1

Introduction

Due to the decreasing cost of VLSI chips, Application Specific Integrated Circuits (ASICs) have emerged as one of the fastest growing aspects of IC design. As the life cycle of IC products is reduced, ASIC designers have to make the design process quicker as well as simpler to further meet the needs of markets. To reduce the IC design turnaround time, the design automation industry has responded by introducing many design tools. These tools include layout generation, placement and routing, module generation, simulation and many others. The development of higher-level design tools is needed due to the constantly increasing market pressure. This raises research interest in behavioral-level synthesis (i.e. high-level synthesis).

1.1 Behavioral Synthesis

High-level synthesis takes an abstract behavioral specification of a digital system and finds a register-transfer level structure which realizes the given behavior. By *behavior* we mean the way that the system interacts with its environment. Usually there are many different structures that can be used to realize a given behavior. One of the synthesis tasks is to find the structure that best meets the constraints (limitations on the performance, area or power) while minimizing other costs.

The input specification algorithm gives the required mappings from sequences of inputs to sequences of outputs. From that input specification, the synthesis system produces a description of a register-transfer level structure. This structure includes a data path as well as a control path. A *data path* is a network of registers,

functional units, multiplexers and buses. A *control path*, on the other hand, is a finite state machine which drives the data paths so as to produce the required behavior. The control path can be realized in terms of microcode, a PLA circuit or random logic.

1.1.1 Data Path Synthesis

The core of data path synthesis can be divided into three subtasks, namely, scheduling, module allocation and binding. They are closely interrelated and depend on each other. Scheduling assigns the operations to time steps. A time step is a fundamental sequencing unit in synchronous systems. The aim of scheduling is to minimize the number of time steps needed to realize the specified behavior under certain limits on the available hardware resources. Scheduling is also used to minimize the amount of hardware resources to realize the specified behavior that is limited by a certain number of time steps.

Module allocation allocates a sufficient amount of hardware to implement the design. The hardware consists of functional units, memory elements and interconnection paths. The interconnection paths include multiplexers, tri-state drivers and buses. They are designed so that the functional units and registers are connected to support the data path transfers required by the schedule. Module binding consists of assigning the operations to operators (hardware resources) and values to registers. The goal of module allocation and binding is to minimize the amount of hardware needed to realize the design.

1.1.2 Control Path Synthesis

Once the schedule and data path has been chosen, it is necessary to synthesize a controller to drive the data paths as specified by that schedule. The controller is derived using the information from the schedule, the register-transfer network and bindings produced by data path synthesis. The activated control signals will select operations to be performed at specific time steps and route the data through appropriated functional units.

The synthesis of the controller itself can be done in different ways. For example, in a non-pipelined design without conditional branches, a control step could correspond to a state in the finite state machine. After the controlling finite state machine has been specified, the state machine can be further synthesized by logic synthesis tools, including state encoding and optimization of the combinational logic.

1.2 Motivation and Problem Approach

Submicron integrated circuits have extremely small, fast gates. The area and delay gaps between functional modules and interconnection modules are narrowing, partly due to smaller feature sizes, and partly due to larger designs being integrated into a chip, requiring proportionally more interconnections. Module binding can affect the performance of the subsequent physical implementation greatly due to long interconnection delays between operators. An "optimal" register-transfer level schedule can actually be quite suboptimal when interconnection delays dominate execution delays. However, interconnection delays can only be determined accurately after floorplanning is completed. Obviously, high-level synthesis tools using submicron technology will not be able to make intelligent scheduling decisions without considering the effects of interconnections using floorplanning.

The thermal problem is another issue in designing a high-speed integrated circuit. A result of reduced feature sizes is an increase in power density, partly due to the higher operating speed, and partly due to the increased component density on an integrated circuit. On the other hand, the operating temperature of a chip is limited to a certain range for acceptable system reliability. For silicon devices, this temperature is in the range of 75–85°C [Jr.83]. With increasing power density and with limits on the operating temperature, thermal limitations of the chip must be considered during the design process. Consequently, it is imperative to study not only the thermal properties of the device and package material but also the run-time thermal properties on the chip/die so that a better thermal layout can be obtained to alleviate potentially high thermal stress during the design stage.

A new approach for scheduling called 3D scheduling,[1] which performs scheduling, module allocation, module binding and floorplanning concurrently to take into account interconnection area and delay effects during the scheduling process, is proposed. Interconnections here include *wiring*, *multiplexers* and *registers* in the data path. Prediction tools are used in the 3D scheduling technique to preview register-transfer level design characteristics before the scheduling process is performed. The 3D scheduling approach can produce schedules with operation chaining.[2] The delay of an operation chain is determined according to characteristics of chained modules, which are described in the module library. For example, the delay of two cascaded carry-select adders is roughly equal to two times of carry-select adder delay. However, the delay of two cascaded ripple-carry adders is slightly longer than the delay of a ripple-carry adder. The 3D scheduling technique estimates the delays of operation chains accurately which allows the 3D scheduler more degree of freedom during the scheduling process.

To resolve potential thermal problems in a design, a new scheduling approach which simultaneously balances heat distribution by means of floorplanning is proposed. Operations are scheduled and placed as close as possible to their data sources to minimize interconnection costs while assuming thermal constraints are not violated. Two strategies are proposed for designs having heat concentration problems. First, "hot" circuits can be spread around the problem area over the unused space on the floorplan. Or, thermal problems may be alleviated by introducing extra redundant operators to reduce the utilization of operators around the problem area.

After the data path is produced, control path synthesis is performed to create the respective controller. The ability to synthesize controllers for pipelined designs with conditional branches contributes to the problem complexity. Two controller implementation schemes, namely, controllers with status registers and controllers without status registers, are proposed to minimize the area of a design. The purpose of introducing status registers into a controller is to store reserved condition

[1] 3D scheduling refers to the problem of simultaneously scheduling time and the X-Y plane.

[2] An operation chain allows two or more operations with data dependencies to be scheduled in the same time step.

values in a design with conditional branches, which can sometimes produce smaller layouts than traditional approaches.

1.3 Thesis Organization

The work presented in this thesis falls into three groups: data path synthesis, control path synthesis and thermal analysis and synthesis. Chapter 2 describes the related research on data path synthesis, prediction methods, floorplanning approaches, controller design and thermal analysis. In Chapter 3, overviews of the data path synthesis and control path synthesis are presented.

Chapter 4 describes the preliminary data path synthesis research. Chapter 5 then presents our current data path synthesis approach in which the 3D scheduling technique is given. The experiments on data path synthesis are presented in Chapter 6. An extensive set of examples were synthesized. Layouts and timing simulation results were used to validate our system. Chapter 7 describes the work on control path synthesis. Two controller implementation schemes are proposed: controllers with and without status registers. The second part of Chapter 7 describes the experiments performed on control path synthesis. Both types of controllers (i.e. with and without status registers) were synthesized and laid out to illustrate the tradeoff between area and performance.

Chapter 8 presents thermal models, an analytical solution and a numerical solution to perform thermal analysis on the surface of a chip/die. A group of thermograms corresponding to different module allocations, bindings and floorplans were produced to simulate temperature profiles of different designs but the same behavior. Chapter 9 concludes the research done in this thesis and discusses future research problems.

Chapter 2

Related Research

In this chapter, we study previous research efforts in the literature. The survey is divided into the following five areas: (i) data path synthesis, (ii) prediction methods. (iii) floorplanning approaches, (iv) control path synthesis and (v) thermal analysis and synthesis.

2.1 Data Path Synthesis

It is very difficult to predict the performance of submicron silicon before the layout is completed. Very little synthesis research has taken into account the physical design effects in high-level synthesis. BUD is an unique program which floorplans prior to synthesis [McF86]. BUD first builds a hierarchical cluster tree based on the common functionality, common interconnections and potential parallelism. Resource allocation and operation assignment are performed by cutting this tree at different levels. A schedule is produced for each configuration (due to each cut) and final design characteristics are estimated using an approximate floorplan.

Fasolt floorplans and analyzes area impact after scheduling [Kna90]. The current schedule is adjusted in the next design iteration to reduce the area overhead caused by interconnections. ELF is an early system which estimates interconnection effects during synthesis [Gir84]. ELF predicts the wiring area from register-transfer level characteristics to make design decisions. Chippe is a constraint driven expert system which predicts wire delays from the structural RT-level

design [BG90]. The worst case wiring delay is predicted after behavioral-level synthesis and this delay is used by the evaluation mechanism of the iteration synthesis process.

HAL uses a force function proposed by Paulin [PK89] to perform the scheduling subtask. The allocation approach in HAL is rule-based. The work of Cloutier and Thomas at Carnegie Mellon University modified the force function to perform scheduling, module allocation and module assignment at the same time due to the close interrelation of scheduling and module binding subtasks [CT90].

Splicer [Pan88] is an example system which uses extensive search techniques to perform synthesis. The technique, based on branch-and-bound search, is used in Splicer to bind the best operation assignment while the interconnections are generated simultaneously. LYRA and ARYL [HCLH90] formulated the operation assignment task as a bipartite graph matching problem. Interconnection allocation is performed by a greedy heuristic after register allocation and operation assignment are done. LYRA performs register allocation first followed by operation assignment. ARYL performs operation assignment first followed by register allocation.

2.1.1 Prediction Methods

The early work on the estimation of functional resource allocation was on the prediction of lower-bound functional area due to varying time characteristics for pipelined and non-pipelined design styles by Jain and Parker [JPP87] [JMP88]. This prediction research was focused on estimating the resource allocation behavior of Sehwa [PP88] and MAHA [PPM86], the pipelined and non-pipelined scheduling programs in the ADAM synthesis system. Only single-cycle operations were considered in this work.

An estimation technique for functional unit allocation was recently proposed by Hagerman [Hag91]. The concept of distribution graphs [PK89] is used to estimate the functional unit allocation to be produced by an imperfect scheduling tool (e.g. a force-directed type scheduling tool). Estimation results are favorable as compared to the scheduling results produced by SAM [CT90].

In the research of the area estimation, PLEST is a program which estimates the layout area of an register-transfer level design using the standard-cell placement and routing style. PLEST estimates areas of layouts with various aspect ratios, feed-throughs and track densities using probabilistic methods. PLEST results were verified by RCA CADDAS standard-cell layouts and were within 10% of actual layout areas [KP89].

Zimmerman proposed a constructive layout estimation based on predicting shape functions for a binary slicing tree [Zim88]. Shape functions of leaf nodes are used to predict shape functions of composite nodes. The process is hierarchically repeated until the shape function of the root node is calculated. Kurdahi and Ramachandran combined Zimmerman's constructive methods with probabilistic methods used in PLEST [KR91]. Since this estimation approach does not traverse the whole slice tree, a favorable speed advantage over purely constructive methods (e.g. Zimmerman's approach) is expected.

2.1.2 Floorplanning Approaches

Most current approaches consider the scheduling and floorplanning problems separately which optimizes designs with different objective functions. Lauther's version of min-cut placement [Lau79] is an early work on floorplanning research. A top-down method that divides cells into two blocks was used. The process continuously divides blocks into smaller blocks until the number of cells in a block is small. Placement decisions made at higher levels of the hierarchy may suffer from lack of lower-level information.

Otten's shape propagation floorplanning technique [Ott83] initially derives a physical hierarchy in the form of a binary slicing tree. The slicing tree is then traversed bottom-up. At each internal node of the tree, a composite shape function is calculated from shape functions of its children and directions of cuts. These shape functions are combined and propagated recursively to the root of the tree. However, in Otten's approach, a binary slicing tree is restrictive and global connection costs are not considered during the bottom-up traversal.

La Potin [PD86] improved the min-cut method by considering shape functions and pad positions during floorplanning. Dai [DK87] extended La Potin's work to

general non-slicing structures and multi-way cluster trees and combined floorplanning with hierarchical global routing. Pedram [Ped91] further extended Dai's work by computing shape functions for cluster nodes bottom-up which incorporates cell sizes, shapes and pin positions to achieve physical and timing constraints.

Zimmerman [Zim88] improved Otten's technique to optimize directions of cuts during the shape function propagation. The wiring area is estimated for each node of the binary slicing tree which shifts the shape functions of nodes to account for wiring areas. Herrigel [HGF89] and Lengauer [Len90] proposed a top-down follow-up phase to minimize the total interconnection length by switching cells across cuts. However, the improvement of this method is limited, because the minimizations of layout area and interconnection length are performed separately.

2.2 Control Path Synthesis

In the area of automated controller synthesis, most of existing high-level synthesis systems only synthesize controllers for *non-pipelined* designs. A widely applied controller style in many existing synthesis system is the ROM-based microprogrammed controller model. Examples include the control allocator for the CMU-DA [NCP82] and ATOMICS system for Cathedral-II [GVM87] [Gaj88a] [ZSRM90]. The control allocator for the CMU-DA system assumes a canonical microprogrammed model, and performs optimizations based on microcode format constraints. The ATOMICS system takes an RT-level description as input and performs microprogram scheduling in order to minimize the global machine cycle count. Three pipeline stages can be identified in the control critical path of designs produced by ATOMICS.

Some systems map the control/data flow graph representing the hardware behavior into a corresponding state transition diagram, which can be implemented by either a PLA circuit or random logic. For example, the Yorktown Silicon Compiler [Gaj88b] implements the controller as a hierarchy of finite state machines, where a finite state machine is associated with each routine. Control state splitting allows the delay through the combinational part of the data path to be reduced to satisfy clock cycle constraints. AT&T's Bridge system [TPL+86] [TWR+88] first creates a

symbolic control table once the data path is allocated. The symbolic control table basically identifies all modules which need to be activated in each cycle. The detailed implementation of the controller will be decided by the module generator. In the Hyper synthesis system [CPTR89], a state transition diagram is derived from a control/data flow graph based on the scheduling information. It is a recursive procedure due to the hierarchy nature of the control/data flow graph. The state transition diagram is then optimized by removing dummy states. The PUBSS system [WTL91] developed at Princeton University synthesizes control-dominated machines by performing state minimization or simplification, state decomposition and state collapsing on the behavior FSMs to achieve user specified constraints. A behavior FSM is an automaton whose inputs and outputs are only partially scheduled (i.e. the timing behavior is incompletely specified).

Kim's controller synthesis research at UC-Irvine [KK91] is an example which synthesizes controllers for pipelined designs using PLAs as its platform. The controller is modeled as a Moore-style finite state machine.[1] Control states are decided by possible execution modes in a scheduled control/data flow graph using a coloring scheme proposed by Park [PP88]. Next state transitions for each control state are determined by the compatibility of two possible execution modes between two consequent group of states. A condition value needs to be produced at least two clock cycles before the conditional execution is performed, because of the use of a Moore-style controller.

Although systems mentioned above are part of powerful synthesis systems and effective in synthesizing some classes of designs, they address only part of all of our objectives—to synthesize controllers for pipelined designs as well as non-pipelined designs with/without conditional branches.

2.3 Thermal Analysis and Synthesis

To the best of our knowledge, no high-level synthesis research which considers thermal effects has been reported. Instead of taking care of the thermal problem

[1]Outputs of a Mealy-style finite state machine depend on inputs as well as the current state. Outputs of a Moore-style finite state machine only depend on the current state.

during the design process, most of the reported works focus on a post-design analysis procedure. Many improvement procedures at the board-level have suggested that thermal design should be considered during the PCB layout design process [OP90]. However, proposed thermal models and improvement procedures used at the board-level cannot be applied directly for designs at the chip-level.

Thermal analysis using a numerical approach has been done many times. Numerical techniques such as finite-difference [MAD83], finite-element [WFM83] and boundary element [CPB88] has been widely used to analyze thermal profiles of electronic circuits with irregular boundary conditions. Some analytical research which focuses thermal analysis on ASIC chips has been published recently. Basically the analyzed structure was simplified to a layer model. The number of layers vary from two [CA78] to four layers [LPM89] [LMP89]. The computation complexity rapidly increases as the number of layers increases. These works analyzed and pointed out potential thermal problems but did not provide any improvement strategies during the circuit design process. However, considering thermal design after the circuit design is completed is usually too late to alleviate potential thermal problems. Especially on some critical circuit designs, it is almost impossible to meet thermal constraints after the circuit design is completely done. In such cases, iterations would have to be performed. The remaining chapters of this dissertation describe our research. Any results described here which are used for our research will be cited at the appropriate time.

Chapter 3

Overview of the Synthesis System Research

The current USC high-level synthesis system synthesizes register-transfer level designs which implement the behavior from algorithmic-level specifications. A portion of the USC high-level synthesis system is shown in Figure 3.1. A register-transfer level design can be subdivided by its functionality into two parts—the *data path* and the *control path*, as shown in Figure 3.2. The *data path* consists of a network of functional units, registers, multiplexers, tri-state-drivers and buses. The data path makes the complex computation in the algorithmic-level specification achievable. The *control path* generates the control signals[1] required by the registers, multiplexers and tri-state-drivers[2] for sequencing the events in the data path, as well as the signals required by the control path itself. In the rest of this chapter, overviews of data path synthesis and control path synthesis are given by following the proposed clocking scheme.

3.1 Limitations

Hierarchical control/data flow graphs are not currently considered; control/data flow graphs should be flattened before high-level synthesis. Pipelined designs as

[1]ALU control signals are not considered in the current CSG (Control Signal Generator). However, they can be obtained similarly to control signals for multiplexers by simple extension of the current CSG.

[2]In our approach, buses are logical interconnection elements whose usages are controlled by tri-state-drivers.

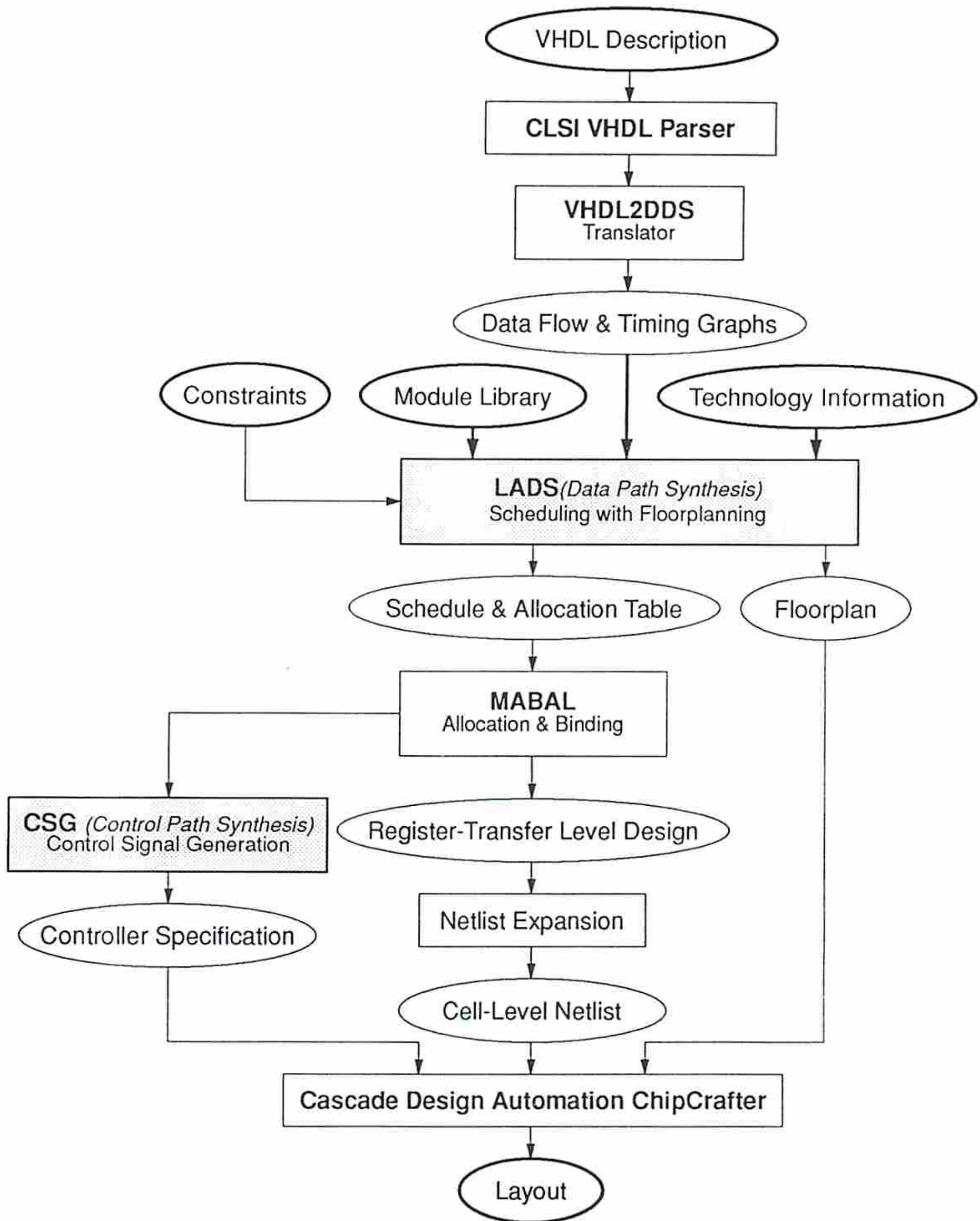


Figure 3.1: A Portion of the USC High-Level Synthesis System

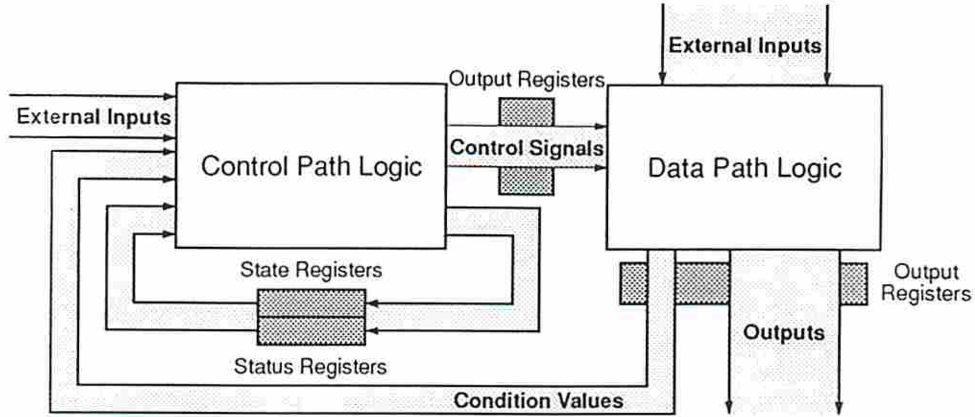


Figure 3.2: Proposed System Architecture

well as non-pipelined designs are synthesized in our data path and control path synthesis programs. The control/data flow graph may contain conditional branches; the mutual-exclusion operations of the same type may share an operator to reduce the hardware cost. Pipelined operators and multi-cycle operators are not considered in the current synthesis programs.

In data path synthesis, control/data flow graphs with nested inner loops are handled but not parallel loops for non-pipelined designs. The controller model is proposed for non-pipelined designs with nested inner loops but not implemented in the current control path synthesis program. Data recursion edges[3] are not considered for the pipelined designs. The possibilities of operation chaining are examined in data path synthesis to shorten the schedule.

The data path synthesis program performs operator allocation and bindings but register allocation, value bindings and multiplexer allocation are done partially. Buses are not considered but can be handled similarly as multiplexers by simple extension of the current data path synthesis program. A set of floorplans with different aspect ratios is created for a tentative design (or schedule). The control path synthesis program considers designs using multiplexer architectures as well as bus architectures. A finite state machine is produced using the Cascade Design Automation ChipCrafter silicon compiler. Two controller implementation styles

[3]A data recursion edge with a degree d is a pseudo edge representing data dependency in the control/data flow graph for a pipelined design, where d means the value used by an operation is generated d iterations earlier.

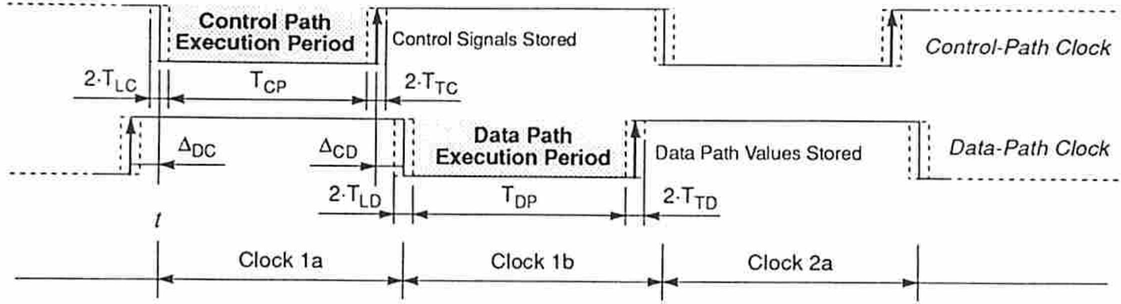


Figure 3.3: Proposed Two-Phase Clocking Scheme

are provided, namely, controllers using status registers and controllers without status registers.

3.2 Clocking Scheme

The clocking scheme chosen is vital because different clocking scheme assumptions may lead to completely different data path and control path synthesis results. In our approach, a conservative two-phase non-overlapping clocking scheme is assumed. A data-path clock cycle, $Clock_{DP}$, is always followed by a control-path clock cycle, $Clock_{CP}$, as shown in Figure 3.3.

Memory elements in the synthesized circuits can be classified into four categories, namely the state registers, output registers, data-path registers and status registers. State registers are used to keep track of the current state, which allows the controller to decide the next state transitions. Output registers hold the output control signals to avoid race conditions between the data path and control path during the data path execution. Data-path registers store the intermediate computation values created by functional elements in the data path at the end of each data path execution. If the status-register controller implementation style is assumed, status registers are used by the control path to store the condition values. Note that all the memory elements can be realized by either positive/negative edge-triggered D-type flip flops or level-sensitive latches depending on the clocking scheme used.

Positive edge-triggered D-type flip flops are proposed to implement all the memory elements, including the registers in the data path and control path. All

values in the controller's registers can only be changed at the rising edges of control-path clocks, and all the values generated by the data path can be stored into registers only at the rising edges of data-path clocks. If each clock lasts for a sufficiently long period then the correct executions of both the control path and data path can be ensured. The possibilities of race conditions can also be avoided because the registers in the data path and control path would never be changed at the same time.

Some significant parameters for an edge-triggered D-type flip flop (ETDFF) are listed below, with brief definitions [UT86]. The *setup time*, T_{setup}, for an ETDFF is the minimum time that the D (input) signal must be stable prior to the triggering edge of C (clock) pulse. The *hold time*, T_{hold}, is the minimum time that the D signal must be held constant after the triggering edge of the C pulse. The *propagation delay*, T_{delay}, is the time required for a signal to propagate from the C terminal to the Q terminal, assuming that the D signal has been set up sufficiently far in advance as specified by the setup time constraint. T_{LC}, T_{TC}, T_{LD} and T_{TD} specify the tolerances of the leading and trailing edges of control path clock and data path clock, respectively. For the clocking scheme we specified, there is a certain tolerance range, within which we can assume the errors will be confined. This is essentially a worst case approach. Δ_{DC} and Δ_{CD} define the phase difference from $Clock_{DP} \uparrow$ to $Clock_{CP} \downarrow$ and $Clock_{CP} \uparrow$ to $Clock_{DP} \downarrow$, respectively.

Assume that, for example, the leading edge of the $Clock_{CP}$ pulse of some cycles would arrive at every ETDFF at time t (illustrated in Figure 3.3). Then, in actual systems, this edge is received in the interval $(t - T_{LC}, t + T_{LC})$. Corresponding assumptions apply for the other three edges. Our goal is to design our system so that if this assumption and the corresponding assumptions about the precision of the components used are valid, then there will be no failure due to timing, even in the extreme case. A set of constraints are developed such that if all the flip flop input signals arrive on time for the first cycle, then they will also arrive on time for the next cycle. By induction, for all succeeding cycles, the flip flop inputs are also stable over appropriate intervals, so that the system will behave according to the specification. The timing constraints which ensure correct behavior are summarized as follows:

- The worst case control path execution period, T_{CP}, should be longer than the sum of the propagation delay T_{delay} of data path's DFFs, the control path combinational logic delay and the setup time T_{setup} of control path's DFFs.

- The worst case of data path execution period, T_{DP}, should be longer than the sum of the propagation delay T_{delay} of control path's DFFs, the data path combinational logic delay and the setup time T_{setup} of data path's DFFs.

- The earliest arrival time of control/data path input signals for the next cycle should not arrive so early as to violate the hold time constraints for the current cycle. i.e. $T_{CP} > T_{hold}$ the hold time of control path's DFFs; and $T_{DP} > T_{hold}$ the hold time of the data path's DFFs.

- The phase difference, Δ_{DC} and Δ_{CD}, should be non-negative to ensure a non-overlapping clocking scheme.

The clocking scheme we defined above implicitly requires condition values to be created at least one clock cycle before they can be used to determine which conditional execution paths are executed. For example, if a condition value is created by the data path during *Clock 1b* period then the value will be stored into a register in the data path at the end of *Clock 1b*. The condition value will be ready to be taken by the controller to decide the proper output signals, according to the condition value, during *Clock 2a* period. It is obvious *Clock 2a* is also the earliest clock period that the condition value can be used by the controller.

3.3 An Overview of Data Path Synthesis

The data path synthesis software in the high-level synthesis system described in this thesis takes VHDL descriptions of algorithmic/behavioral specifications as inputs [VHD88]. The VHDL descriptions characterize the data dependencies but not parallelism of the input algorithm. For convenience of analysis, the VHDL description is translated into a data flow graph and a timing graph in our approach. The DDS [Kna86], the internal representation in USC/ADAM synthesis subsystem, maintains separate representations of data flow and control timing information. A

VHDL description of an FIR filter is shown in Figure 3.4. Figure 3.5 shows the translated data flow graph of the FIR filter design.

After the preprocessing of the VHDL description is done, the data path synthesis program LADS (Layout And Data path Synthesis) takes user specified constraints, a module library, technology dependent information[4] and the translated data flow graph and timing graph to perform scheduling, allocation and binding synthesis subtasks using the concurrently generated floorplan. The schedule and one of the possible floorplans for the non-pipelined 10-time-step FIR filter design are shown in Figures 3.6 and 3.7, respectively.

LADS uses a 3D scheduling technique to perform scheduling simultaneously with floorplanning. The 3D scheduler first applies prediction tools to estimate the lower bound on the number of operators and the interconnection cost from the data flow graph. The prediction results are employed to decide module allocation and bindings of scheduled operations with a more global view before the whole scheduling process is completed. Module allocation and binding must be done along with scheduling in order to be able to floorplan with the bound modules. The 3D scheduler uses a modified force function to determine the scheduling and binding for each operation at a time. The area and delay effects of wires, multiplexers and registers are considered by the floorplanner during scheduling in our approach.

A constructive cluster-tree technique is used in the floorplanning process. The floorplan generated by the 3D scheduler is used to feedback the low-level physical information, such as the impacts of wiring area and delays, to the scheduler. In this way, the effects of interconnection in a register-transfer level design can be estimated more precisely during scheduling. Therefore, the 3D technique is able to produce more timing-balanced schedules as compared to other traditional approaches that only consider the delays of functional modules.

After LADS finishes scheduling and binding, the allocation and binding program MABAL takes the schedule and bindings produced by LADS to complete the register-transfer level design. One possible register-transfer level design for the

[4]Technology dependent information includes transistor parameters, layout parameters and other parameters.

```
entity FIR_FILTER is
   port(f1, f2, f3, f4, f5, f6, f7, f8, f9, f10, f11, f12,
   f13, f14, f15, f16, f17, f18, f19, 20, f21, f22, f23, f24 : in Integer;
   outf : out Integer);
end FIR_FILTER;

architecture Behaviour of FIR_FILTER is
begin
   process
     variable e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12,
     e13, e14, e15, e16, e17, e18, e19, e20, e21, e22 : Integer;
   begin
     e1 := f1 + f2;
     e2 := f3 + f4;
     e3 := f5 + f6;
     e4 := f7 + f8;
     e5 := f9 + f10;
     e6 := f11 + f12;
     e7 := f13 + f14;
     e8 := f15 + f16;
     e9 := e1 * f17;
     e10 := e2 * f18;
     e11 := e3 * f19;
     e12 := e4 * f20;
     e13 := e5 * f21;
     e14 := e6 * f22;
     e15 := e7 * f23;
     e16 := e8 * f24;
     e17 := e9 + e10;
     e18 := e17 + e11;
     e19 := e18 + e12;
     e20 := e19 + e13;
     e21 := e20 + e14;
     e22 := e21 + e15;
     outf <= e22 + e16;
   end process;
end Behaviour;
```

Figure 3.4: The VHDL Description for FIR Filter Design

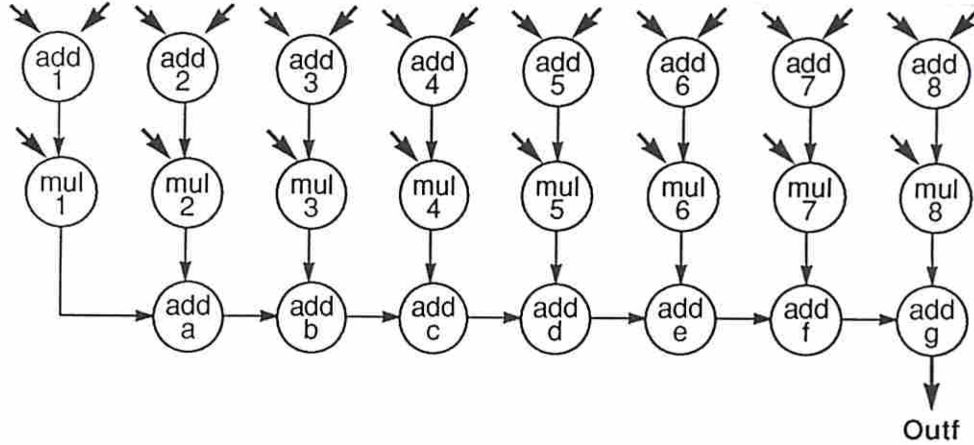


Figure 3.5: The Translated Dataflow Graph for FIR Filter Design

non-pipelined 10-time-step FIR filter design is shown in Figure 3.8. The register-transfer level designs can be further synthesized by logic-level and circuit-level CAD tools, such as the Cascade Design Automation ChipCrafter compiler, to obtain chip layouts.

3.4 An Overview of Control Path Synthesis

In synchronous systems (the only kind we consider in this approach), a controller can be described by a finite state machine that can be implemented by PLA circuits, random logic or microprograms. Data-path scheduling, module allocation and module binding are assumed to have been performed before control path synthesis. As shown in Figure 3.1, the control path synthesis program CSG takes MABAL's output to synthesize a controller in the current USC high-level synthesis system.

To retain flexibility in choosing the controller implementation style, only the behavior of the finite state machine is produced. Flexibility is important because we wish to synthesize digital circuits that have proper functionality and also meet the performance and area constraints. To accomplish this, the controller implementation style may vary from problem to problem. In this thesis, the controller is assumed to be implemented with a PLA circuit. However, if a microprogrammed

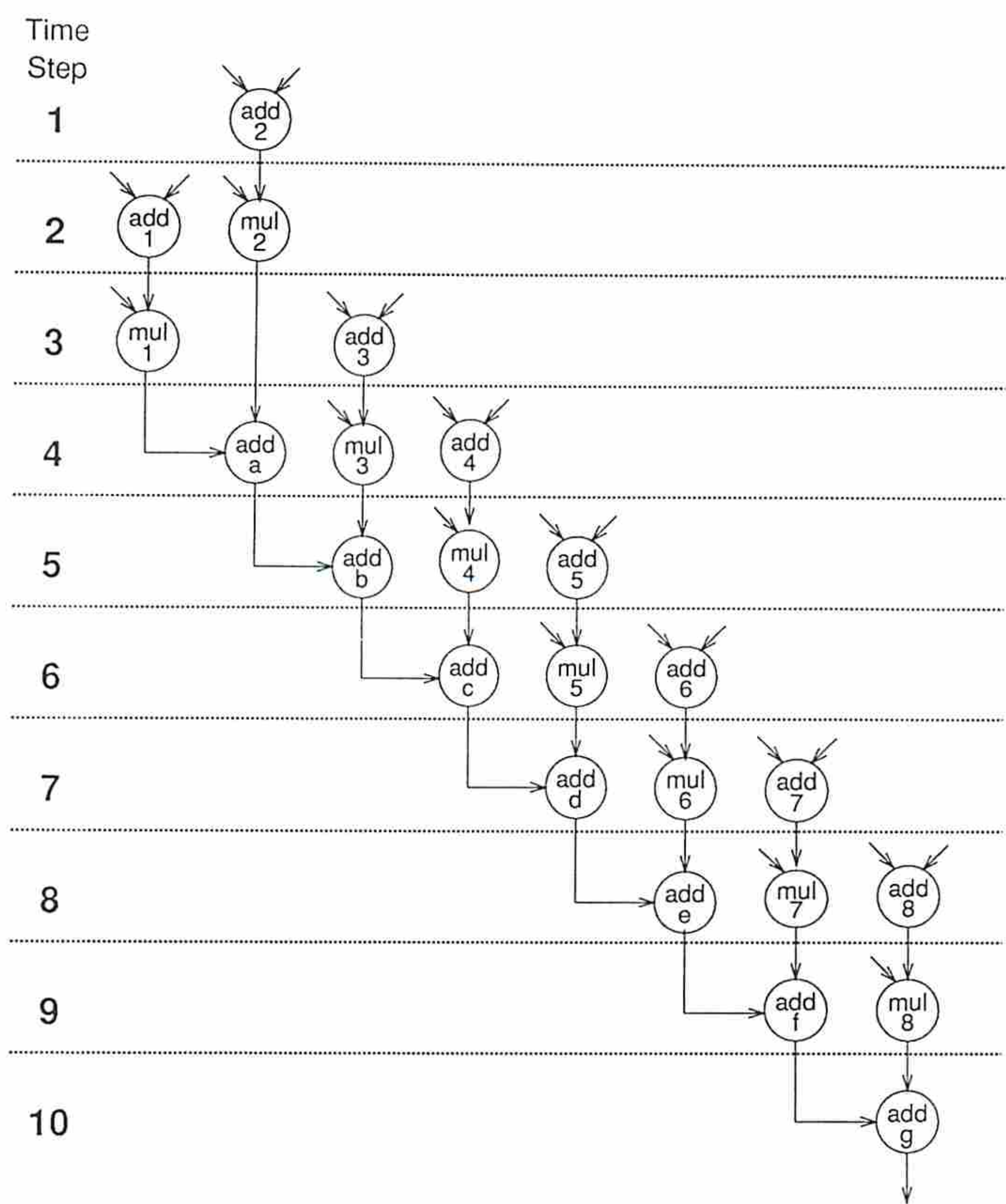


Figure 3.6: The Schedule for the 10-Time-Step Non-Pipelined FIR Filter

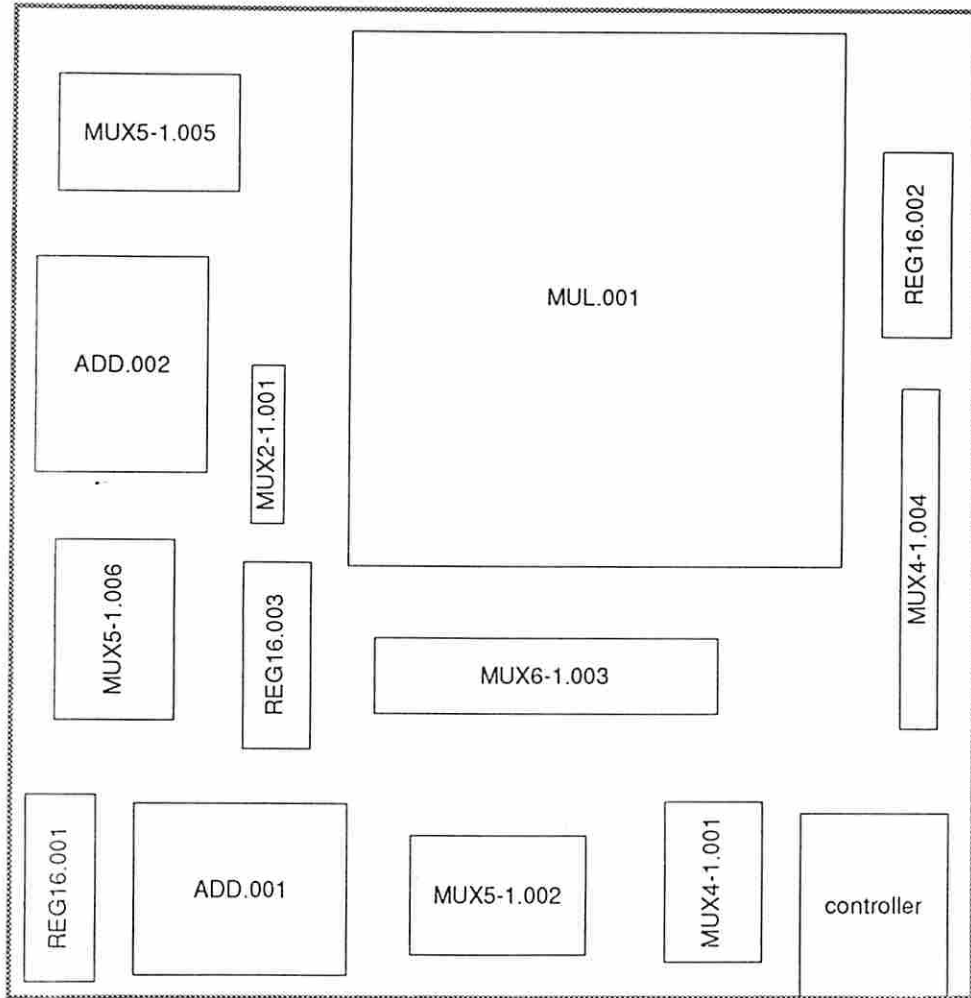


Figure 3.7: The Floorplan for the 10-Time-Step Non-Pipelined FIR Filter

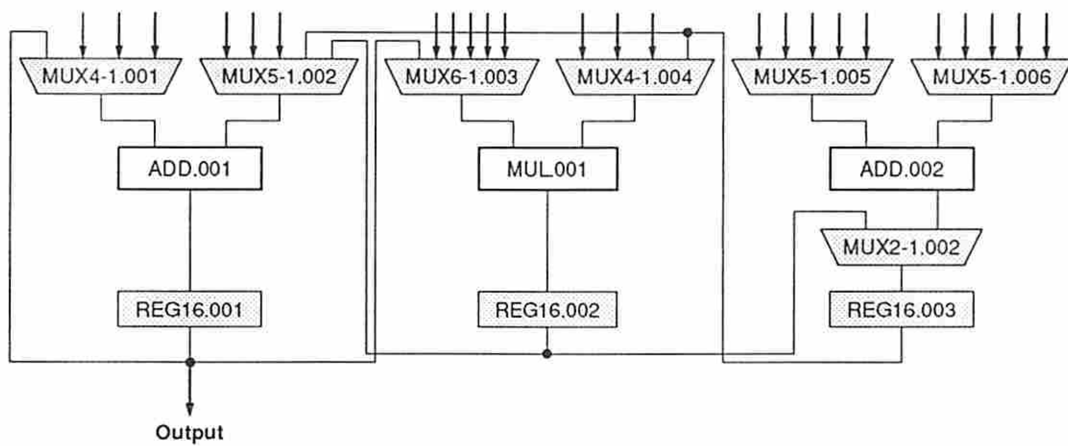


Figure 3.8: An RTL Design for the 10-Time-Step Non-Pipelined FIR Filter

controller were preferred, the finite state machine could also be implemented with a microprogrammed controller.

3.4.1 Controller for Designs without Conditional Branches

For a design without conditional branches, the controller specification is simple, and its state transition diagram is a simple ring shape. Figure 3.9 illustrates a non-pipelined design example without conditional branches. The dashed lines in the control/data flow graph present the proposed schedule, which is generated by the scheduling program in the data path synthesis system. An outer loop is assumed implicitly in this graph but would be found explicitly in the corresponding control timing graph not shown here. As described in the Limitations section, a more complex timing scheme is not supported here since the problem was simplified in order to make progress on the combined issues of scheduling and floorplanning. In this example, there are three time steps. A possible non-pipelined register-transfer level data path and controller are shown in Figures 3.9b and 3.9c, respectively.

The default action of data-path registers is to keep values stored in the previous time step. The contents of data-path registers are modified only when the control signals *"Register_WRITE"* are activated by the controller during the clock period. The state transition diagram of a controller implicitly has a feedback arc from the last time step to the first time step that allows the data path to process the next set of inputs. It has been found from this example that every state in the state transition diagram can be matched to a time step in the scheduled control/data flow graph. All the control signals specified in a state are necessary for the data path to complete the scheduled operations correctly.

3.4.2 Controller for Designs with Conditional Branches

Distribute-join-pairs in the control/data flow graph are used to represent *OrFork* and *OrJoin* operations on the conditional execution paths. The exact execution paths are determined by condition values that may come from external inputs or be created by the data path during execution. In our approach, we assume condition

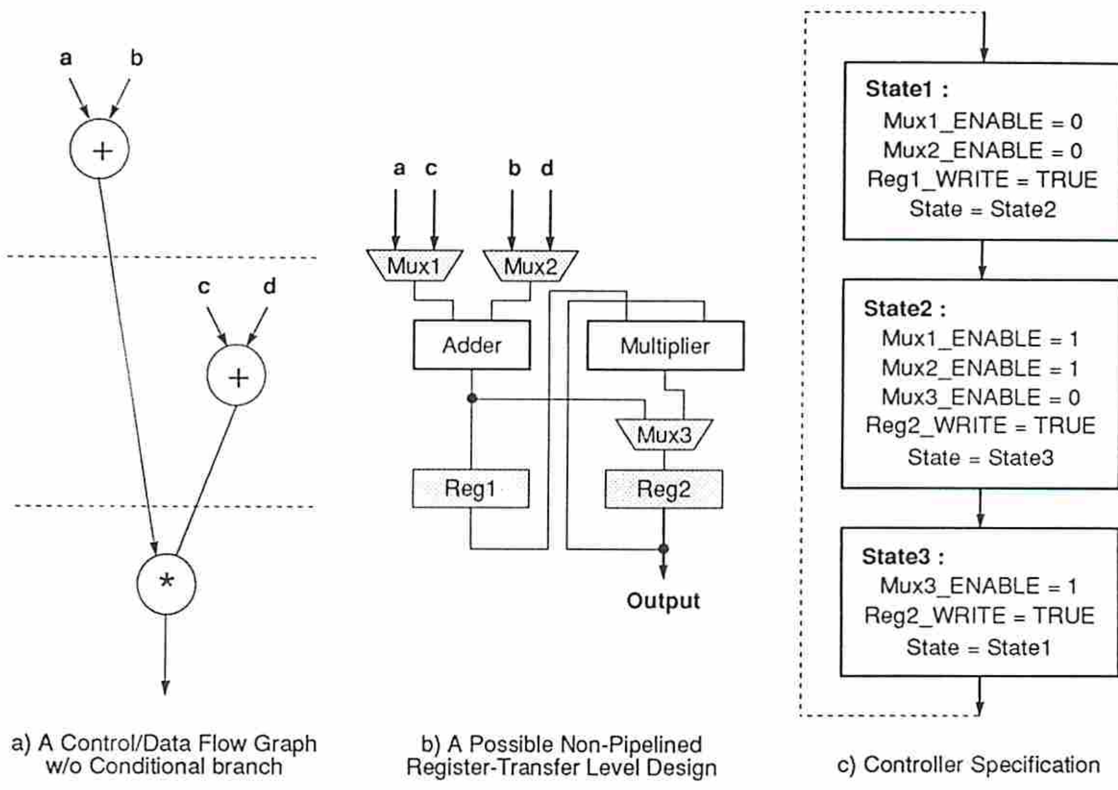


Figure 3.9: A Synthesis Example without Conditional Branches

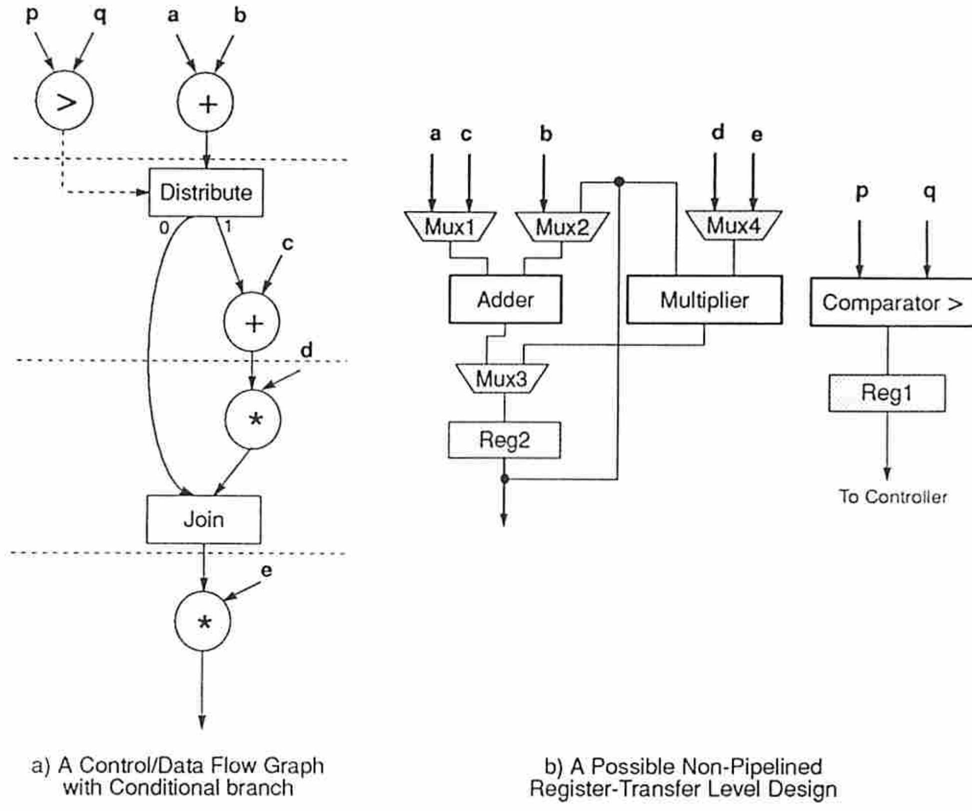


Figure 3.10: A Synthesis Example with a Conditional Branch

values generated by the data path will be stored in the data-path registers at least one clock cycle. Since inputs of the controller we synthesize are assumed either coming from the external inputs or data-path registers, data stored in the data-path register at least one clock cycle can avoid race conditions. This assumption can be clarified from the proposed clocking scheme. Condition values should be ready in the data-path registers for controller use no later than the time step that the conditional execution path begins. A condition value is then maintained by the controller until no more operation executions are dependent on it.

Figure 3.10 shows a non-pipelined design example with a conditional branch. There are four time steps and one distribute-join pair in this scheduled control/data flow graph. The conditional execution paths are found in time steps 2 and 3. The condition value "$p > q$" determines the output is "$(a+b)*e$" or "$(((a+b)+c)*d)*e$." In this example, the condition value "$p > q$" should be created no later than time step 1, since the conditional operation (the addition of $a + b$ to c) is scheduled to

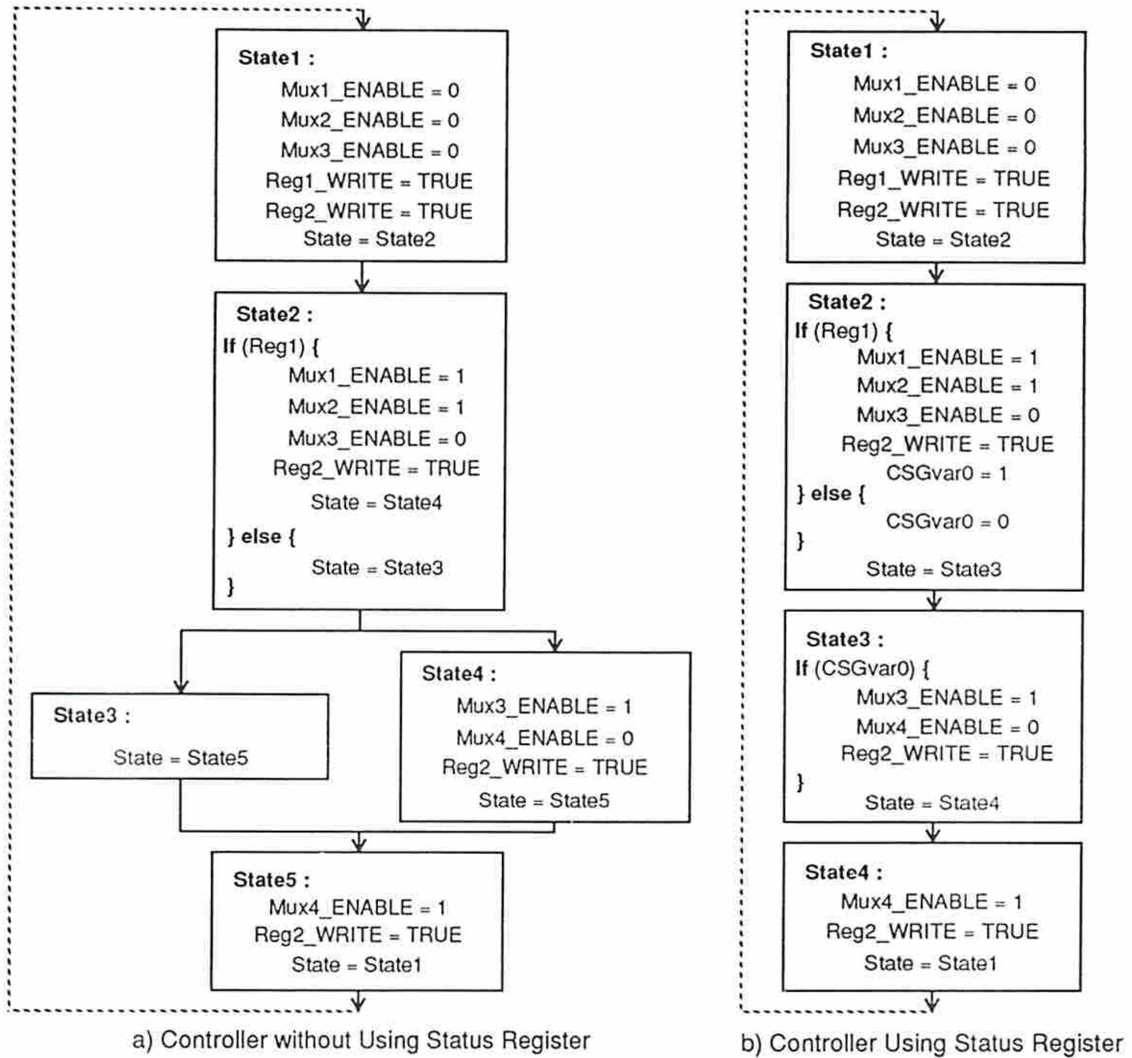


Figure 3.11: Two Possible Controllers for the Example Shown in Figure 3.10

be executed in time step 2 and the controller needs to decide the execution path to be processed at the beginning of time step 2. Also, the condition value generated in time step 1 should be stored into one of the data path registers (*Reg1* in this example).

In order to "memorize" the condition values, two controller implementations are proposed. The first approach uses status registers to store the condition values. An alternative approach uses different states to distinguish different condition value combinations.

Figure 3.11a shows a possible controller implementation for the synthesis example in Figure 3.10. In *state2*, *Reg1* stores condition value "$p > q$," and is used to determine the next state transition (*state3* or *state4*). Using this implementation, the condition value is stored in the *state memory*. For example, *state3* and *state4* represent the condition value combinations "$p > q$" *FALSE* and *TRUE*, respectively. Also, both *state3* and *state4* have *state5* as their next state transition. Since no conditional execution path exists after time step 3, the condition value "$p > q$" dies at the end of time step 3.

A possible controller using status registers is shown in Figure 3.11b. The condition value is initially stored by *Reg1* in the data path; and it is then stored in the status register *CSGvar0* that is an internal control-path register created by the control path synthesis program *CSG*. The value of *CSGvar0* is set in time step 2 (*state2*) and is then discarded after time step 3 (*state3*). The state transition diagram is a simple ring shape; and the number of states is the same as the number of time steps. It is easy to see that the controller without status registers avoids the use of status registers at the cost of using extra states. In the extreme cases, the number of extra states is exponentially proportional to the number of status registers required in that time step.

Chapter 4

Preliminary 3D Scheduling Research

High-level synthesis systems start with an abstract behavioral specification, such as a VHDL description of a digital system and find a register-transfer level structure that realizes the given behavior [MPC88]. Data path synthesis can be subdivided into three subtasks, namely scheduling, module allocation and binding, which are the core of the transformation from behavior to structure. Those three subtasks are closely related. Scheduling involves assigning the operations to time steps. Module allocation allocates a set of modules (i.e. functional units, registers and interconnection units) which is sufficient to realize the schedule. Binding assigns the operations and values to modules. A time step denotes a fundamental sequencing time unit (a clock cycle) in synchronous systems.

The preliminary 3D scheduling approach uses the concept of *freedom* [PPM86] as its scheduling priority function. A floorplan is constructed simultaneously with scheduling in order to feedback wiring delays to the scheduler. Wiring area is estimated by a fudge factor that is proportional to the area of allocated functional modules. Note that the effects of multiplexers and registers are not considered in the preliminary 3D scheduling approach.

The basic idea behind the preliminary 3D scheduling work is as follows: when operations are scheduled and then functional modules are allocated, simultaneously we decide their shape and position on the floorplan. In the floorplanning process, the module with minimal area among the allocated functional modules is chosen as a basic area unit. A basic dimension unit is equal to the square root of the basic

area unit. A module with non-minimal area is round to be the ceiling integer time of the basic area unit.

To construct a floorplan, we first determine the number of basic area units required by the floorplan using the operator lower-bound estimation proposed by R. Jain [JPP87]. The bounding box of the floorplan is then approximated as a *discrete square* whose size is the number of the basic area units. A *discrete square* is a rectangle in which the difference between the width and height of the rectangle is less or equal to one basic dimension unit. The area of the rectangle of a *discrete square* is minimal. The shape of a newly allocated module is assumed to be a *discrete square* whose size is equal to the number of the basic area units, if it can be placed within the current bounding box. Otherwise, the shape of this newly allocated module is a general rectangle which results in the minimal area increase of the current bounding box in order to place the module.

The approach taken is to schedule the operations along the critical paths (longest paths through the data flow graph) first and assign operators according to their data dependencies. The critical paths are determined using the module delays obtained from the module library. Note that the wiring delays are assumed to be zero in the derivation of the critical paths. A partial floorplan for the currently allocated functional modules is constructed once all the operations along the critical paths are scheduled and assigned. The off-critical path operations are then scheduled and assigned to the operators (and placed when a new operator is allocated) with the minimum overall estimated interconnection length. After all the operations have been scheduled, the routine checks the timing constraints and tries pairwise interchanges to reduce the interconnection delays. *Redundant operators*[1] are introduced in order to reduce the interconnection delays if the routine still cannot achieve the timing constraints.

[1]Redundant operators are operators not required for the minimum feasible design.

4.1 Algorithm Description

The preliminary 3D scheduling approach can be divided into two parts: (1) scheduling and module assignment and (2) wiring delay improvement procedure. The first part is a constructive procedure; the second part is an iterative procedure.

4.1.1 Scheduling and Module Assignment

The preliminary 3D scheduling approach is to develop the schedule, module allocation and bindings simultaneously. We start from a null module set and add module units only when the scheduled operations cannot share existing ones. The ordering in the scheduling algorithm is to schedule the operations on the critical paths first. For the operations along the critical paths, we schedule the operations whose predecessors have been scheduled based on their data dependencies. The earliest possible time steps (ASAP) are assigned to the scheduled operations.

The scheduling approach is similar to that used by Nagle [NCP82] and used in MAHA [PPM86]. The rest of the unscheduled operations are scheduled by their *freedom* in ascending order. *Freedom* is defined as the time-step difference between the latest possible scheduled time step (ALAP) and the earliest possible scheduled time step (ASAP) for a given clock cycle. The operation with the least *freedom* is first scheduled to its earliest possible time step (ASAP). When two or more operations have the same *freedom*, the operation with the larger area will be scheduled first. Once each operation has been scheduled, we assign it to a specific module unit.

In module assignment, we use the best-first approach. We first assign the operation to the module available at that time step which performs the same function with the minimum estimated wiring cost. The wire length between two connected modules is estimated by the longest possible wire length (i.e. the worst case wire length). In our code, the fanout is assumed to be one for all the modules, but it would be trivial to assume a larger (constant) fanout. The wiring delay is then estimated by a first-order RC model taken from Gupta [Gup91], which is originally published by Sakurai [Sak83]. These assignments are iteratively improved later by the improvement procedure. The scheduling algorithm is outlined in Figure 4.1.

1. **comment:** *Process the operations along the critical path first*
2. **for** all operations along the critical path whose predecessors have been processed **do**
3. **begin**
4. **comment:** *Search all possible time slots on allocated module list*
5. **for** all allowable time steps **do**
6. **begin**
7. *use the best-first approach to find an available module which can perform the operation with minimum wiring cost*
8. **if** found
   then continue to next operation
   comment: *Otherwise, a new module needs to be allocated*
   else *allocate a new module of this operator type*
9. *free all the schedulings of this operator type and reschedule them*
10. **end**
11. **end**
12. **comment:** *Now, assign the positions for new allocated modules*
13. *generate a floorplan of allocated modules using the constructive method*
14. **comment:** *Process the rest of unscheduled operations*
15. *repeat the procedure from step 4 to step 9 for the off-critical path unscheduled operations, except use freedom as the priority function for choosing the operation to schedule next; floorplan unplaced modules as they are allocated*

Figure 4.1: Scheduling and Module Assignment Procedure

4.1.2 Improvement Procedure

The improvement procedure contains two phases, operation rebinding and redundant operator allocation. The purpose of both phases is to minimize the total wiring delay. Operation rebinding sequentially examines each potential module exchange which may result the reduce of the wiring delays. If the wiring delay is reduced by this exchange, we then switch the bindings of these two operators. Otherwise, the bindings are not changed.

After searching all the possible reassignments, redundant operators are allocated to reduce the wiring delays while the *generosity* [KP90] constraint is still satisfied. The *generosity* indicates the maximum tolerance of introducing redundant operators. From our experiments, we found that one iteration usually produces a satisfactory result. The outline of the improvement procedure is shown in Figure 4.2.

4.2 Experimental Results

A simple prototype program was written to verify the preliminary 3D scheduling approach. In this first prototype, only non-pipelined designs without conditional branches are handled. Also no delay nodes are inserted during the scheduling process. All of these limitations have been relaxed in our current program described in Chapter 5. The inputs to the prototype program are a data flow graph, a module library and a generosity factor. The constraint can be either the maximum allowable area cost or the maximum allowable execution delay. Note that the execution delay includes the delays both caused by the functional modules and interconnection wiring.

We applied our prototype to an FIR filter example. Figure 4.3 shows the snapshots of the scheduling process of a 2-time-step non-pipelined FIR filter design. There are two critical paths in this example which are shown by solid lines in the figures. A partial floorplan is constructed, which is shown in Figure 4.3c, once all the operations along the critical paths are scheduled. The final schedule of the 2-time-step non-pipelined FIR filter design is shown in Figure 4.4. A floorplan of the 2-time-step non-pipelined FIR filter design that minimizes the number of

1. **comment:** *Process the operations along the critical path first*
2. **for** all operations along the critical path whose predecessors have been processed **do**
3. **begin**
4. **comment:** *Search all possible exchanges*
5. **for** all modules with the same operator type **do**
6. **begin**
7. **if** this module is assigned to an operation on critical path **then** *skip it* and **continue**
8. **if** the total wiring cost is reduced after exchange **then** switch the module assignments of these two operations
9. **end**
10. **comment:** *Allocate a redundant operator to improve the wiring cost*
11. **if** allocated redundant operators *greater* than generosity allowed **then goto** step 2 to process next operation
12. *allocate a redundant operator and place it as close as possible to its predecessor to reduce the total wiring delays*
13. **if** the total wiring cost is reduced **then** put the allocated redundant operator into allocated module list **else** free the allocated redundant operator
14. end
15. **comment:** *Processing the non-critical path operations*
16. **for** all operations ordered by freedom **do**
17. **begin**
18. *repeat the procedure from step 4 to 13, except skip the possible exchanges with operations along the critical path*
19. end

Figure 4.2: Iterative Improvement Procedure

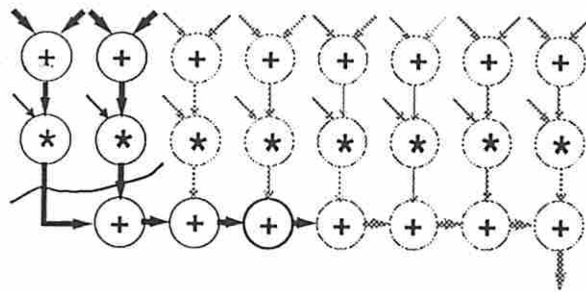
Technology	16 bit adder		16 bit multiplier	
	Delay (ns)	Area (mil^2)	Delay (ns)	Area (mil^2)
3 μm	34	4200	375	49000
2 μm	22	1867	250	21778
1.6 μm	18	1195	200	13938
1.2 μm	13	672	150	7840

Table 4.1: First Library Set Used for FIR Filter

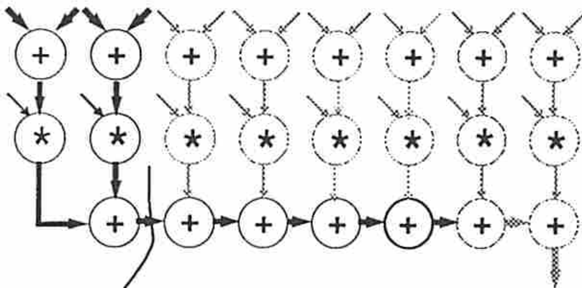
operators is shown in Figure 4.5. The floorplan of an alternative design with a redundant adder is shown in Figure 4.6. Those two designs took 0.2 second of CPU time on a SUN 4/460 machine.

Table 4.1 lists the first library set we used for this example. For comparison purposes, we first linearly scaled down each functional unit area and delay from 3-micron process parameters. We used different device parameters for different fabrication processes to calculate wiring delay in this program. We later show an actual scaled library (see Table 4.3) used to test our concepts. Table 4.2 lists the predicted delay time for both designs for the linearly scaled fabrication technology. The errors caused by considering functional delays only are shown clearly in this table. The library set shown in Table 4.1 is originally obtained from a 3-micron RCA design library. The linearly scaled adder and multiplier of the 1.2-micron technology are still very large as compared to the adder and multiplier derived from a 1.2-micron silicon compiler (see Table 6.1). For example, the 1.2-micron multiplier in Table 4.1 is three times larger than the multiplier shown in Table 6.1. The large module sizes and long operator chains (i.e. five operators are maximally chained in a time step) result in $44ns$ estimated wiring delays in each time step for the design of 1.2-micron process shown in Table 4.1.

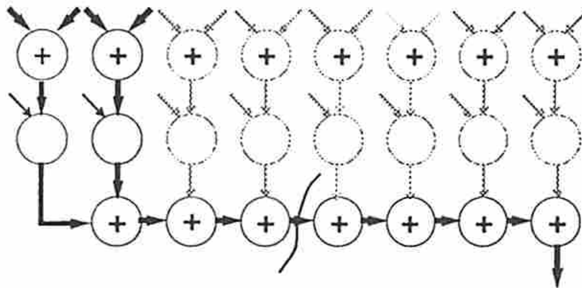
By inspecting Table 4.2, we found that the differences between functional delays and the overall system delay are quite linear. However, this is not true of real process parameters. We believe this phenomenon is due to linearly scaling the area and delay of functional units. For this reason, we used Cascade Design Automation ChipCrafter to create an 8-bit adder and an 8-bit multiplier with 1.2 micron fabrication technology. The library data is shown in Table 4.3. We resynthesized the FIR 2-time-step example, and the resynthesized results are shown in



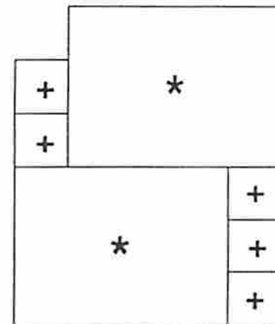
a) 2 Adders and 2 Multipliers allocated

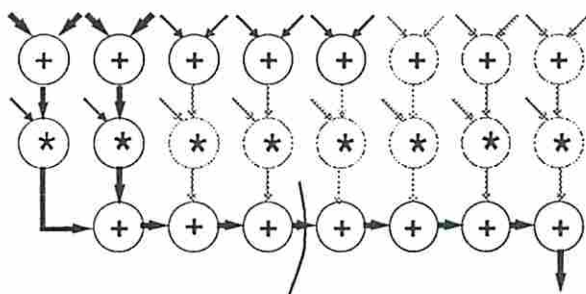


b) 3 Adders and 2 Multipliers allocated



c) 5 Adders and 2 Multipliers allocated





d) 7 Adders and 2 Multipliers allocated

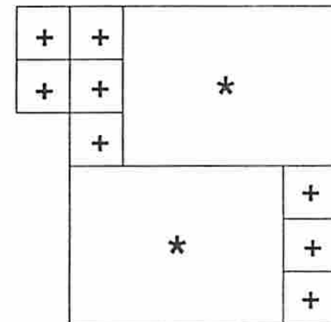


Figure 4.3: The Scheduling Process of a 2-Time-Step Non-pipelined FIR Filter

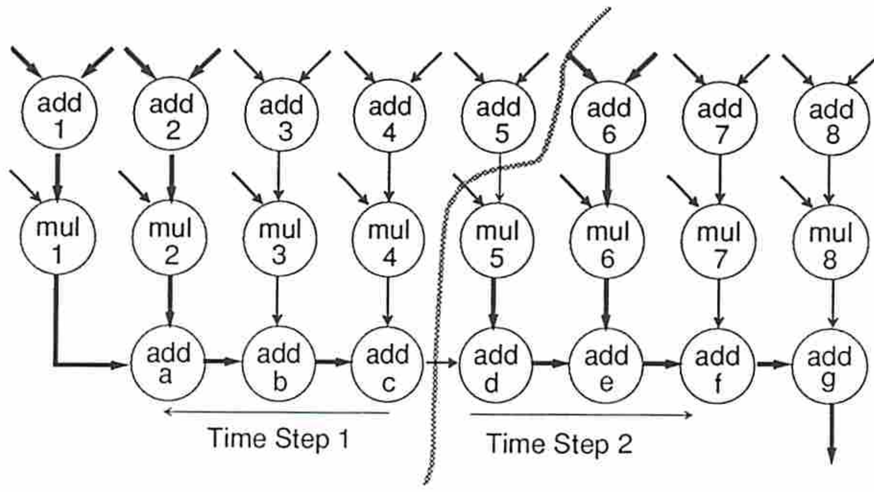


Figure 4.4: The Schedule of the 2-Time-Step Non-pipelined FIR Filter Design

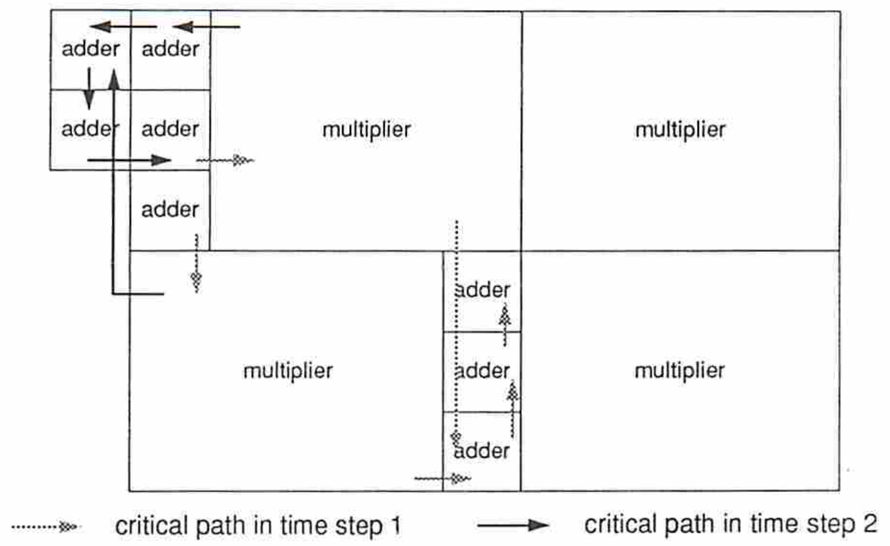


Figure 4.5: The Floorplan of the 2-Time-Step FIR Filter with Minimum Operators

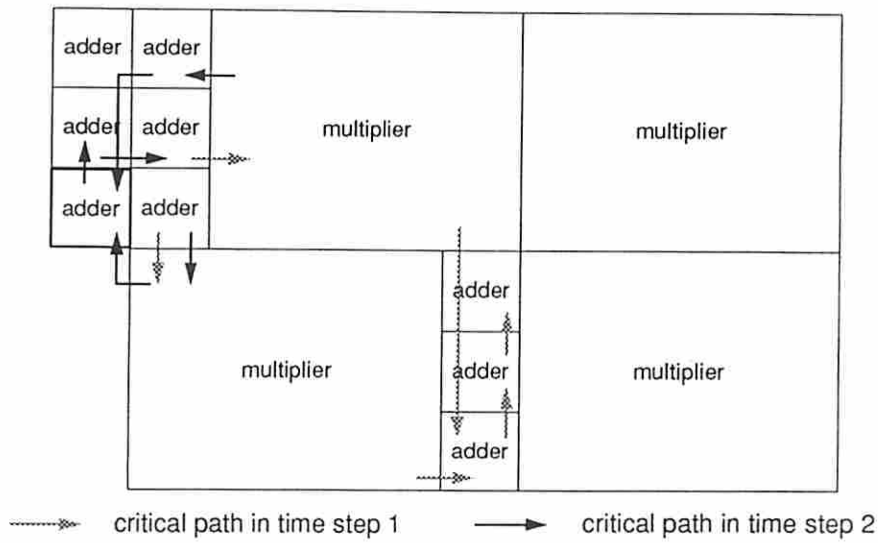


Figure 4.6: The Floorplan of the 2-Time-Step FIR Filter with a Redundant Adder

3 μm Technology	Time Step 1	Time Step 2	Total Delay	Est. Area
Functional Units	511 ns	511 ns	1022 ns	229600 mil^2
with Est. Wiring	586 ns	589 ns	1178 ns	275520 mil^2
with a Red. Adder	586 ns	583 ns	1172 ns	280560 mil^2
2 μm Technology	Time Step 1	Time Step 2	Total Delay	Est. Area
Functional Units	338 ns	338 ns	676 ns	102048 mil^2
with Est. Wiring	419 ns	423 ns	846 ns	122458 mil^2
with a Red. Adder	419 ns	416 ns	838 ns	124698 mil^2
1.6 μm Technology	Time Step 1	Time Step 2	Total Delay	Est. Area
Functional Units	272 ns	272 ns	544 ns	65312 mil^2
with Est. Wiring	326 ns	328 ns	656 ns	78374 mil^2
with a Red. Adder	326 ns	324 ns	652 ns	79808 mil^2
1.2 μm Technology	Time Step 1	Time Step 2	Total Delay	Est. Area
Functional Units	202 ns	202 ns	404 ns	36736 mil^2
with Est. Wiring	244 ns	246 ns	492 ns	44083 mil^2
with a Red. Adder	244 ns	243 ns	488 ns	44890 mil^2

Table 4.2: Minimum-Operator Designs versus Redundant-Operator Designs

Technology	8 bit adder		8 bit multiplier	
	Delay (ns)	Area (mil^2)	Delay (ns)	Area (mil^2)
1.2 μm	7	75	32	903

Table 4.3: Library Set Created by ChipCrafter

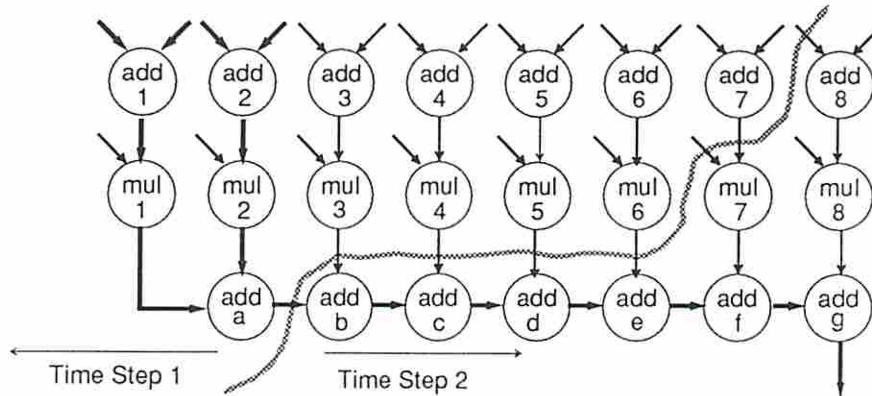
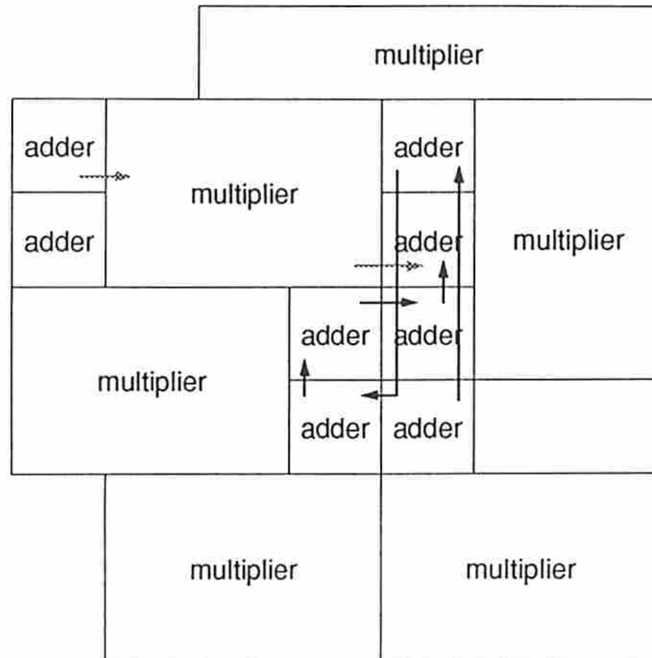


Figure 4.7: A Non-pipelined FIR Filter Using ChipCrafter Library Set

Figures 4.7, 4.8 and 4.9 and Table 4.4. The errors caused by considering only functional delays still remain the same (about 20% to 30%). The effect of introducing redundant operators is also obvious, especially with submicron processes, since the area overhead decreases in significance. It should be noted that even though wiring delays are reduced for the same design, as feature size decreases, smaller feature sizes allow large designs to fit into a single chip. For these large designs, wires will be longer and wiring delays will have a significant effect on performance.

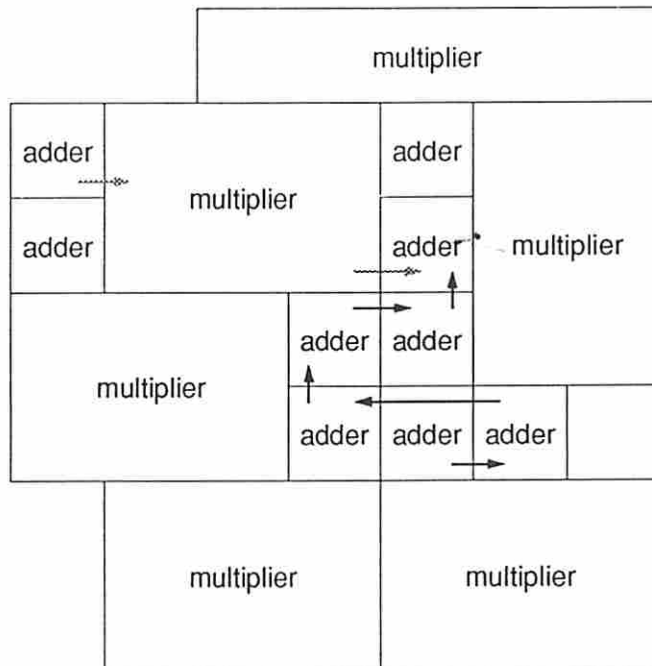
1.2 μm Technology	Time Step 1	Time Step 2	Total Delay	Est. Area
Functional Units	46 ns	46 ns	92 ns	6018 mil^2
with Est. Wiring	59 ns	62 ns	124 ns	7222 mil^2
with a Red. Adder	59 ns	59 ns	118 ns	7312 mil^2

Table 4.4: A FIR Filter 2-Time-Step Design Using ChipCrafter Library Set



Total Delay = 124 *ns* Dimension : 119 *mil* by 119 *mil*

Figure 4.8: Floorplan with Minimum Operators Using ChipCrafter Library Set



Total Delay = 118 ns
(5 % improvement)

Dimension : 119 mil by 119 mil

Figure 4.9: Floorplan with a Redundant Adder Using ChipCrafter Library Set

Chapter 5

Data Path Synthesis

This chapter presents a comprehensive integrated data path synthesis technique called 3D scheduling which performs scheduling simultaneously with floorplanning. The 3D scheduler performs scheduling, module allocation, binding and floorplanning at the same time to meet the user specified constraints. The inputs to the 3D scheduler are a VHDL description, the design style (e.g. non-pipelined design style or pipelined design style), a tentative clock cycle, a module library, the process-dependent parameters and the design constraints. The constraints include the performance constraint (i.e. the total execution delay), area constraint (i.e. the chip area), and the floorplan aspect-ratio constraint. The outputs of the 3D scheduler are a schedule, a set of allocated functional modules, a set of operator bindings, a set of value bindings, and a set of floorplans with different aspect ratios.

The 3D scheduling technique uses a modified force function similar to that of Paulin [PK89] to perform scheduling as well as binding for the operations in the data flow graph. Since one step lookahead is used by the force function heuristic, it is expected to give us more globally optimum results than other techniques.

In the 3D scheduling approach, the effects of interconnections are considered during the scheduling process using a concurrently constructed floorplan. Input multiplexers are estimated, allocated and placed in the floorplan when a functional unit is allocated. Wiring delays which guide the scheduling process to produce better schedules are estimated using the first order RC model presented in Section 5.2.2.2. The worst-case wire lengths are used to estimate the wiring delays in our current approach. The worst-case wire length of two connected modules is

equal to the half perimeter of the box enclosing these two modules. Since the fanout of a module is unknown until the register-transfer level network is completed, the fanout is currently assumed to be one for all the modules in our approach. The prediction/estimation and floorplanning techniques are used by the 3D scheduler to make effective scheduling decisions.

In the rest of this chapter, the force-directed scheduling proposed by Paulin [PK89] is first discussed. The prediction technique used in 3D scheduling is given. The 3D scheduling technique is then introduced, followed by a presentation of the constructive cluster-cut-tree floorplanning technique.

5.1 Force-Directed Scheduling

Force-directed scheduling proposed by Paulin [PK89] places similar operations in different time steps, so as to balance the concurrency of operations assigned to the functional units without increasing the total execution time. By balancing the concurrency of operations, each functional unit is ensured to have a high utilization which in turn decreases the total number of units required. This balancing is done in three steps: determination of a possible time frame for each operation, creation of a distribution graph and calculation of the force associated with each possible assignment. We repeatedly select the assignment associated with minimum force and execute the force calculation process until all the operations are assigned.

Determine possible time frame

The time frame of each operation is determined by evaluating ASAP (as soon as possible) and ALAP (as late as possible) schedules. By combining the results for both schedules, we can derive the time frame for each operation. The data flow graph of a partial FIR filter design shown in Figure 5.1 illustrates this process. Nodes represent the functional operations, and edges represent the data dependencies between these operations.

We assume the clock cycle delay due to functional units is $53ns$, the delay of the adder is $25.5ns$ and the multiplier is $53ns$. The ASAP and ALAP schedules are shown with horizontal lines in the figure. The resulting time frame is drawn in

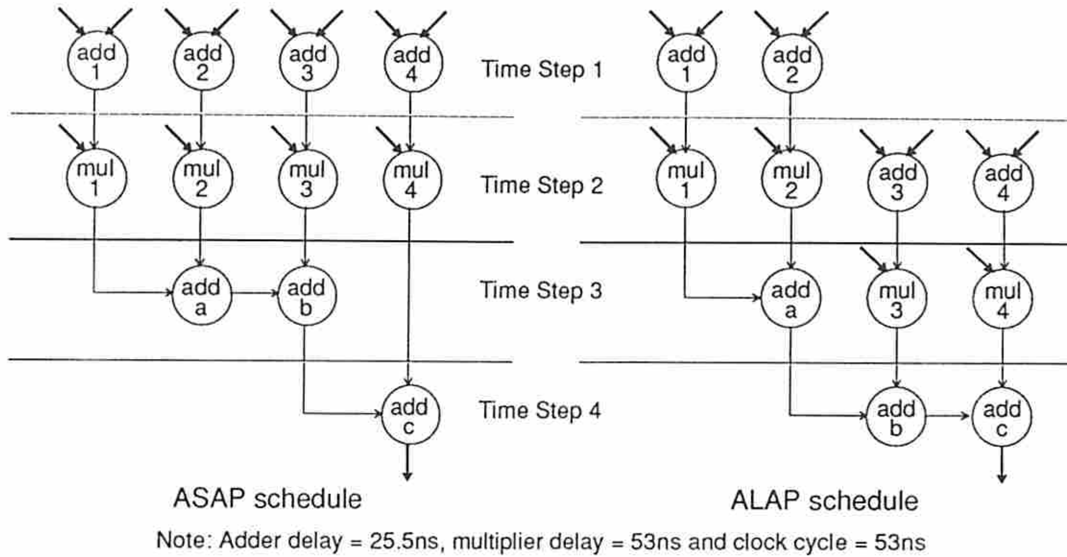


Figure 5.1: A 4-time-step FIR Filter ASAP and ALAP Schedule

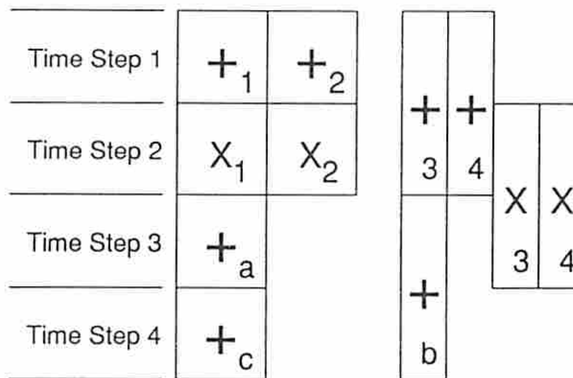


Figure 5.2: Time Frames for the 4-time-step FIR Filter Example

Figure 5.2. The width of each box represents the possibility that an operation will be assigned to any given time step. Without lose of generality, the possibilities of operation assignments are assumed to be uniformly distributed.

Creation of the distribution graph

The next step is to sum up the probabilities of each type of operation for each time step in a data flow graph.

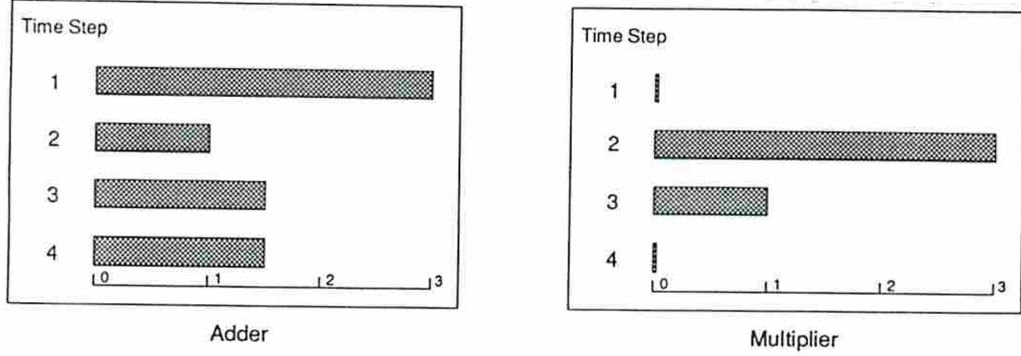


Figure 5.3: Distribution Graph for 4-time-step FIR Filter Example

Definition 5.1 *The distribution graph $\mathcal{DG}_k(i)$ of a data flow graph of type k operations in time step i is defined as*

$$\mathcal{DG}_k(i) = \sum_{o \in Opn_k} Prob(o, i) \tag{5.1}$$

where Opn_k is the set of operations of type k in the data flow graph; and $Prob(o,i)$ is the probability of the operation o assigned in time step i.

The distribution graphs indicate the concurrency of similar operation type(s) in a time step. Using Figure 5.2, the distribution graphs $\mathcal{DG}_+$ and $\mathcal{DG}_\times$ for the addition and multiplication operations are calculated. The results are depicted in Figure 5.3.

Force calculation

The last step is to calculate the forces associated with every feasible time step scheduling of each operation. The force-directed scheduling process temporarily reduces the operation's time frame to the selected time step. The force associated with the reduction of the initial time frame (bounded by time steps t and b) to a new time frame (bounded by time steps nt and nb) is calculated by the following equation:

$$Force(nt, nb) = \sum_{i=nt}^{nb}[\mathcal{DG}_k(i)/(nb - nt + 1)] - \sum_{i=t}^{b}[\mathcal{DG}_k(i)/(b - t + 1)] \tag{5.2}$$

Each sum represents the average of distribution values for the time steps bounded by that time frame. The force is therefore equal to the difference between the average distribution for the time steps bounded by the new time frame and the average for time steps of the initial one. The force induced by assigning a operation o to a specific time step i itself is called *direct force.* We also calculate the force for all predecessors and successors of the current operation whenever their time frames are affected. These additional forces are called *indirect forces.* The total force is the sum of the direct and indirect forces.

The force is calculated for each possible assignment of every operation in the data flow graph. In each force calculation pass, the operation assignment with the minimal force is assigned. The possible time frames of unscheduled operations are recalculated after each operation assignment. The scheduling process proceeds by repeatedly assigning a schedule to the unscheduled operation with minimal force and is terminated when all the operations in the data flow graph are scheduled.

5.2 Prediction Technique

The prediction model includes the lower-bound estimation for the number of functional modules, and area and delay predictions for the wiring and controller. The predictions are based solely on the behavior of the target design, its area/performance constraints, the tentative clock cycle time and a module library. Predictions are performed by scanning through the input behavior without dealing with the actual synthesis details. Predictions are fast and helpful in guiding the data path synthesis program to prune infeasible designs at an early synthesis stage. Due to the close relationship between the scheduling and binding processes, predictions are also helpful in determining operation-to-operator bindings more globally. The prediction methods introduced in this section combine former theoretical and heuristic prediction research done at USC along with new additions to operator lower-bound estimation.

5.2.1 Operator Estimation

The input to the operator estimation consists of an user-input tentative clock cycle time,[1] a directed acyclic data flow graph[2] (can be obtained from an input VHDL behavioral description), a module library, tentative task delay and initiation interval (for pipelined designs only). A single-cycle architecture model is assumed in the operator estimation. The single-cycle architecture model is defined as a scheduling model in which all operations are assumed to be combinational and take at most one time step to execute. Therefore the clock cycle time has to be longer than any operation delay to ensure the correct execution of the behavioral specification.

The pipeline model (functional pipelining) assumed in this section is the same as the one in [PP88] [Jai89]. A non-pipelined design is modeled as a special case of a pipelined design with the same initiation interval as the pipeline length. In the following operator estimation, we will discuss the theorems/heuristics for pipelined designs only; the non-pipelined designs are handled in a similar fashion. We first define the *utilization* of an operator type. Then the lower-bound prediction for pipelined designs used by R. Jain [JPP87] is introduced. Next, a new tighter operator lower-bound estimation algorithm for non-pipelined designs is presented.

5.2.1.1 An Operator Lower-Bound Estimation for Pipelined Designs

Definition 5.2 *The utilization u_k of operator type k in a pipelined design is defined as*

$$u_k = \frac{n_k}{o_k \times l} \tag{5.3}$$

where n_k is the number of type k operations in the behavior, o_k is the number of type k operators in the register-transfer structure and l is the initiation interval of the pipelined design.

---

[1]The tentative clock cycle should be greater than the delays of the operations in the input data flow graph. For a multiple-chip design, the tentative clock cycle is usually the system clock cycle.

[2]For a design with inner loops, the data flow graph is still acyclic. The loop information is characterized in the timing graph but not considered in the operator estimation process.

In [PP88], Park predicts the lower bound on the operation allocation for a pipelined design as shown in Equation 5.4:

$$o_k \geq \lceil \frac{n_k}{l} \rceil \tag{5.4}$$

The existence of the ceiling function in Equation 5.4 is due to the fact that discrete characteristics of the operators and any fractional requirements for operators have to be rounded to the smallest integer larger than the required amount. If the distribution graph is used to explain the operator lower-bound estimation, then Equation 5.4 is rewritten as

$$o_{k(st,et)} \geq \lceil \frac{\sum_{t=st}^{et} \mathcal{DG}_k(t)}{et - st + 1} \rceil \tag{5.5}$$

where $o_{k,(st,et)}$ is the operator of type k needed to implement the design, st and et are the starting and ending time step of the design, respectively, and $1 \leq st \leq et \leq l$. For an n-time-step pipelined design with an initiation interval l, o_k is equivalent to $o_{k,(1,l)} \geq \lceil \frac{\sum_{t=1}^{l} \mathcal{DG}_k(t)}{l} \rceil$. Equation 5.4 shows that the minimal number of type i operations needed to implement the operations of type i from time step st to time step et can be estimated by summing all the type i distribution graphs from time step st to time step et.

5.2.1.2 An Operator Lower-Bound Estimation for Non-Pipelined Designs

Equation 5.4 shows the theoretical lower bound on the operation allocation for a pipelined design without looking into the structure of the data flow graph. The operator lower bound estimated by Equation 5.4 is exact (i.e. necessary and sufficient) if there are no maximum time constraints between operations. However, a tighter operator allocation lower bound can be derived for a non-pipelined design by inspecting the distribution graphs of operation types in a data flow graph. We define a *maximally consecutive distribution window* $\mathcal{DG}_{k,mc}$ first. The procedure for estimating a tighter operator lower bound for non-pipelined designs is then presented.

Definition 5.3 *A distribution window* $\mathcal{DG}_{k,w}(st, et)$ *is a subgraph of distribution graph* $\mathcal{DG}_k$ *of operator type* k*. It is characterized by a starting time step* st *and an ending time step* et *for an* N*-time-step pipelined design, where* $1 \leq st \leq et \leq N$*.*

Definition 5.4 *A consecutive distribution window* $\mathcal{DG}_{k,c}(st, et)$ *is a distribution window* $\mathcal{DG}_{k,w}(st, et)$ *of operator type* k *in which no* $\mathcal{DG}_k(i)$ *is zero for* $st \leq i \leq et$*.*

Definition 5.5 *A consecutive distribution window* $\mathcal{DG}_{k,c}(mst, met)$ *is called a maximally consecutive distribution window* $\mathcal{DG}_{k,mc}(mst, met)$ *if* $\mathcal{DG}_k(mst - 1) = 0$ *and* $\mathcal{DG}_k(met + 1) = 0$*.*

The following procedure, Procedure 1, using the characteristic of maximally consecutive distribution windows iteratively calculates a tighter operator lower bound for a given non-pipelined data flow graph. $\mathcal{X}_{op}$ of a distribution window $\mathcal{DG}_{k,w}(st, et)$ denotes the set of operations of type k which can be scheduled into time step i when $\mathcal{DG}_k(i)$ is smaller than the currently estimated operator lower bound for $st \leq i \leq et$. Note that $\mathcal{X}_{op}$ is an empty set at the beginning of the operator lower-bound estimation.

Procedure 1, *LowerBound*, is a recursive procedure. The basic idea of this procedure is to estimate the operator lower bound of the current distribution window iteratively using Equation 5.4. The procedure first computes the operator low bound of the current distribution window and $\mathcal{X}_{op}$ is then recalculated. The distribution graph of the current distribution window is recomputed due to the change of $\mathcal{X}_{op}$. The operations in $\mathcal{X}_{op}$ are excluded in the calculation of the distribution graph because these operations could be scheduled into the time step whose expected operator demand (i.e. the value of the distribution graph) is lower than the current estimated operator lower bound. The maximally consecutive distribution windows are then identified and the operator lower bounds of the maximally consecutive distribution windows are derived similarly by recursively calling *LowerBound*. The procedure stops and outputs the estimated operator lower bound when the recursion termination condition (i.e. st is equal to et) is satisfied.

GenerateDG$(\mathcal{G}, st, et, \mathcal{X}_{op})$ calculates the distribution graph of the distribution window $\mathcal{DG}_{k,w}(st, et)$ from time step st to time step et for all the type k operations in the data flow graph $\mathcal{G}$ except the ones already put into $\mathcal{X}_{op}$. To estimate

the tighter operator lower bound of type k for a non-pipelined design, *Lower-Bound*$(\mathcal{G}, 1, N, \emptyset)$ is calculated.

Procedure 1

/* *Subroutine LowerBound*$(\mathcal{G}, st, et, \mathcal{X}_{op})$ *calculates operator lower bound of type k for the digital circuit behavior specified by the data flow graph* $\mathcal{G}$. */

if st is equal to et

/* *No more iterations are necessary.* */

return $\mathcal{DG}_k(st)$

/* *Calculate the lower bounds for the distribution window* $\mathcal{DG}_{k,w}(st, et)$. */

$\mathcal{OP}_{lb} = \lceil \frac{\sum_{i=st}^{et} \mathcal{DG}_k(i)}{et-st+1} \rceil$

$\mathcal{X}_{op} = \mathcal{X}_{op} \cup \{op \mid op$ can be scheduled to time step i and $\mathcal{DG}_k(i) \le \mathcal{OP}_{lb}$ for $st \le i \le et\}$

/* *Recalculate the distribution window* $\mathcal{DG}_{k,w}(st, et)$ *due to the change of* $\mathcal{X}_{op}$. */

GenerateDG$(\mathcal{G}, st, et, \mathcal{X}_{op})$

for every *maximally consecutive distribution window* $\mathcal{DG}_{k,mc}(mst, met)$ between time step st and et **do**

$\mathcal{OP}_{lb} = \max(\mathcal{OP}_{lb}, LowerBound(\mathcal{G}, mst, met, \mathcal{X}_{op}))$

return $\mathcal{OP}_{lb}$ □

The following example shows the use of Procedure 1.

Example: For the 4-time-step non-pipelined FIR filter design shown in Figure 5.1, the multiplier lower bound is first estimated as $\lceil \sum_{i=1}^{4} \mathcal{DG}_\times(i) \rceil / 4 = 1$ by applying Procedure 1. Since $\mathcal{DG}_\times(1)$ and $\mathcal{DG}_\times(4)$ are zeros which are less than the estimated lower-bound value 1, we can remove $\mathcal{DG}_\times(1)$ and $\mathcal{DG}_\times(4)$ from our lower-bound estimation. The distribution window now is from time step 2 to time step 3, and the new lower bound then becomes $\lceil \sum_{i=2}^{3} \mathcal{DG}_\times(i)/2 \rceil = 2$.

Since $\mathcal{DG}_\times(2)$ is 3 (that is greater than the currently estimated lower bound), we do nothing for this time step. $\mathcal{DG}_\times(3)$ is 1 (that is less than the currently estimated lower bound) and it is contributed to by *mul3* and *mul4*, so $\mathcal{X}_{op}$ is equal to $\emptyset \cup \{mul3, mul4\}$ which is $\{mul3, mul4\}$. *GenerateDG*$(\mathcal{G}, 2, 2, \mathcal{X}_{op})$ is then called

to recalculate the distribution graph due to the change of $\mathcal{X}_{op}$. $\mathcal{DG}_{\times}(2)$ is thus equal to 2. Since no further iterations are possible, the multiplier lower bound estimated by Procedure 1 is equal to 2.

As compared to the multiplier lower bound estimated by Equation 5.4 (i.e. $\lceil \frac{4}{4} \rceil = 1$), Procedure 1 predicts a tighter multiplier lower bound for this non-pipelined example. Notice that the multiplier lower bound estimated by Equation 5.4 is insufficient to realize a design, but the multiplier lower bound predicted by Procedure 1 can implement a real-life design.

□

Procedure 1 provides a systematic way to estimate operator lower bounds using distribution graphs. As compared to Hagerman's work at Carnegie Mellon University [Hag91], the estimation result of an operator lower bound strongly depends on the order of manipulating distribution graphs (i.e. a different order in managing distribution graphs may obtain a different operator lower-bound estimation result).

Theorem 5.1 *Procedure 1, LowerBound($\mathcal{G}, 1, N, \emptyset$), computes the operator lower bound of type k for a non-pipelined design and the operator lower bound estimated is greater than or equal to the one calculated by Equation 5.4.*

Proof: From the estimation of the loose operator bound, Equation 5.4 can be rewritten to Equation 5.5 $o_{k(st,et)} \geq \lceil \frac{\sum_{t=st}^{et} \mathcal{DG}_k(t)}{et-st+1} \rceil$. Since the operations in $\mathcal{X}_{op}$ can be scheduled into any given time step i and $\mathcal{DG}_k(i)$ is smaller than the currently estimated operator lower bound $\mathcal{OP}_{lb}$, it is then possible to schedule the operations into time step i in order to increase the operator utilization. In Procedure 1, a maximally consecutive distribution window is derived by excluding the operations in $\mathcal{X}_{op}$, so that the new the operator lower bound is derived from the new maximally consecutive distribution graph. The operator lower bound of the new maximally consecutive distribution graph is also the lower bound of the data flow graph $\mathcal{G}$, because the new maximally consecutive distribution graph is contributed to by a subset of operations in the data flow graph $\mathcal{G}$.

Procedure 1 initially calculates the operator lower bound $\mathcal{OP}_{lb}$ from $st = 1$ and $et = N$ using Equation 5.5. The procedure then recursively searches the new

maximally consecutive distribution windows within the current distribution windows due to the change of $\mathcal{X}_{op}$. The operator lower bounds of the new maximally consecutive distribution windows are calculated. Since the new operator lower bound $\mathcal{OP}_{lb}$ is the maximum of the current $\mathcal{OP}_{lb}$ and the operator lower bounds newly calculated, the operator lower bound estimated by $LowerBound(\mathcal{G}, 1, N, \emptyset)$ is greater than or equal to the one calculated by Equation 5.4. □

5.2.2 Wiring Estimation

Considering only the functional characteristic of RTL design is usually not enough to make correct design decisions in high-level synthesis. With the incorporation of the floorplanning process into the scheduling process, the effects of wiring delays can then be estimated more precisely than traditional approaches. In our wiring estimation, a channel routing technique is assumed in the layout process. We first present a wiring area approximation technique. A first-order RC model used to estimate wiring delays is then introduced.

5.2.2.1 Wiring Area Estimation

In the estimation of wiring area, a channel routing technique is assumed in the layout system; the input/output, clock and power terminals of the module are assumed uniformly distributed around the perimeter. A technology dependent routing factor called *Track Utilization* $\mathcal{U}_t$ is used to reflect the possibility of routing wires that share a routing channel in a specific layout system. We assume the wiring tracks are equally distributed along the boundary. The effect of extra area induced by the wiring is modeled by $\triangle x$ on each side which is estimated as

$$\triangle x = \frac{ibw_i + obw_i + cbw_i + pbw_i}{4} \times \mathcal{W}_t \times \mathcal{U}_t \qquad (5.6)$$

where ibw_i and obw_i are the input and output bit widths of module i; cbw_i and pbw_i are the bit widths required to connect the clock and power terminals for module i. $\mathcal{W}_t$ is the average width of a routing track in the layout system under current

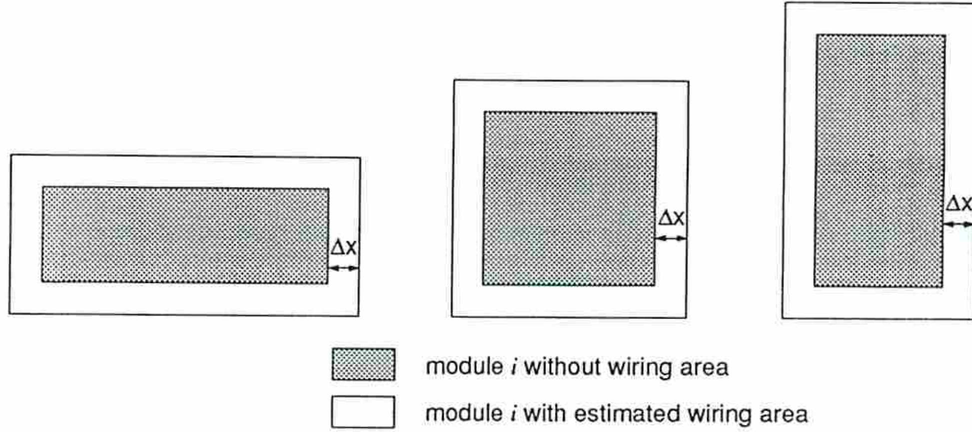


Figure 5.4: The Estimation of Wiring Area

technology. The effective area of a module including the wiring area $\mathcal{A}_e$ can be approximated by

$$\mathcal{A}_e \approx (\mathcal{W} + 2 \cdot \triangle x) \times (\mathcal{H} + 2 \cdot \triangle x) \tag{5.7}$$

where $\mathcal{W}$ and $\mathcal{H}$ are the width and height of the module before considering the wiring area. Note that $(\mathcal{W} + 2 \cdot \triangle x)$ and $(\mathcal{H} + 2 \cdot \triangle x)$ are the effective width and height of this module, respectively. Figure 5.4 shows examples of a module with different aspect ratios before and after considering wiring area. Notice that a module with different aspect ratios may result in a different wiring area in the routing process, which is true in the real world design. The effective area of the allocated modules will be used in the floorplanning process to take the impacts of wiring area into account. Wiring area will increase the wire lengths which result longer wiring delays in a design.

5.2.2.2 Wiring Delay Estimation

Wiring delay estimation is based on a first-order RC model, taken from Gupta [Gup91] and was originally published by Sakurai [Sak83]. In this model, the wiring delay is modeled as a function of the wire length and fanout of the driver. In the wiring delay model, wire delay arises from signal propagation through the RC network and also from charging the load capacitance. The wire length is used to

estimate the sheet resistance and capacitance along the signal propagation in the RC delay model; the fanout of the driver is used to estimate the load capacitance and the charging time associated with that capacitance. Wiring delay can therefore be represented by the following first-order relationship [Sak83]:

$$t_{0.9} = 1.02RC + 2.21(C_tR_t + C_tR + CR_t) \quad (5.8)$$

where:

$R =$ Total resistance of the wiring.
$C =$ Total capacitance of the wiring.
$C_t =$ Load capacitance.
$R_t =$ Equivalent resistance of the driving transistor.

R, C, R_t and C_t are calculated as described in [Gup91]. These parameters are strongly dependent on the design and fabrication process. In the case of metal wiring R is very low as compared to R_t and can be neglected in the estimation procedure. Therefore, Equation 5.8 reduces to ([Gup91])

$$t_{0.9} = 2.21R_t(C_t + C) \quad (5.9)$$

Using Equation 5.8 (or Equation 5.9), we can calculate the worst-case wiring delay from the worse-case wire length. The worst-case wire length is measured by the diagonal rectilinear distance of the bounding box containing the two connected modules on the floorplan. Although this is only a first-order RC model, it does produce a very good approximation of wiring delays as Gupta verified by SPICE simulations [Gup91].

5.2.3 Controller Estimation

For the controller estimation, we will focus on controllers implemented by programmable logic arrays (PLAs). The estimation of controller characteristics is performed in two phases. The first phase estimates PLA characteristics, namely the number of PLA *inputs*, the number of PLA *outputs* and the number of PLA

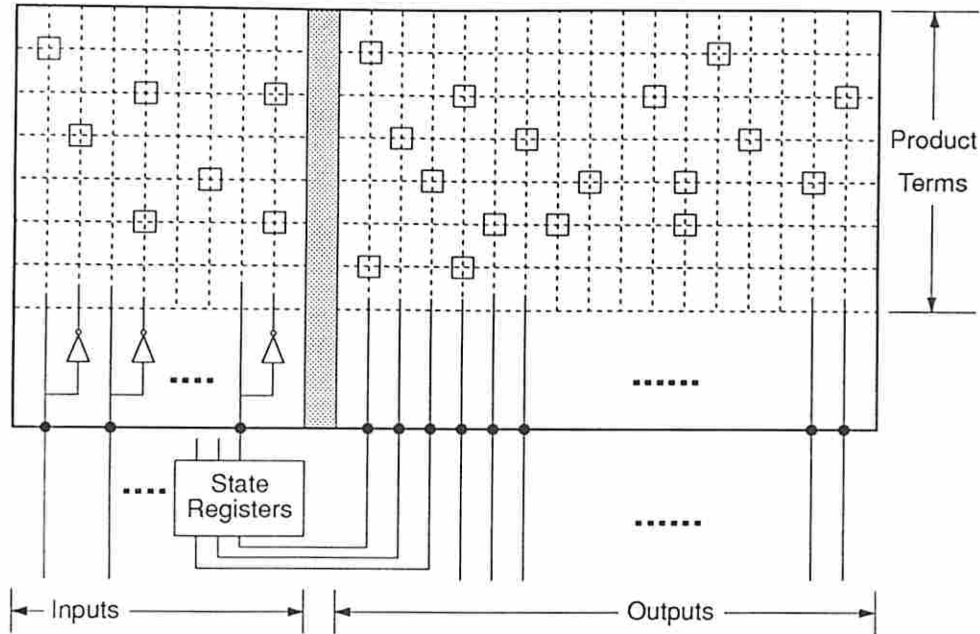


Figure 5.5: A Finite State Machine Implementation Using PLA

product terms. The second phase estimates the actual PLA area and delay from abstract PLA characteristics.

5.2.3.1 PLA Abstract Characteristic Estimation

Estimations of PLA abstract characteristics are taken from Mlinar [Mli91]. As depicted in Figure 5.5, a PLA consists of two regions. On the left side, one or more *inputs* (and/or their complements) are *AND*ed together to form each product term. On the right side, each ***output*** line is driven by one or more product terms which are *OR*ed together.

Several assumptions are made in the development of this PLA characteristic estimation model:

- The controller area/delay estimations are specific to finite state machines.
- The PLA state feedback registers and output registers are considered to be part of the controller.
- Wiring area/delay within a controller is estimated similar to the wiring area/delay estimated in the data path.

Inputs to a PLA finite state machine include external inputs and the state/status signals fed back from the outputs of the PLA, as shown in Figure 3.2. These external inputs reflect the status of external hardware or impacts of loop control/conditional branch execution. The number of states ζ in a finite state machine determines the number of state bits, $\lceil \log_2 \zeta \rceil$. The number of PLA inputs i_c thus can be estimated as ([Mli91])

$$i_c = \lceil \log_2 \zeta \rceil + i_{cond} + i_{loop} \quad (5.10)$$

where i_{cond} is the number of condition value inputs, which is affected by the type and the number of conditional branches in the data path; i_{loop} is the number of loop status inputs, which is determined by data path and/or loop counter status. Similarly, the PLA outputs can be estimated as ([Mli91])

$$o_c = \lceil \log_2 \zeta \rceil + R_c + M + A + o_{loop} + o_{cond} \quad (5.11)$$

where R_c and M are the number of control signals associated with registers and multiplexers, respectively. A is the number of control signals for operating ALUs and other multi-function hardware. o_{loop} and o_{cond} are the control signals related to loop counter controls and conditional branch selections. The prediction of the number of product terms p_c in a PLA is the major difficulty in controller estimation. It is clear that the number of the product terms in a PLA is a function of control states as well as the number of output control signals. Loops and conditions also have impacts on the determination of product terms of a PLA. More details about Mlinar's PLA estimation techniques can be found in [Mli91].

5.2.3.2 Controller Area Estimation

The area of a controller consists of three parts, the PLA, state/status/output registers and wiring area. The PLA area is based on predefined macro cells as proposed by Mlinar. Given the number of PLA inputs i_c, outputs o_c and product terms p_c from earlier equations, the area of a PLA can be defined as [Mli91]

$$PLA_{area} = c_1 \cdot (2i_c + o_c) \cdot p_c + c_2 \cdot p_c + c_3 \cdot (2i_c + o_c) + c_4 \quad (5.12)$$

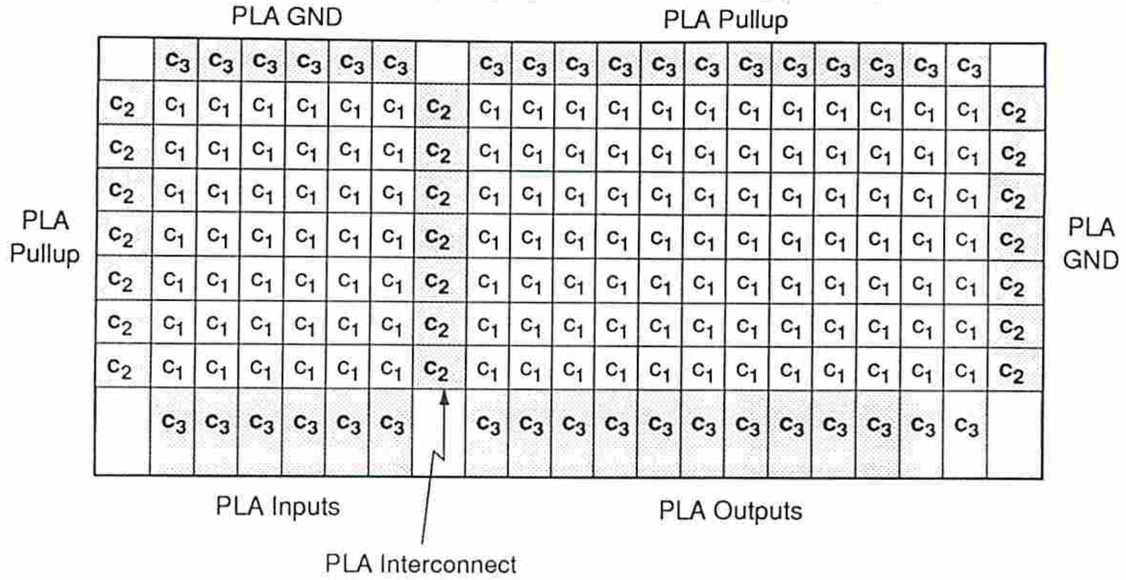


Figure 5.6: Internal Construction of PLA (Taken from [Mli91])

where c_1, c_2, c_3 and c_4 are constants depending on the technology. Each term in Equation 5.12 represents a portion of the PLA area, which is clarified by Figure 5.6, taken from [Mli91].

The number of registers required in the controller is equal to the number of PLA outputs. Wiring area is estimated in a similar manner to the approach in Section 5.2.2.1. The geometry of the estimated PLA is presumed to be a rectangular block with aspect ratio being $\frac{i_c+o_c}{p_c}$ and wiring area is estimated as previously. The controller area thus includes the area of the PLA and registers,[3] with estimated wiring area in each module.

5.2.3.3 Controller Delay Estimation

PLA delays are transition and personality matrix dependent. Our aim is to estimate the PLA delay according to the number of inputs, the number of outputs and the number of product terms. Since the exact personality matrix is not available during the estimation phase, the PLA delay estimation is characterized by three cases, namely the best, the average and the worst cases.

[3]The geometries of the registers can be obtained from the module library directly.

The PLA delay estimation technique is taken from Gupta [Gup91] in which the PLA charging delay t_{total_ch} and discharging delay t_{total_dis} are estimated as

$$t_{total_ch} = t_{inv1_ch} + t_{inv2_dis} + t_{pt_ch} + t_{op_dis} + t_{inv3_ch} \quad (5.13)$$

$$t_{total_dis} = t_{inv1_dis} + t_{inv2_ch} + t_{pt_dis} + t_{op_ch} + t_{inv3_dis} \quad (5.14)$$

where t_{inv1_ch}/t_{inv1_dis} and t_{inv2_ch}/t_{inv2_dis} are input inverter charging/discharging delays, t_{pt_ch}/t_{pt_dis} is the product term stage charging/discharging delay, t_{op_ch}/t_{op_dis} is the output stage charging/discharging delay and t_{inv3_ch}/t_{inv_dis} is the final inverter charging/discharging delay.

An RC model, which is similar to the approach in wiring delays, is used to characterize the PLA delay. Each term in the delay equations is estimated in two steps. The capacitances and resistances are first estimated based on the proposed empirical models. Three types of PLA delay estimations were used [Gup91], namely

- *Best-Best* Case: Only one product term is connected to the output and all the inputs are driving the product term (minimum load capacitance and maximum driving capability).

- *Average-Average* Case: Only half of the product terms are connected to the output and only half of the inputs are driving the product terms (average load capacitance and average driving capability).

- *Worst-Worst* Case: All product terms are connected to the output and only one input is driving the product term (maximum load capacitance and minimum driving capability).

Once the capacitances and resistances are known, each term in the PLA delay equations can then be estimated by Equation 5.8 or Equation 5.9. The estimated results are also verified by spice simulation results. More details can be found in [Gup91].

5.3 Floorplanning Technique

A floorplan contains information about cell geometries as well as cell locations. Floorplanning is beneficial in providing an accurate estimate of the area and delay of circuits. The accurate estimate of area and delay characteristics of a circuit makes an early feedback loop in a high-level synthesis system possible; it also helps the high-level synthesis process prune infeasible designs at an earlier synthesis stage.

A typical floorplanning approach first determines the floorplan topology using the connectivity information among modules/cells. Various optimizations are then performed in order to minimize the specified cost function (e.g. area and/or performance). The general floorplanning problem has been proven to be NP-complete, so heuristic methods are used. In our approach, we use the cluster tree structure, which was also adopted by Dai [DK87], to represent the module topology. Figure 5.7 shows the flow chart of the floorplanning process. For the sake of efficiency, we limit the cluster size to four. However, algorithms presented here can be extended to a cluster size of five or larger.

Once the cluster tree is determined, polynomial time optimal algorithms for a slicing cluster tree [Sto83] or a non-slicing cluster tree [WW90] can be applied to obtain floorplans with different aspect ratios. Figures 5.8 and 5.9 show two additional floorplans with different aspect ratio goals for the 10-time-step non-pipelined FIR filter design shown in Figure 3.6.

We define our terminology in the following. A *k-room floorplan pattern* is a floorplan structure with exactly k rooms where a *room* is an enclosed region. An *orientation* of a pattern is a clockwise rotation of the pattern with respect to its boundary. A *labelling* of a pattern corresponds to an assignment of nodes of the cluster tree to individual rooms. A *topological possibility* refers to a particular choice of floorplan pattern, pattern labelling and pattern orientation [Ped91]. A 3-room floorplan example which illustrates the definitions is shown in Figure 5.10.

A *primitive* corresponds to a functionally non-divisible module, such as an adder, a multiplexer or a register. A *cluster node* is a non-primitive node in the cluster tree. The *shape function* for a node in the cluster tree gives the lower bound

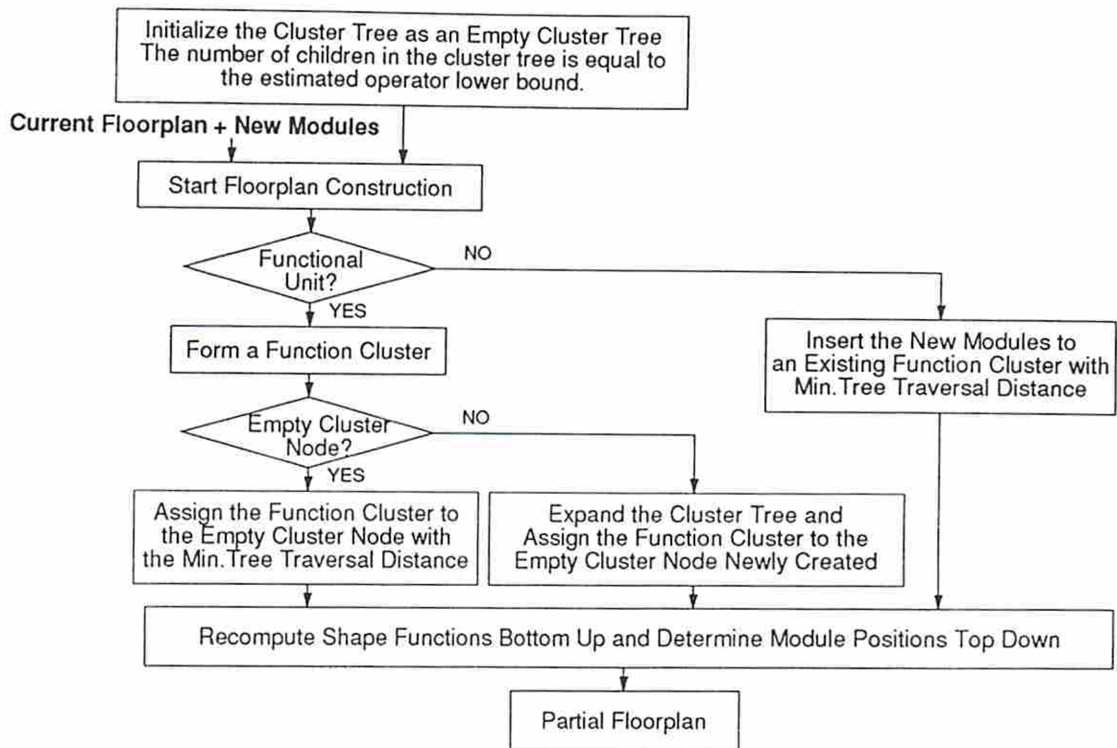


Figure 5.7: The Flow Chart of the Floorplanning Process

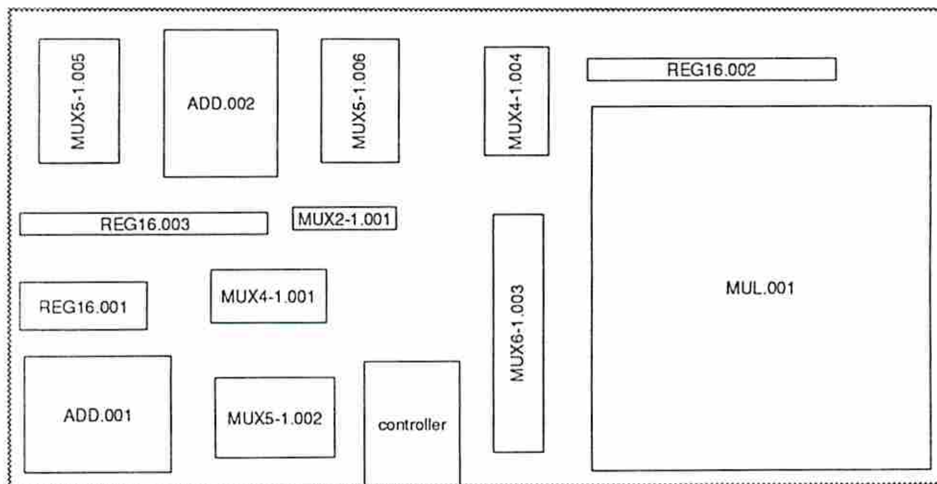
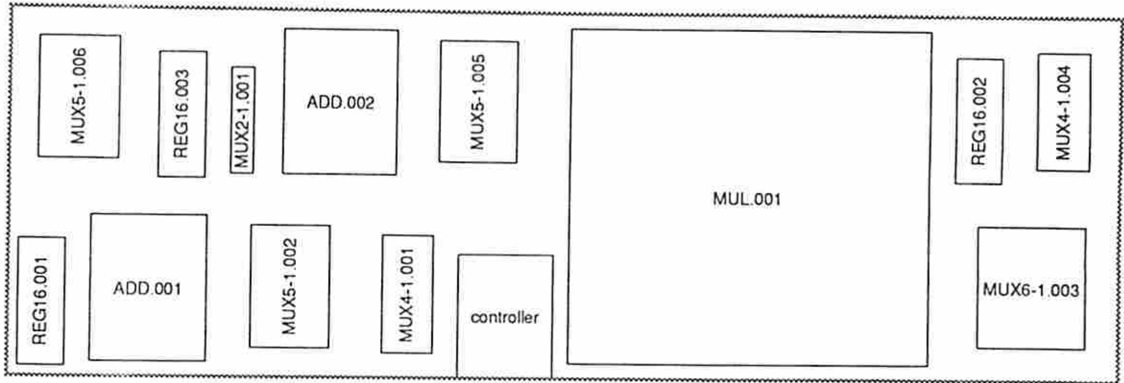


Figure 5.8: FIR Filter Floorplan with Overall Aspect Ratio Goal 2:1



Size: 4314.09 um x 1414.75 um, Area: 6103358.61 um^2

Figure 5.9: FIR Filter Floorplan with Overall Aspect Ratio Goal 3:1

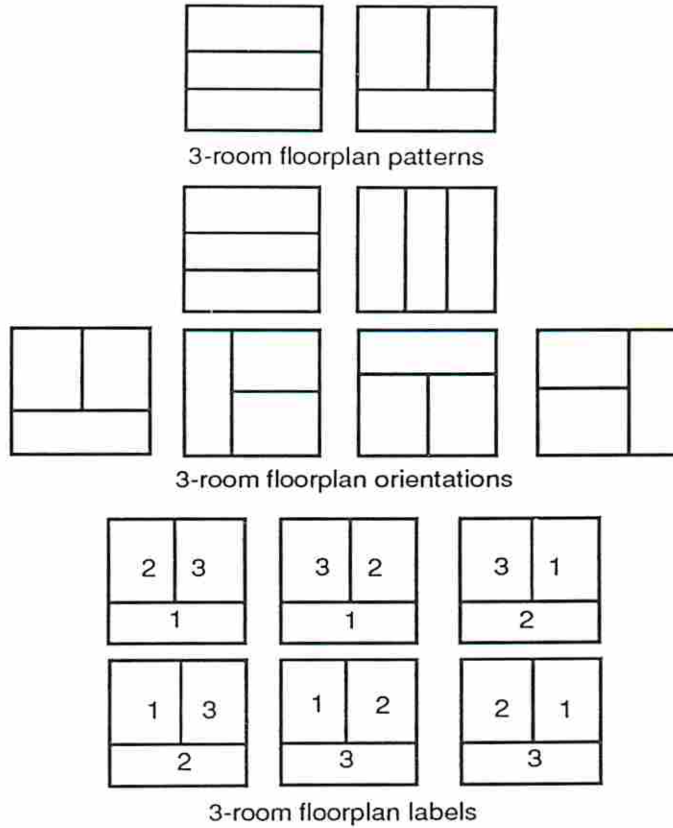


Figure 5.10: 3-Room Floorplan Patterns, Orientations and Some Possible Labelings

on the height of the node as a function of its width. Note that a shape function is a discrete function; each discrete value in a shape function is called a *shape point*. The shape function of a primitive, which is a collection of variable shape points, is given in the input module library. The shape function of a cluster node is computed by merging shape functions of its children [Ott83] [Zim88] [Ped91].

5.3.1 Incremental Cluster Tree Generation

Hierarchy is a typical way to deal with the complexity of a large problem. In the context of floorplanning, the hierarchy usually is represented by a cluster tree where highly connected primitives are grouped together. This hierarchical scheme simplifies the floorplanning problem, since floorplanning algorithms can recursively operate on one hierarchical child (which may be a primitive or a cluster node) at a time. The tree itself may have a restricted structure (e.g. binary slicing with fixed cut directions and cluster-to-room labelings) or a flexible structure (e.g. a multi-way unoriented cluster tree). A more flexible structure allows a higher degree of floorplan optimization. A multi-way unoriented cluster tree is used in our floorplanning procedure.

The cluster tree approach allows us to consider more floorplan topologies than that which is represented by either horizontal or vertical cuts. The maximum branching factor in the cluster tree is restricted to a small value (four in our approach). The complexity of floorplanning a cluster node is the same as that of the general floorplanning problem, if the branching factor is too large. The non-slicing floorplan is avoided in our approach, also for efficiency reasons. However, the procedures presented here can be applied to non-slicing floorplans as well as slicing floorplans.

In order to incorporate the floorplanning process into the scheduling process, a cluster tree is incrementally constructed as follows. A new cluster node is added to the cluster tree whenever a new functional unit is allocated during the scheduling process. The main shortcoming of this incremental approach is its greedy, sequential nature which can not account for the global connectivity information. However, this drawback is somewhat alleviated by utilizing prediction tools. The prediction results are helpful since they allow us to overview the design before

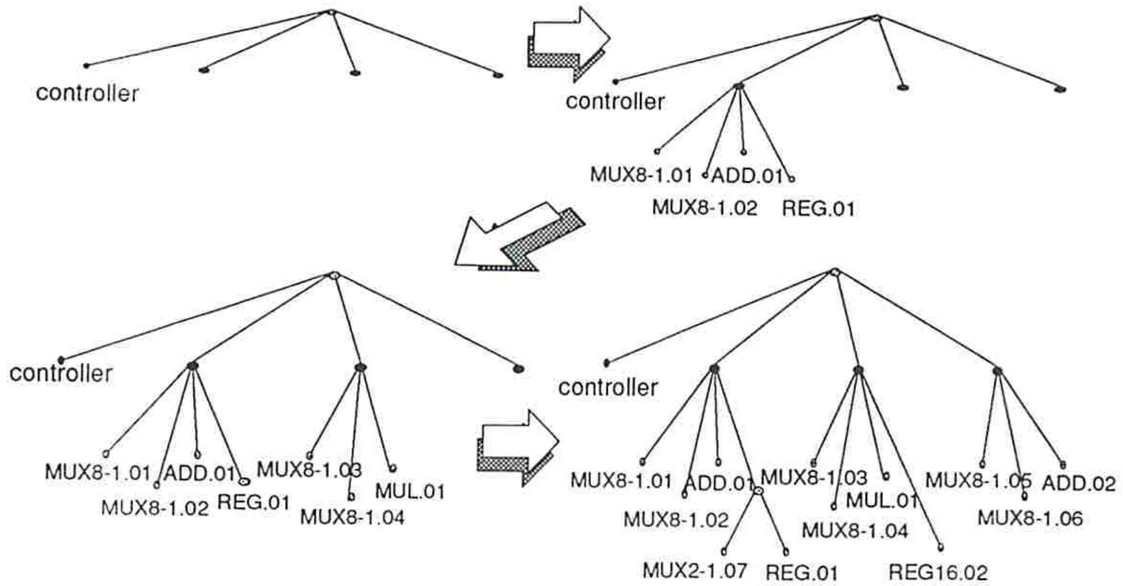


Figure 5.11: The Construction of a Cluster Tree

scheduling is performed (or completed). They are also beneficial in making better trade-off decisions with a more global view at early stages of the synthesis process. For example, in contrast to using a greedy approach to operation binding, the estimation operator allows us to select operation binding among all possible operators. Also, estimating wiring area of a module allows us to consider the wiring effects during scheduling and floorplanning.

The proposed clustering strategies are based on the characteristics of register-transfer level design architecture and functionality. A *function cluster*, consisting of a functional unit, possible multiplexers (which directly connected to input/output terminals of the functional unit) and a possible register (which stores the values generated by the functional unit) is used in our scheme. The cluster tree shown in Figure 5.11, for example, has four function clusters: one function cluster for the controller, two function clusters for the adders and one function cluster for the multiplier. Function clusters are shown by solid dots in the figure. The function cluster itself may have a two-level hierarchy, such as the function cluster **add.01**. Function clusters represent "natural" partitioning of a register-transfer level design as is intuitive because modules inside a function cluster are tightly connected and thus should be placed together in order to reduce the wiring area and delay.

In the data path synthesis process, each allocated functional unit corresponds to a function cluster in the cluster tree. The process of constructing a cluster tree contains the following steps. The cluster tree contains empty *spots* at the beginning of the scheduling process. The number of empty *spots* n initially allocated to the cluster tree is equal to the number of estimated operator lower bound. In determining the topology of a cluster tree with n empty *spots*, a cluster tree of height l is used whose leaf nodes are of level 0 and the root node is of level l [AHU74]. The cluster tree with n empty *spots* ($n > 1$) is determined as

$$n_{i+1} = \lceil \frac{n_i}{m} \rceil \qquad 0 \leq i < l \tag{5.15}$$

where m is the cluster size, n_i and n_{i+1} are the number of nodes of level i and $i+1$, respectively. Note that $n_0 = n$ and $n_l = 1$. The number of children $m_{i,j}$ of a node in level i is determined as

$$m_{i,j} = \begin{cases} \lceil \frac{n_{i-1}}{n_i} \rceil & j = 1 \text{ and } i > 0 \\ \lceil \frac{n_{i-1} - \sum_{k=1}^{j-1} m_{i,k}}{n_i - (j-1)} \rceil & 1 < j \leq n_i \text{ and } i > 0 \end{cases}$$

The next step is to determine the assignment of a newly allocated functional unit (i.e. a function cluster) to an empty *spot* in the cluster tree. The location of a newly constructed function cluster is based on its connectivity to already placed function clusters in the cluster tree. The number of edges on the path connecting two *spots* is used to represent the interconnection cost between the two function clusters placed in these two *spots*. The interconnection cost for assigning a newly formed function cluster to some candidate empty *spots* is thus calculated as the minimal number of edges between the candidate *spot* to occupied *spots* which are connected to the new function cluster. Note that the interconnection cost between an unscheduled predecessor[4] and the currently scheduled operation is not considered and assumed to be zero in our approach. The *spot* with the minimum interconnection cost is selected as the location for the function cluster. A new *spot* is added to the floorplan cluster tree only when all *spots* in the initial floorplan

[4]Let node A and node B be two operations in the data flow graph. Node A is a predecessor of node B if any input of node B comes from the output of node A.

cluster tree are occupied. Once all the functional units are allocated and placed, the construction of the floorplan cluster tree is complete.

For illustration purposes, the floorplan cluster tree construction process for a 10-time-step non-pipelined FIR filter design is shown in Figure 5.11. A cluster tree with four empty spots is created initially using the results of operator lower-bound estimation. The next allocated function module is **add.01**; the corresponded function cluster is formed and placed to the cluster node next to the controller. **mul.01** and **add.02** are then allocated and placed. The function cluster **add.01** is modified after it is placed. The multiplexer **mux2-1.07** is inserted between **add.01** and **reg.01** in the register-transfer level design due to the sharing of **reg.01**. Note that the function cluster **add.01** now has two levels of hierarchy because of the limitation of cluster size.

5.3.2 Shape Function Computation

Once the cluster tree is constructed, the floorplanner needs to determine the exact topological relationships of the cluster nodes as well as the geometries of the primitives. To obtain this information, the floorplanner first performs a bottom-up traversal to the cluster tree to compute the shape functions for the cluster nodes. A top-down traversal then follows in order to determine the topological relationships of cluster nodes and the geometries of primitives to specific shape points.

In [Sto83] and [Ott83], Stockmeyer and Otten showed how to combine two shape functions due to a horizontal cut or a vertical cut. Briefly, the procedure proposed by Stockmeyer is similar to merging two sorted lists. The procedure merging two shape functions, $\mathcal{S}'(h', w') = \{(h'_1, w'_1), (h'_2, w'_2), \ldots, (h'_p, w'_p)\}$ and $\mathcal{S}^*(h^*, w^*) = \{(h^*_1, w^*_1), (h^*_2, w^*_2), \ldots, (h^*_q, w^*_q)\}$, to a new shape function $\mathcal{S}(h, w)$ for a vertical cut is summarized in Figure 5.12.

Procedure 2

> /* *Merge two shape functions* $\mathcal{S}'(h', w')$ *and* $\mathcal{S}^*(h^*, w^*)$ *due to a V cut.* */
> Initialize $i \leftarrow 1$, $j \leftarrow 1$.
> **for** $i \leq p$ **or** $j \leq q$ **do**
> **begin**

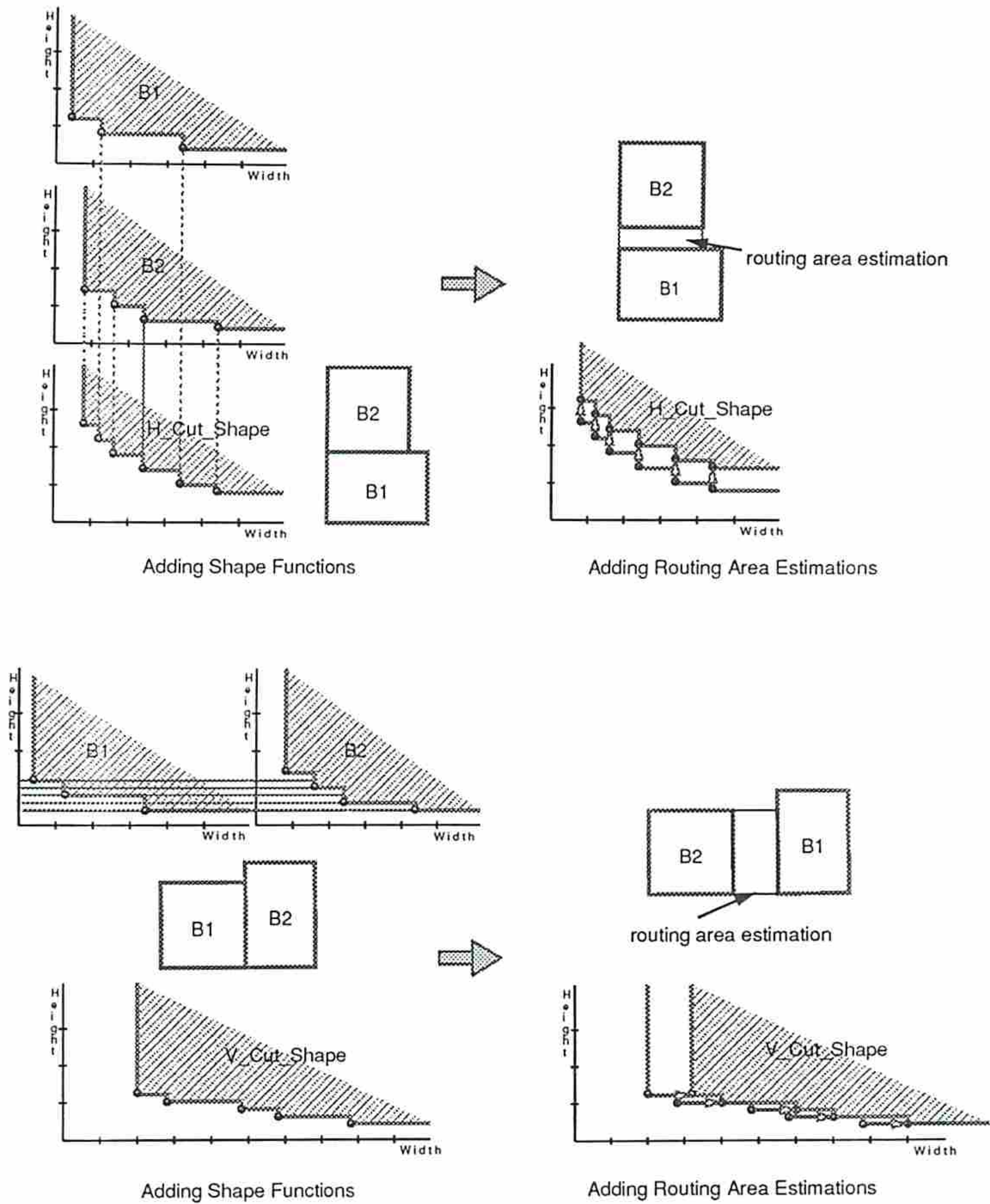


Figure 5.12: Shape Function Addition for Horizontal and Vertical Cuts

Add $\mathcal{J}((h'_i, w'_i), (h^*_j, w^*_j))$ to shape function $\mathcal{S}(h, w)$.

if $h'_i > h^*_j$ **then**

$i \leftarrow i + 1$; **continue.**

if $h'_i < h^*_j$ **then**

$j \leftarrow j + 1$; **continue.**

if $h'_i = h^*_j$ **then**

$i \leftarrow i + 1$; $j \leftarrow j = 1$; **continue.**

end

□

$\mathcal{J}((h'_i, w'_i), (h^*_j, w^*_j))$ is a join function which puts two shape points, (h'_i, w'_i) and (h^*_j, w^*_j), together:

$$\mathcal{J}((h'_i, w'_i), (h^*_j, w^*_j)) = (\max(h'_i, h^*_j), w'_i + w^*_j) \tag{5.16}$$

A key fact is that we do not have to consider all $p \cdot q$ new pairs, since many of them are clearly suboptimal. A theorem proved by Stockmeyer [Sto83] and Otten [Ott83] shows only $p + q$ pairs need to be considered.

Zimmerman extended the shape function computation procedure for a binary cut with unspecified cut orientation [Zim88]. Briefly, to obtain the merged shape function for an unspecified orientation cut, the shape functions corresponding to both horizontal and vertical cuts are calculated. The resulting shape function is derived by merging the shape points on the lower bound of the two shape functions. The procedure is illustrated in Figure 5.13.

The shape function computation for a binary cut with unspecified cut orientation was further extended by Dai to calculate the shape function for an unoriented multi-way cluster tree [DK87]. For a cluster node with k children, the shape function can be determined as the following. Assuming that the cluster tree corresponds to a slicing structure,[5] we can always further decompose the cluster tree to a binary composition tree without specifying cut orientations. For a binary

[5] For a topological relationship that corresponds to a non-slicing structure, the shape function can also be calculated similarly. For example, the calculation of the shape function of a non-slicing structure with 5 children was published in [WW90].

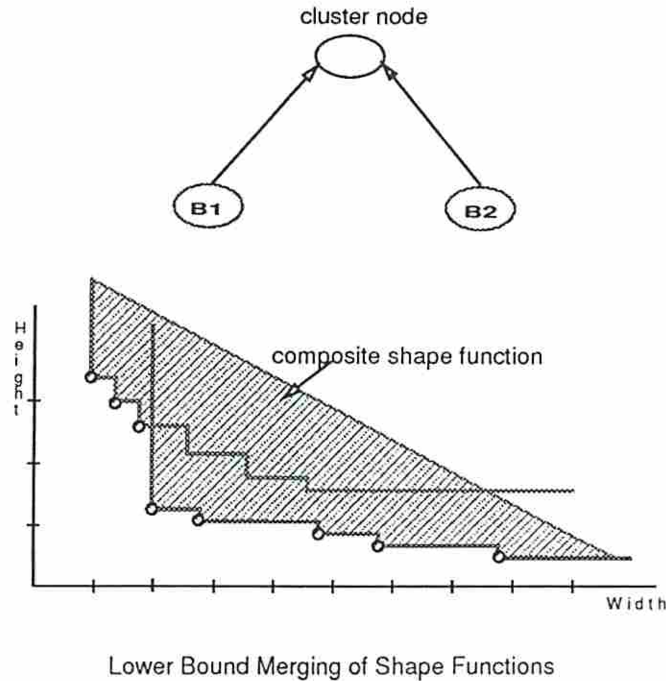


Figure 5.13: Merging of Shape Functions

decomposition tree, there exists an unique canonical representation, such as the *Polish expression* [WL86]. Note that the leaves of the binary decomposition tree correspond to the children of the cluster node and the internal nodes of the binary decomposition tree correspond to the unspecified orientation cut nodes in a binary tree structure. The combined shape function for the root of this cluster tree, which is first decomposed to a binary tree with unspecified cut orientation, can therefore be calculated in the same manner as the procedure described for a binary tree with unspecified cut orientation.

In the cluster tree generation phase, a multi-way cluster tree is created without determining the topological relationships of cluster nodes. The shape function for an unoriented multi-way cluster node is first computed and the topological relationships of cluster nodes are then determined. The wiring area is estimated for each child in the cluster node before the shape function is calculated [Ped91]. (This is in contrast to Zimmerman's approach [Zim88] which calculates the shape function for each child without estimating the wiring area first and then shifts

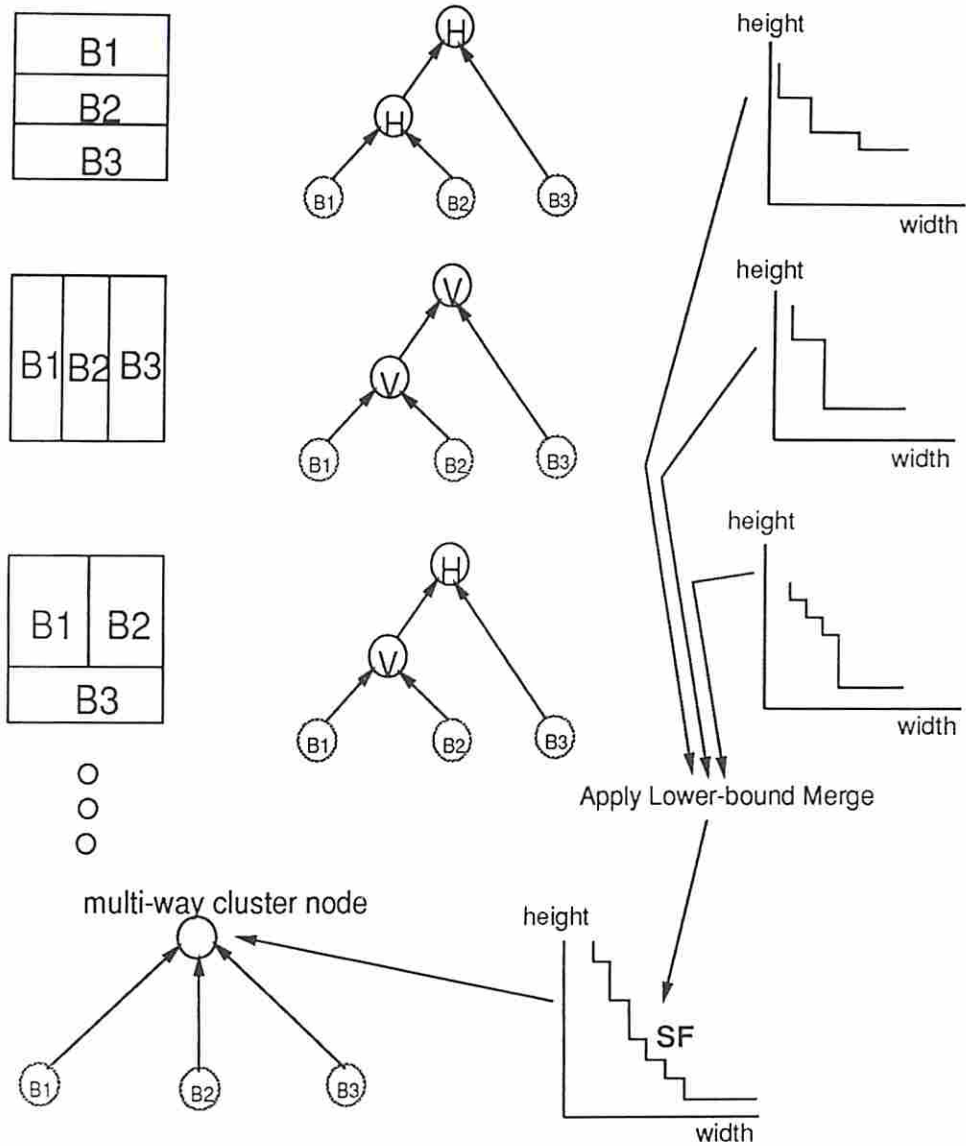


Figure 5.14: Calculation of the Shape Function for a Cluster Node with 3 Children

the whole shape function in order to account for the wiring area.) Next, the lower bound of merging all the calculated shape functions is taken to obtain the shape function for the unoriented multi-way cluster node. Figure 5.14 depicts the calculation of shape function for an unoriented multi-way cluster node. Note that the number of shape points of a cluster node are linearly reduced to a limited number for efficiency and to reduce complexity. For example, twenty-five to thirty shape points for a cluster node are usually sufficient to produce a fairly smooth shape function based on our experiments.

This procedure is recursively applied up the cluster tree until the composite shape function for the root node is calculated. This bottom-up calculated shape function is used to generate a feasible floorplan by propagating the associated geometries of the floorplan solution, which corresponds to a shape point in the shape function, to the leaf nodes. As long as the cost function is non-decreasing in all of its arguments, the above procedure minimizes the floorplan with respect to the cost function [Sto83].

5.4 3D Scheduling Technique

In this section, the 3D scheduling algorithm is presented. We first discuss some assumptions which were made by the 3D scheduling algorithm. Loops are considered in the 3D scheduling algorithm, however, designs with nested loops are handled but not parallel loops. The 3D scheduling algorithm is then presented. After presenting the complexity analysis of the 3D scheduling, we finally justify the use of force-directed scheduling and incremental cluster tree techniques.

5.4.1 Timing Model Assumptions

As stated before, a two phase non-overlapped clocking scheme is assumed: a control-path clock followed by a data-path clock. More precisely, the delay through a control path consists of the wiring delay of the values generated by the data path and then fed back to the PLA circuit (if any) plus the internal delay of the PLA circuit, and the wiring delays of control signals to the data path. The data path execution always starts from register outputs, multiplexer or external inputs and

ends at registers or external outputs. Several assumptions on timing are made in the calculation of control path and data path delays:

1. The external input values are ready (stable) to be fetched by the data path at the beginning of the respective data path execution cycle. These values are assumed to be latched by an external circuit. This assumption was made to simplify the problem and could be relaxed without changing the basic approach used here.

2. The external output values are produced and latched by the data path until the data path execution is completed. The delay introduced by the output latch circuit is considered.

3. The data path execution paths start from either external inputs or registers and terminate at either external outputs or registers.

4. The condition values (internal values fed back to the controller) produced by the data path are created at least one clock cycle before they are used by the control path.

5. The delay for transferring a value directly between two data-path registers (with no operations in between) is less than the longest operation delay in the data path.

Note that the external input values are latched by additional logic until the values are no longer needed by the data path. The additional logic may be created by a binding/allocation program, such as MABAL [KP90]. The values created by the data path are latched by the data-path registers. Since synchronous circuits are the only kind being considered here, values that are produced in the data path, but not "consumed" in the time step generated (i.e. the value will be used by some operations in the later time steps), are stored in registers to avoid possible race conditions. This register requirement is consistent with the third assumption about the data path execution paths. The fourth assumption made about condition values is due to the timing dependencies between the control path and data path.

The assumption concerning the delay of transferring a value between two data-path registers is important. In a circuit design, the designer may want to share

hardware modules which may result in some values being created but not consumed in the following execution cycle. In that case, a value it may become necessary to move from one register to another during the data path execution cycle. (This situation frequently happens in a pipelined design.) Since data-path registers are only partially allocated and the floorplan is incrementally produced, the data-path registers allocated by the 3D scheduling process are floorplanned as close to the operators as possible. The data-path registers allocated by an allocation and binding program (like MABAL) are placed at the boundary of the partial floorplan produced by the 3D scheduling process. The assumption, that the delay of transferring a value directly between any two data-path registers is less than that of the longest delay of the data path, allows us to give higher priority to binding the values just produced in the current data path execution cycle to the near of registers without violating the clock cycle constraint (i.e. degrading system performance).

5.4.2 Loop Model

For designs with loops, the 3D scheduling algorithm considers only nested loops but not parallel loops.[6] Figure 5.15 shows the representation of a loop in our DDS translator [Che91]. A block in the data flow graph model represents a set of operations. A loop structure in our approach consists of a *loop entrance*, a *loop condition* and a *loop body*. The loop feedback values are implicitly specified and represented by the subscripts. The loop condition determines the execution of the loop body. α and ω points are used to represent loop structures in our approach. An α point is a point with one incoming edge and two outcoming edges. An ω point, in contrast, has two incoming edges and one outcoming edge. A loop structure in a DDS timing graph always begins at an α point and ends at an ω point. The loop condition is represented by a pair of *OrFork* and *OrJoin* nodes in the timing graph. The left edge of the *OrFork* point always has one time range to which the loop exit is bound. The right side of the *OrFork* usually has several consecutive

[6]Parallel loops are loops without data dependencies in between which could be executed in parallel.

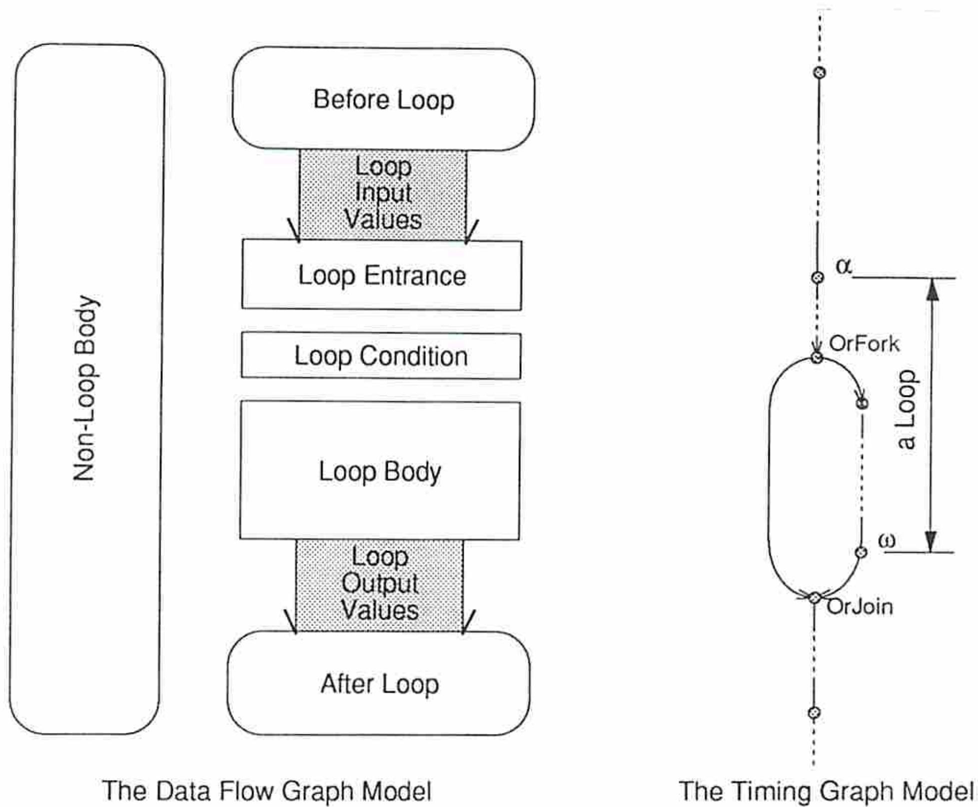


Figure 5.15: The Loop Model

ranges to which the loop-body operations are bound. Note that a pseudo/dummy edge (or range) is added between the ω point and the *OrJoin* point to make the timing graph connected.

Due to timing considerations in implementing a future controller for a design with loops, *break nodes* are introduced in the 3D scheduling algorithm. A break node is a pseudo node in the data flow graph that is inserted in the beginning of the loop structure and right after the loop body. The break nodes ensure that the 3D scheduler assigns different time steps to the operations outside the loop structure and the operations inside the loop structure. After the break nodes are inserted, the data flow graph can be scheduled in the same way as a design without loops. Operations outside the loop which are parallel to the loop structure are scheduled concurrently with the operations in the loop structure. However, these operations are executed in the first instance of loop structure execution; the resources to which the operations outside the loop structure are bound are idle until the execution of

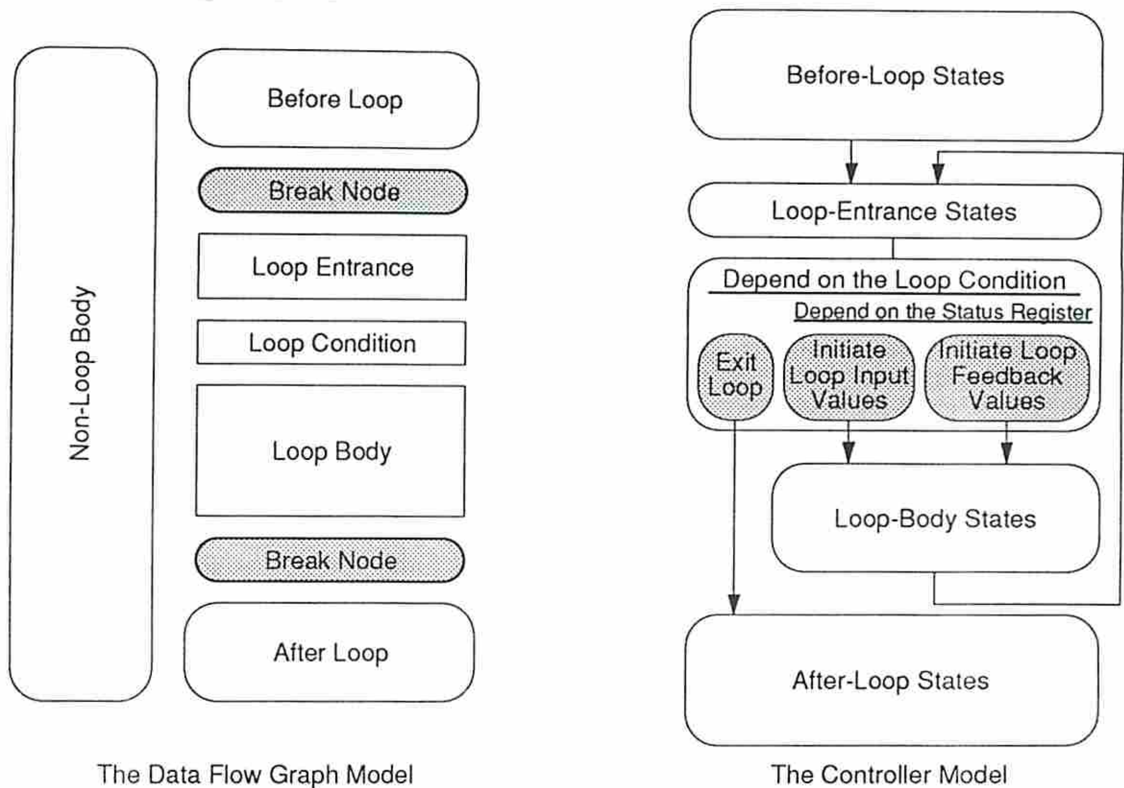


Figure 5.16: The Proposed Approach for Loop Designs

the loop structure is completed. Since the number of loop iterations is sometimes non-deterministic, only area constraints can be accounted for to the 3D scheduler for designs with loops.

Figure 5.16 shows the insertion of break nodes into the data flow graph and the proposed controller model. The states that activate the control signals for the operations outside the loop structure are derived the same manner as the designs without loops. The states corresponding to the operations in the loop structure are classified into three groups, namely *loop-entrance states*, a *loop-condition state* and *loop-body states*. A status register is used here to indicate the execution status of the loop structure. Three possibilities are found in the *loop-condition state*:

- If the loop condition is false, the controller exits the loop and jumps to the first state of after-loop states.

- If the loop condition is true and the loop has not been executed before, the controller initiates the loop input values and executes the operations inside the loop structure.

- If the loop condition is true and the loop has been executed before, the controller initiates the loop feedback values and continuously executes the operations inside the loop structure.

Note that the status register is set to *not-visited* status by the controller before entering the first state of *loop-entrance states*. The status register is set to *visited* status in the last state of *loop-body states*. The operations outside the loop structure are executed only when the status register is in *not-visited* status.

5.4.3 3D Scheduling Algorithm

The basic idea of 3D scheduling is the following: edges in a data flow graph are annotated to represent the interconnection delays in the register-transfer network. In the beginning of the scheduling process, edges are initialized with the estimated wiring delays and possible multiplexer delays. As an operation is scheduled, the module assignment is performed according to the cost function. A new module is allocated and floorplanned if no allocated module is available or allocating a new module incurs less cost. The cost function is a combination of the distribution sum, instance cost, connection compatibility and register cost. The user specifies relative weights among these factors. We will explain these factors more in the later of this section. The scheduler then does the register assignment if the time steps of the newly scheduled operation and its scheduled successors are different. The scheduler also inserts the necessary multiplexers between functional modules and registers. The effects of wiring and allocated registers and multiplexers are considered in the floorplanning. The wiring delays obtained from the partial floorplan are then fed back to adjust the interconnection delays in the data flow graph, which will affect the ASAP, ALAP and distribution graphs used by the 3D scheduler in the next scheduling iteration.

Figure 5.17 shows the flow chart of the 3D scheduling algorithm. A cluster tree is constructed at the beginning of the scheduling process, using the results of

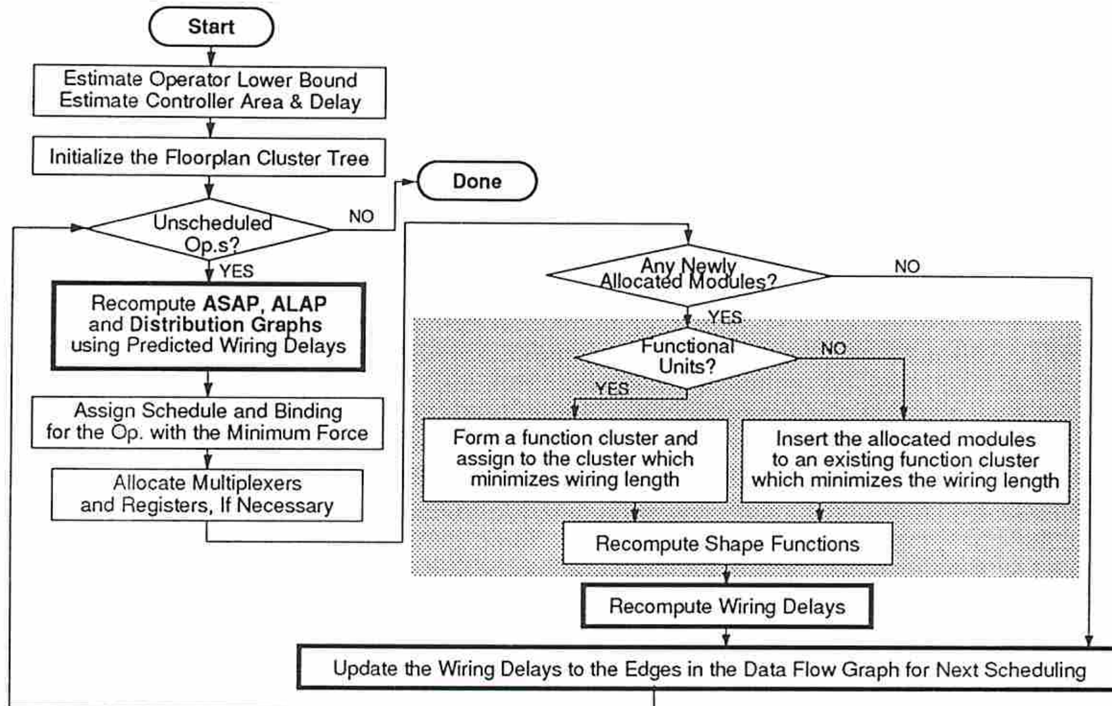


Figure 5.17: The Flow Chart of 3D Scheduling Algorithm

the operator lower-bound estimation. The 3D scheduling algorithm then performs scheduling, module allocation and module binding for the operations in the data flow graph. This is achieved by extending force-directed scheduling to include the individual module instances in the force function [CT90]. The blocks with solid lines in the flow chart show the effects of the wiring delays fed back from the partial floorplan to the 3D scheduler. The blocks covered by the hatch patterns show the floorplanning process. The detail floorplanning process has been shown in Figure 5.7.

The 3D scheduling algorithm takes the same two dimensional view of scheduling and binding that Park presented [PP88]. Operations in a data flow graph are assigned in a two dimensional allocation table, one dimension defined by the time steps and the other by instances of hardware modules. Figure 5.18 shows the allocation table for the 4-time-step FIR filter example (the data flow graph is shown in Figure 5.1). Each small rectangle in the table is called a *cell*. A *cell* is a logical unit which represents the possible binding of an operation to the module in a certain time step. By using the allocation table, the 3D scheduling algorithm

Figure 5.18: The Allocation Table for the 4-Time-Step FIR Filter Example

is able to consider time steps as well as module instances during scheduling. The potential for assigning an operation to each instance of a hardware module is the self force which is calculated as

$$
\begin{aligned}
& self\ force(op, cs_range) = \\
& \quad \sum_{j \in new_range} \{\Delta prob(op, j) \times springK(op, j, instance(j))\} - \\
& \quad \sum_{j \notin new_range} \{prob(op, j) \times springK(op, j, instance(j))\}
\end{aligned}
$$

where $cs_range(op)$ is a period of time steps for which the force is calculated (see Section 5.1 for the definitions), $range(op)$ is the set of cells into which the op can be scheduled and binded, $new_range(op)$ is the set of cells that are within cs_range, and $instance(cell)$ projects the cell to the associated module instance. $prob(op, j)$ is the probability that op will be assigned to cell j. Without lose of generality, equal probabilities are assumed for all cells in a range. Thus $prob(op, j)$ is equal to $1/|range(op)|$. $\Delta prob(op, j)$ is the change of probability associated with op and cell j when a $range$ is reduced to include only the new cells in the new_range.

For example, in Figures 5.1 and 5.18, the current $range$ of $addb$ covers 2 cells (4,7) so each has a probability of 1/2. However, when $addb$ is limited to time

step 3 (i.e. *cs_range*), the *new_range* covers one cell (7) only and the $\Delta prob$ is $1 - 1/2 = 1/2$.

The $springK$ is made up of four terms:

$$
\begin{aligned}
&springK(operation, cell, instance) = \\
&\quad \alpha \times distribution\ sum(operation, cell) + \\
&\quad \beta \times instance\ cost(instance) + \\
&\quad \gamma \times connection\ compatibility(operation, instance) + \\
&\quad \delta \times register\ cost(operation, instance)
\end{aligned}
$$

where α, β, γ and δ are user-specified constants which allow the users to change the relative importance of the four terms. The *distribution sum* term is a measure of how strongly the operation should be assigned to the cell. The distribution sum is equal to the value of the distribution graph in the time step which the cell occupies. A larger distribution sum means more functional modules are required in this time step. The *instance cost* term is used to represent the allocation cost of the tentative schedule. The area of the functional module with the minimal area is chosen as the unit-instance area. There is no instance cost if the number of functional modules allocated is less than the number of functional modules predicted. Otherwise, the area of the newly allocated functional module normalized to the unit-instance area is used as the instance cost.

The *connection compatibility* term is a measure of how similar the connection requirements of the operation match the existing connections. The estimated wiring area in establishing the connection paths is used to reflect the potential connection cost. The connection paths considered are the paths from the predecessors of the operation to the instance (i.e. a functional module) and the paths from the instance to the successors of the operation. There is no connection cost if a connection path already exists. To estimate the connection cost, the wiring area is approximated by the estimated wire length multiplied by the track width, which is a parameter depending on the technology. For a scheduled predecessor/successor (i.e. it is bound to a placed functional module) and a placed instance, the wire

length is estimated by the half perimeter of the bounding box of these two functional modules. Notice that the instance may not be placed yet if it is a newly allocated functional module.

For an unscheduled predecessor/successor or an unplaced instance, an effective wire length is used to estimate the wire length. The effective wire length $\mathcal{L}_e$ is estimated as

$$\mathcal{L}_e = \sqrt{\mathcal{A}_{ef} + (n_{if} \cdot \mathcal{A}_{em})} \tag{5.17}$$

$\mathcal{A}_{ef}$ is the effective area of the unplaced functional module (see Equation 5.7 for the definition of effective area), n_{if} is the number of inputs of the unplaced functional module and $\mathcal{A}_{em}$ is the estimated area of the input multiplexer. The number of the multiplexer inputs is estimated by $\lceil \frac{n_k}{o_k} \rceil$, where n_k is the number of operations whose type is the same as the unplaced functional module in the data flow graph and o_k is the number of operators predicted.

For the case that the predecessor/successor has been scheduled but the instance has not been placed, the wire length for the connection cost prediction $\mathcal{L}_w$ is estimated as

$$\mathcal{L}_w = \mathcal{W}_{es} + \mathcal{H}_{es} + \mathcal{L}_{ei} \tag{5.18}$$

where $\mathcal{W}_{es}$ and $\mathcal{H}_{es}$ are the effective width and height of the placed predecessor/successor, respectively, and $\mathcal{L}_{ei}$ is the effective wire length of the unplaced instance.

For the case that the predecessor/successor has not been scheduled but the instance has been placed, the wire length for the connection cost prediction $\mathcal{L}_w$ is estimated as

$$\mathcal{L}_w = \mathcal{L}_{es} + \mathcal{W}_{ei} + \mathcal{H}_{ei} \tag{5.19}$$

where $\mathcal{L}_{es}$ is the effective wire length of the unscheduled predecessor/successor, $\mathcal{W}_{ei}$ and $\mathcal{H}_{ei}$ are the effective width and height of the placed instance, respectively.

For the case that neither the predecessor/successor nor the instance have been scheduled and placed, the wire length for the connection cost prediction $\mathcal{L}_w$ is then estimated as

$$\mathcal{L}_w = \max(2\mathcal{L}_{es} + \mathcal{L}_{ei}, \mathcal{L}_{es} + 2\mathcal{L}_{ei}) \tag{5.20}$$

The sum of the estimated connection cost of the connection paths is finally normalized to the area of a 2-to-1 multiplexer to obtain the value of the connection compatibility term. Note that the allocated modules should not be shared if the connection cost is larger than the instance cost.

The *register cost* term reflects the register requirements of the operation under the tentative schedule. The register cost of a scheduled operation is dependent on the lifetime of the values produced. In our approach, we consider the register cost of an operation whose successors are scheduled. For an operation whose successors are not scheduled, the register cost is assumed to be zero. There is no register cost if a value is produced and consumed in the same time step (i.e. the value is produced inside an operator chain). Otherwise, the register cost is equal to the number of time steps between the time step produced and time step last being used (i.e. the lifetime of this value).

The total force is the sum of self force and indirected force. The indirected force is calculated the same manner as Paulin's approach (Equation 5.2). Once the total force has been used to schedule an operation, the 3D scheduling algorithm must then decide on its operator binding (and value binding, if necessary). In the 3D scheduling algorithm, the instance and interconnection costs are included in the terms of $springK$; the instance with minimum $springK$ in the scheduled time step determines the binding of the scheduled operation.

In the 3D scheduling process, each allocated functional unit corresponds to a function cluster in the cluster tree. A new function cluster is created if a functional unit is newly allocated. The position of a newly constructed function cluster in the cluster tree is determined by its interconnections to other placed function clusters. Multiplexers and registers are allocated and added into the cluster tree when the existing interconnections are not adequate. A partial floorplan is then obtained by computing the shape functions. The wiring delays derived from the floorplan are

fed back to the scheduling process to guide the next scheduling iteration. Once all the operations in the control/data flow graph are scheduled, the scheduling process is completed.

5.4.4 Complexity Analysis of the 3D Scheduling Algorithm

The run-time complexity of the 3D scheduling algorithm depends on

- the complexity of the operator lower-bound estimation,
- the complexity of the force calculation, and
- the complexity of the floorplanning procedure.

The complexity of calculating the distribution graph is $\mathcal{O}(n)$ where n is the number of the operations being scheduled in the data flow graph. Since the tighter operator lower-bound estimation procedure reduces the size of the maximally consecutive distribution window every iteration, the distribution graph is calculated $\mathcal{O}(c)$ times at most to obtain the operator lower bound, where c is the number of time steps into which the data flow graph is being scheduled. Therefore, the complexity of the operator lower bound estimation is $\mathcal{O}(cn)$.

In the calculation of the force, $\mathcal{O}(ci)$ forces are calculated for an unscheduled operation where i is the number of functional modules being allocated in the design. To determine the operation with minimum force, the forces of $\mathcal{O}(n)$ unscheduled operations are calculated every scheduling iteration. Since there are $\mathcal{O}(n)$ unscheduled operations at the beginning of the scheduling process, the complexity of the force calculation is $\mathcal{O}(cn^2i)$. This is i times the complexity of the force-directed scheduling used by Paulin. The additional i term is caused by the fact that all the possible instances for an operation are examined for every force calculation.

The floorplanning procedure consists of a bottom-up traversal and a top-down traversal. Because the cluster tree is balanced and the number of shape points and children in a cluster node are limited, the complexity of the bottom-up traversal is $\mathcal{O}(i)$. Since the process is repeated for every functional module allocated, the

complexity of the top-down traversal is also $\mathcal{O}(i)$. The complexity of the floorplanning procedure is $\mathcal{O}(i^2)$ due to $\mathcal{O}(i)$ functional modules allocated in the scheduling process. The overall run-time complexity of the 3D scheduling algorithm is $\mathcal{O}(cn + cn^2i + i^2)$ which can be simplified to $\mathcal{O}(cn^2i)$.

5.4.5 Discussion

From our experiments, we found that the scheduling order of force-directed scheduling is not fixed. The operations along the critical path are not always scheduled earlier than the operations on off-critical paths. The quality of the constructive cluster tree is affected by the order of new modules inserted. And, the order of constructing the floorplan cluster tree depends on the order of the operations being scheduled and bound. Therefore, a scheduling technique which schedules the operations along the critical path first is expected to produce better floorplan cluster trees.

The freedom-based scheduling technique is an example which always schedules the operations along the critical path (i.e. the operations with smaller freedom [PPM86]) first. However, a good decision on inserting dummy operations along the critical path in order to produce more serialized designs is not easy to obtain. A modified force-directed scheduling technique which partially changes the scheduling order to schedule the operations along the critical path first may be a solution in our application. A scheduling approach which combines the freedom-based approach and force-directed approach is another possible solution to this problem.

The floorplanning approach is greedy. However, most information in register-transfer level is unknown during scheduling. Since a cluster tree represents the floorplan in our approach, the number of edges traversed is used to measure the distance between two function clusters. This is simply heuristic and not well reflecting connection costs in the floorplan. A new heuristic is needed in the feature development.

Chapter 6

Data Path Synthesis Experiments and Results

This chapter describes several experiments performed using the data path synthesis program, LADS (Layout And Data path Synthesis). We compared the schedules created by LADS to those created by traditional scheduling programs/techniques, such as MAHA and Force-Directed Scheduling. Various filter designs and a robot arm controller were synthesized to experiment with the 3D scheduling technique under different design cases.

6.1 Introduction

The 3D scheduling algorithm for data path synthesis was implemented in a program called LADS using the C language. LADS performs scheduling, operation bindings and partial value bindings for a VHDL input according to user-specified constraints. A library contains the characteristics of available modules (such as delays, areas and geometries), fabrication-dependent parameters (such as transistor sizes, resistances and capacitances) and other adjustment parameters to guide the scheduling process as well as the floorplanning process. Table 6.1 shows the module library we used in our data path experiments.

In the data path experiments, we used LADS to generate schedules and bindings. Some examples were further synthesized by MABAL to produce register-transfer level designs. CSG took the outputs from MABAL to generate controller specifications. The register-transfer level designs and controller specifications were

1.2μm Technology	*Delay (ns)*	*Area (μm^2)*
16-bit Carry Select Adder	25.0	279264.00
16-bit Ripple Adder	35.0	81557.50
16-bit Ripple Subtractor	38.0	113529.25
16-bit Comparator	33.5	134380.50
16-bit Multiplier	53.0	1804522.00

Table 6.1: Design Library Set Used by LADS (Obtained from ChipCrafter)

compiled by the Cascade Design Automation ChipCrafter silicon compiler to obtain layouts.

The schedules produced by MAHA and the Force-Directed Scheduling technique were used as examples to compare the differences among schedules, register-transfer level designs and final chip layouts. We also used the timing analyzer to compare and validate the delays estimated by LADS.

We tested LADS with a number of examples. The filter examples included a FIR filter, an AR filter and an Elliptic filter. A robot arm controller served as a non-filter example. Both pipelined and non-pipelined design schedules were also experimented with for most of the designs.

6.2 A Non-Pipelined FIR Filter Example

For the FIR filter non-pipelined design experiments, we used a 10-time-step design as our example. We used **carry select adders** for all additions in this experiment. The schedule created by LADS is shown in Figure 6.1. We used MABAL to synthesize the register-transfer level design using the schedule (shown in Figure 6.2). and bindings (shown in Appendix A) produced by LADS. The register-transfer level design was then compiled by ChipCrafter to obtain the layouts (see Figures A.1-A.5 in Appendix A). The solid lines on the layouts are the critical paths obtained from the timing analysis program within ChipCrafter. The figures are scaled in order to reflect the relative dimensions among the layouts.

Table 6.2 summarizes the results of the layouts produced by ChipCrafter. *LADS Prediction* shows the prediction results produced by LADS at the scheduling stage. As compared the prediction results to the actual layouts, the estimated

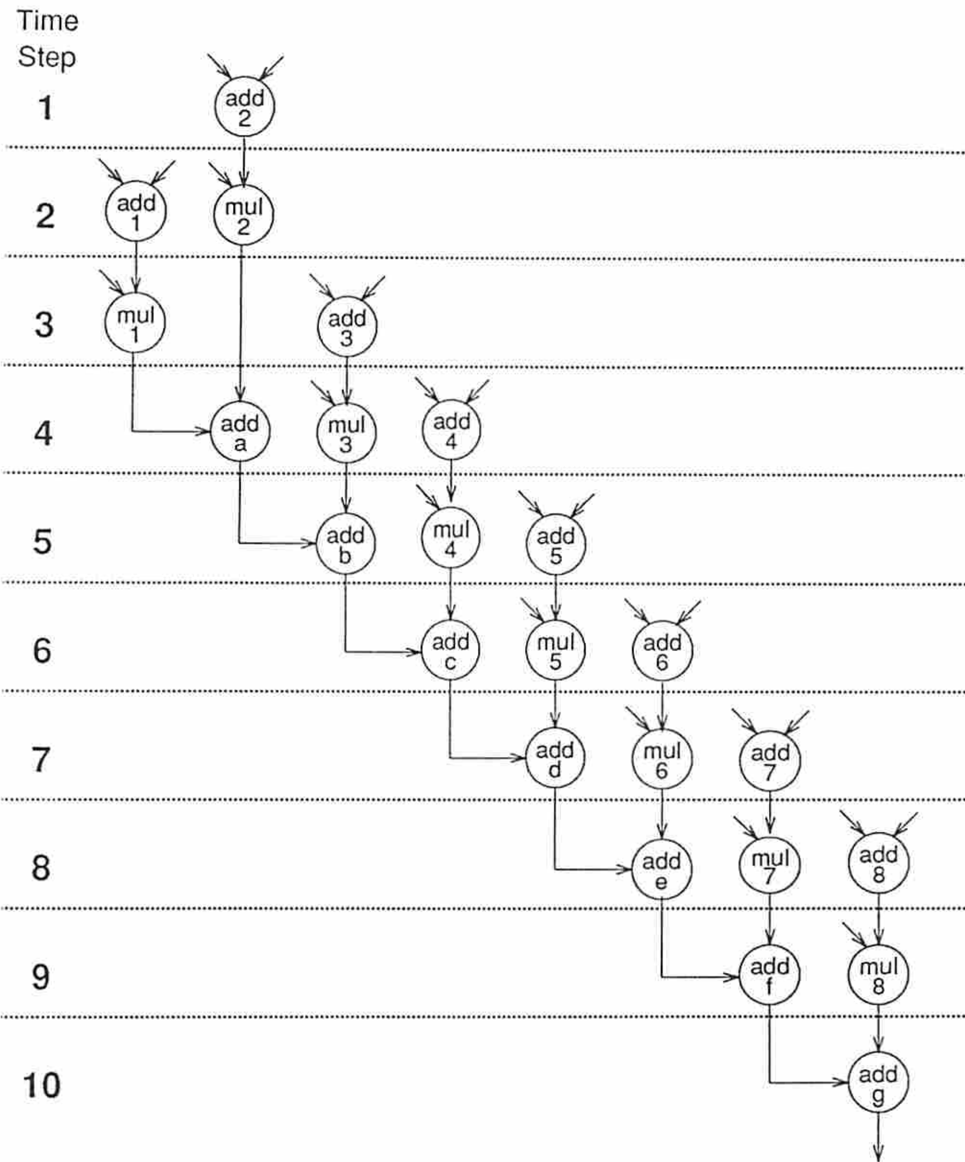


Figure 6.1: The 10-Time-Step FIR Filter Schedule Produced by LADS

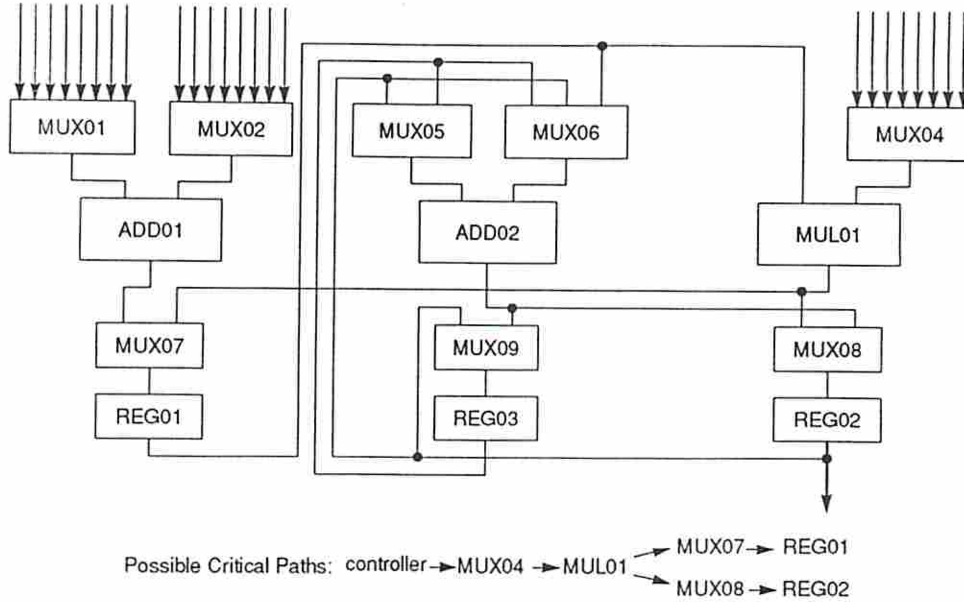
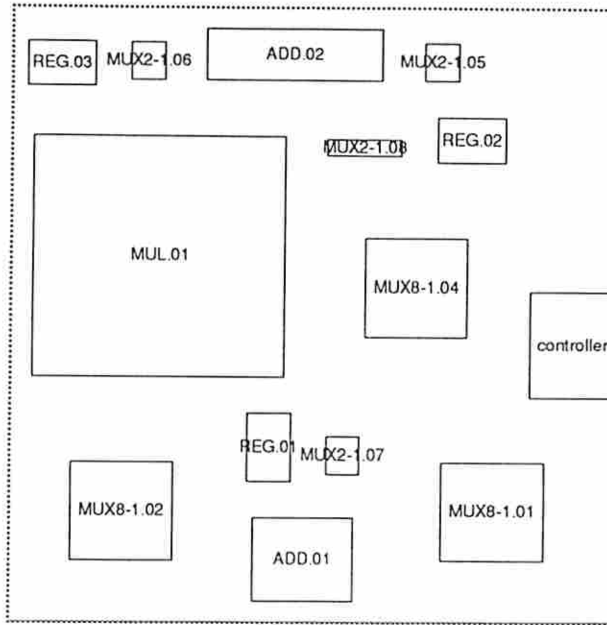


Figure 6.2: The RTL 10-Time-Step FIR Filter Design using the LADS Schedule

area and aspect ratio are favorable in this experiment. However, the estimated data path delay is longer than the actual data path delay. This may be caused by using the worst-case wire length and ignoring the effects of buffers in estimating wiring delays. The estimated control path delay is also longer than the actual controller delay. This may be due to the merging of duplicate outputs in the PLA optimization phase which is not considered in the current PLA estimation program.

The placement program in ChipCrafter uses a simulated annealing algorithm. Figures A.1-A.3, *ChipCrafter Placement I*, *ChipCrafter Placement II* and *ChipCrafter Placement III*, show the layouts using LADS schedule but with different cooling schedules in the placement program. In order to compare design characteristics of the layouts using the placements produced by LADS to ChipCrafter, we created a layout using the placement produced by LADS. Figure 6.3 shows the floorplan created by LADS. Since LADS only performs partial value bindings, extra registers and/or multiplexers are expected to appear in the final register-level design created by MABAL. Figure 6.4 shows the floorplan for our final implementation which is obtained by manually modifying the floorplan created by LADS



NPD-10 (est. D.P. delay 96.10, C.P. delay 51.56)
Size: 3320.94 x 3329.75, Area: 11057903.83 (A.R. 1.00)

Figure 6.3: The Floorplan created by LADS

to include modules added by MABAL. Cross hatched patterns indicate newly allocated modules or modules which have been modified. The *std_group* is a set of standard cells that are produced by the finite state machine synthesis program within ChipCrafter.

Figure A.4 shows the actual layout of *LADS Placement I* using the placement of the modified LADS floorplan shown in Figure 6.4. In this experiment, we found that the data path and control path delays using the LADS placement are reduced from those in the ChipCrafter placements. The critical paths shown in Figures A.1-A.4 reveals that the floorplan produced by LADS is able to reduce the delays along the critical path. Note that two possible critical paths (i.e. MUX04 → MUL01 → MUX07 → REG01 and MUX04 → MUL01 → MUX08 → REG02) are found in the register-transfer level design shown in Figure 6.2. The simulation critical paths are not identical in these layouts.

The area of the LADS layout is comparable to *ChipCrafter Placement I* and *ChipCrafter Placement III*, however, it is larger than *ChipCrafter Placement II.*

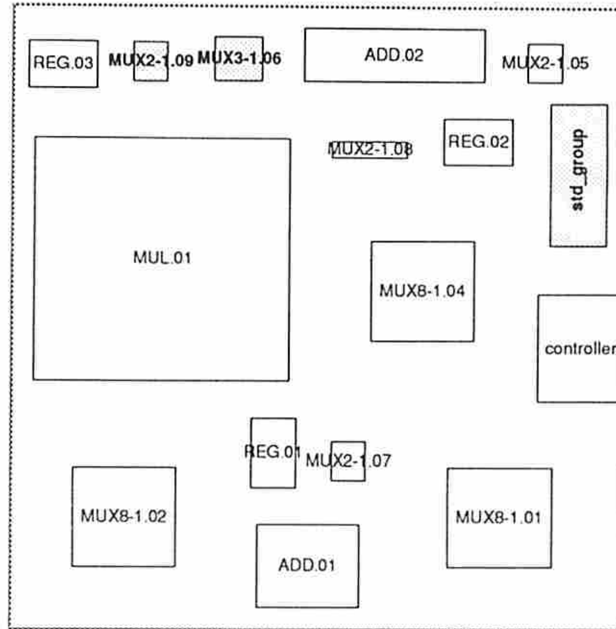


Figure 6.4: The Final Implementation Floorplan using the LADS Schedule

By inspecting the layout shown in Figure A.4, we found a large area was a result of wiring. We believe the excessive amount of wiring area results from the misplacement of pins and may be reduced through proper pin assignments [PMSK90] [Con89]. We thus manually performed pin assignment to *LADS Placement I* to validate our idea. Figure A.5 shows the layout of *LADS Placement II* which uses the same placement as *LADS Placement I* but with manual pin assignment. The result is encouraging; the chip area is reduced dramatically (about 30%) which is consistent to the one predicted in pin assignment papers. Note that the performance is slightly improved in this particular case due to the reduce of chip area. The simulation critical path in the layout with manual pin assignment is also changed.

To compare the schedule results among different scheduling techniques, we applied the same functional unit area constraint to scheduling programs using different techniques. The schedule produced by a Force-Directed Scheduling (FDS) program is shown in Figure 6.5. The schedule is further synthesized by MABAL to obtain the register-level design as shown in Figure 6.6. Note even though the

allocated functional units are the same as the LADS design, the schedule and allocated interconnection units (e.g. multiplexers and registers) are different.

The results of the layouts using FDS schedule is shown in Table 6.3. The layouts and simulation critical paths are shown in Figures A.6-A.10. The delays of the layouts using FDS schedule increase about 25% as compared to the results of the LADS schedule. The performance degradation in this schedule is due to the chaining of two adders in some time steps which in turn changes the critical path from the multiplier (in the LADS schedule) to the two adders (e.g. MUX12 $\rightarrow$ ADD01 $\rightarrow$ MUX21 $\rightarrow$ ADD02 $\rightarrow$ MUX71 $\rightarrow$ REG04 for the layout shown in Figure A.6).

We also manually created a performance-driven floorplan (see Figure 6.7) to reduce the delays along the critical paths. We used this placement to produce a timing-driven layout, *Manual Placement I*. The experiment shows that the performance is better than two of ChipCrafter's placement cases but slightly worse than one of ChipCrafter's placements. For comparison purposes, we also performed pin assignment manually, as shown in *Manual Placement II*, to reduce wiring area. There is not much improvement in this example. We believe this is due to the more complicated interconnections between modules; a good pin assignment was not possible to obtain manually in this example.

Similar experiments were performed using MAHA and MABAL. The schedule and register-transfer level design are shown in Figures 6.8 and 6.9, respectively. The timing-driven floorplan, which is derived manually, is shown in Figure 6.10. The layouts and simulation critical paths are shown in Figures A.11-A.15. The experimental results are summarized in Table 6.4. We found that in the designs using the MAHA schedule, the delays of data path and control path are slightly worse than the layouts using the Force-Directed Scheduling technique (about 10-15% increase). This is probably due to the more complex interconnections and larger wiring area requirements in this register-transfer level design.

To compare different scheduling techniques and layouts, a set of charts were drawn. Figure 6.11 compares the area, data path delay, control path delay and total execution delay among the three different ChipCrafter placements and three different scheduling techniques. It can be seen from these charts that different cooling schedules of the simulated annealing placement algorithm cause different

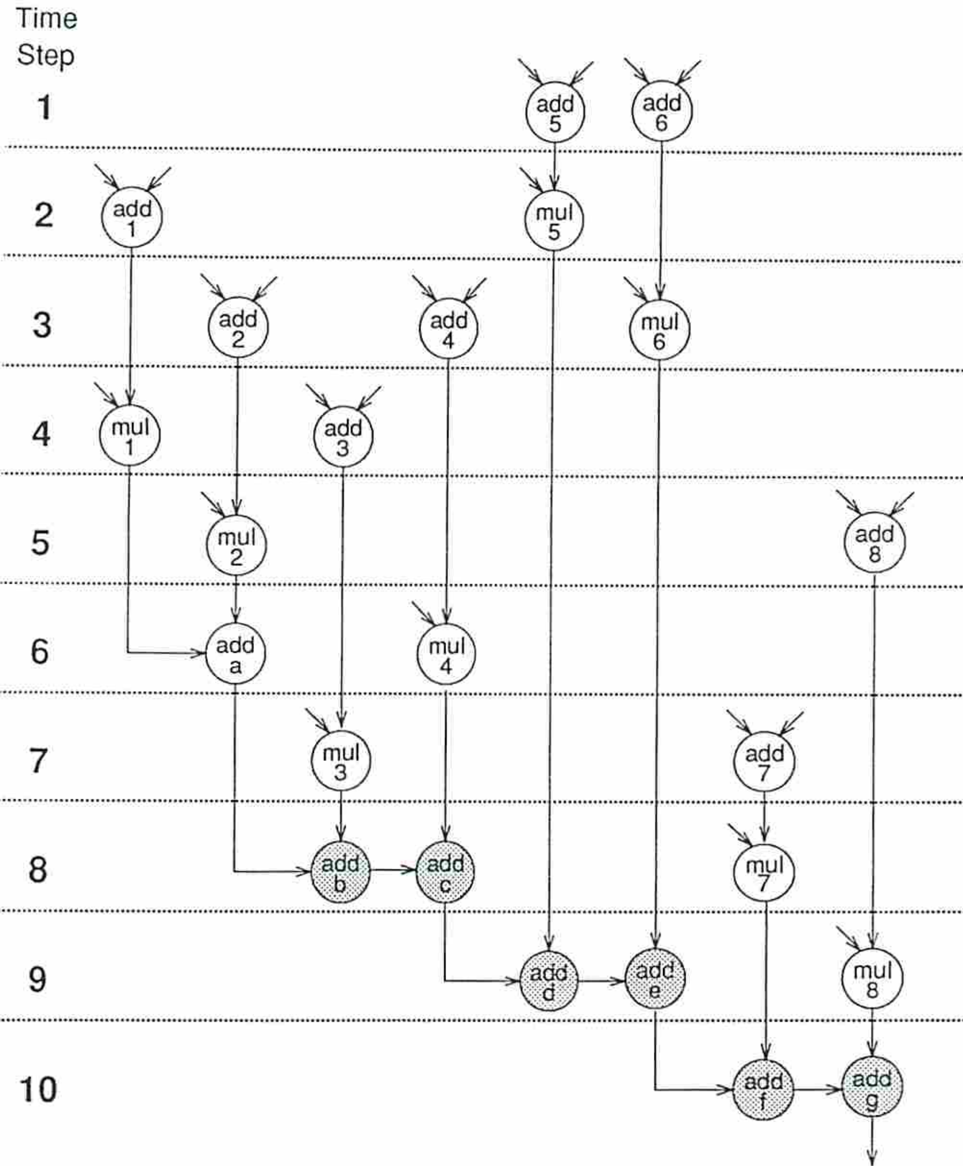


Figure 6.5: The 10-Time-Step FIR Filter Design Produced by FDS

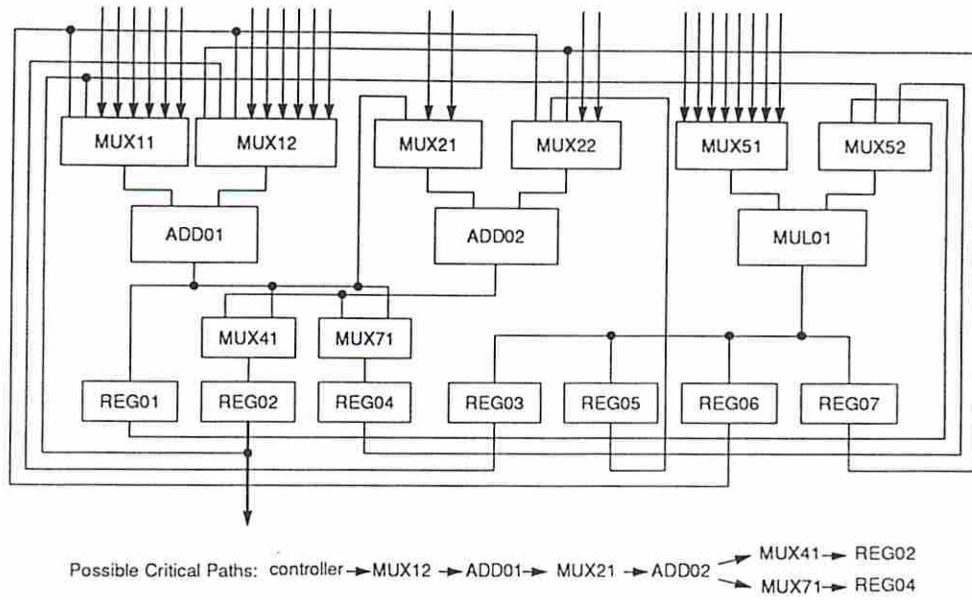


Figure 6.6: The RTL 10-Time-Step FIR Filter Design using the FDS Schedule

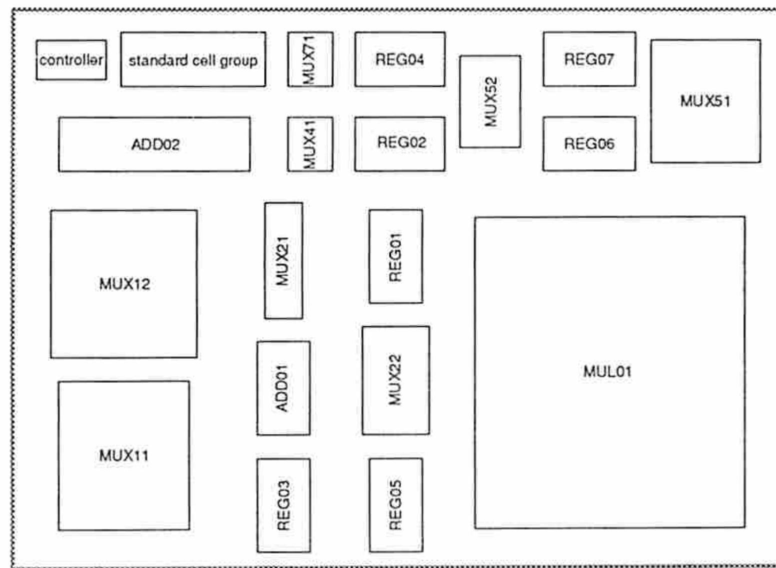


Figure 6.7: A Manually Created Timing-Driven Floorplan using the FDS Schedule

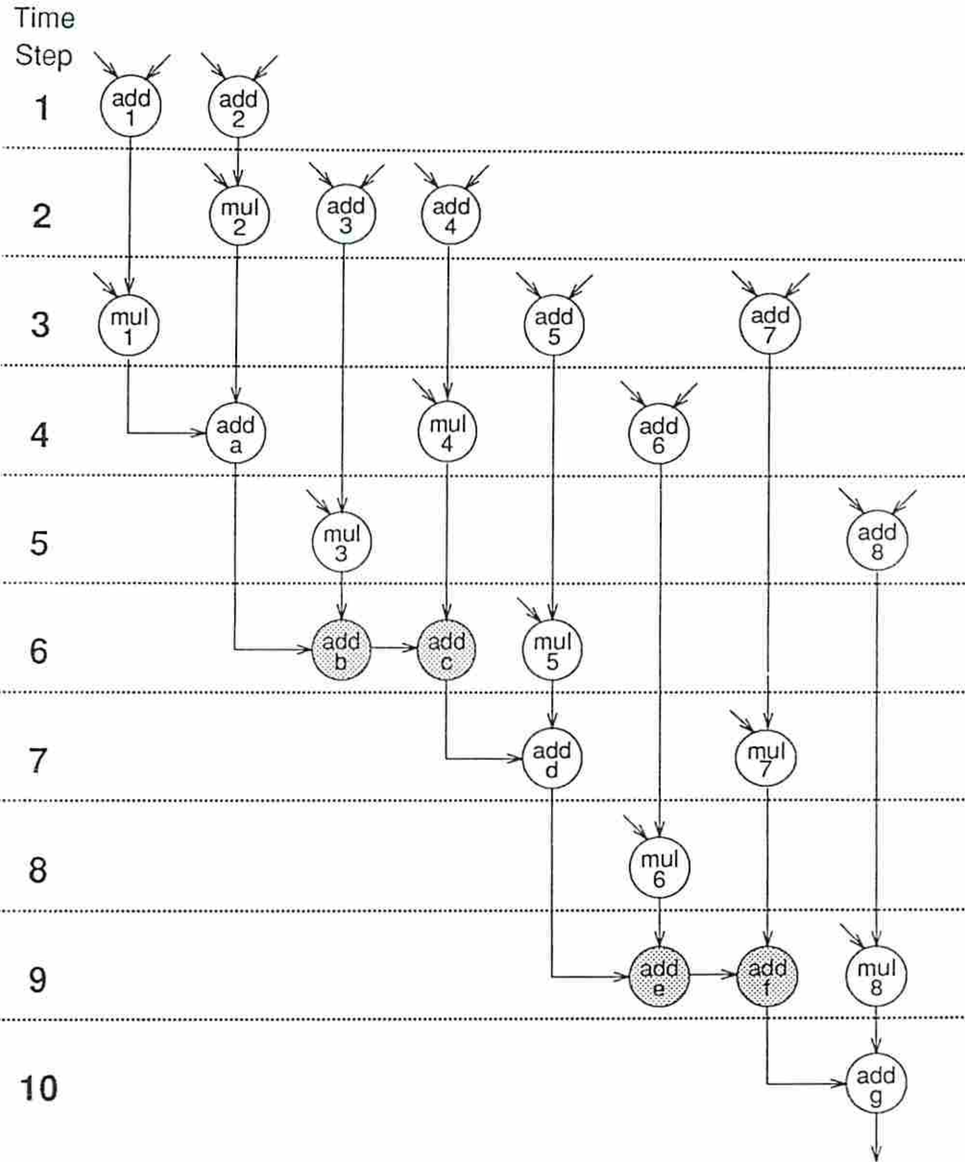


Figure 6.8: The 10-Time-Step FIR Filter Schedule Produced by MAHA

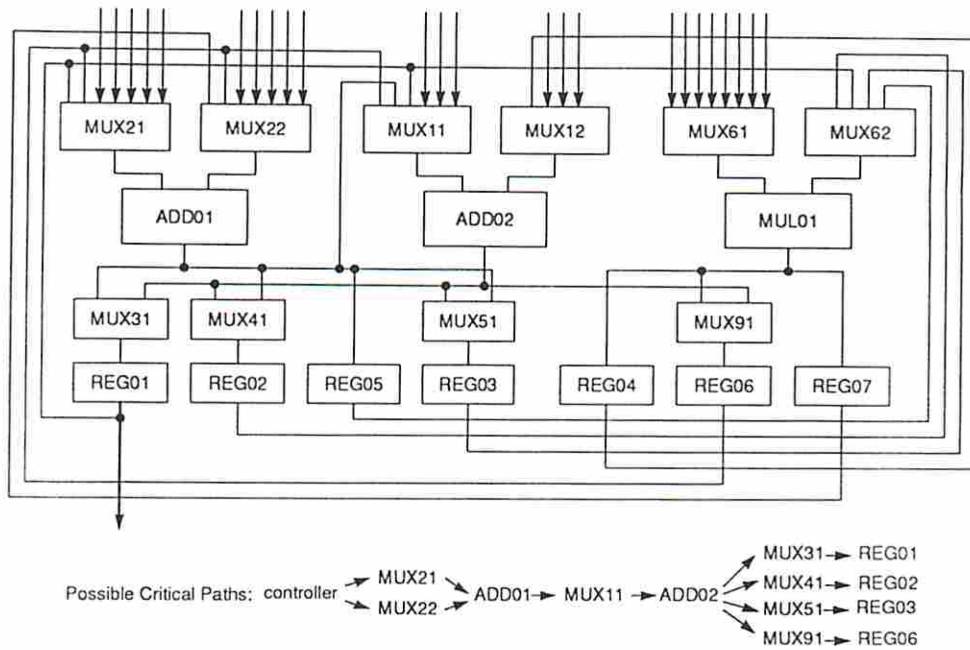


Figure 6.9: The RTL 10-Time-Step FIR Filter Design using the MAHA Schedule

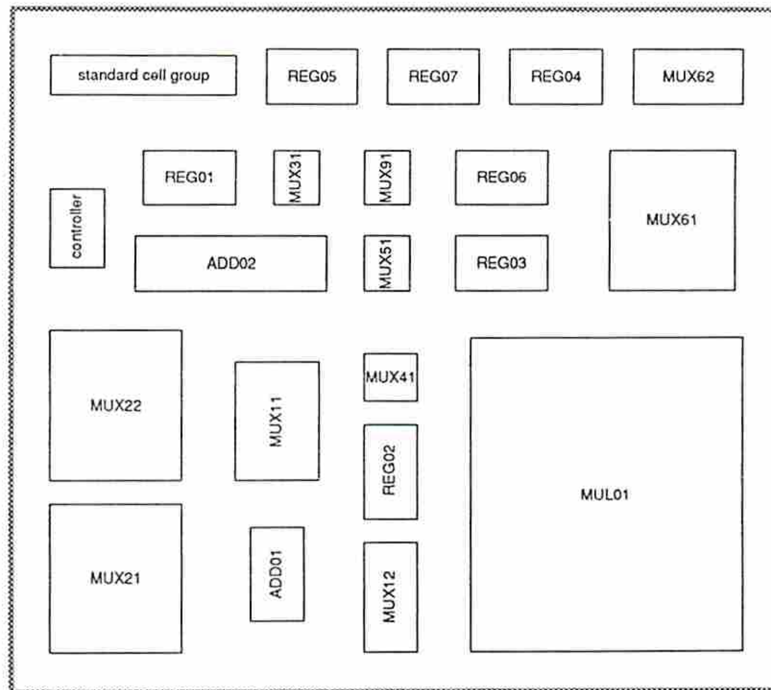


Figure 6.10: A Manually Created Timing-Driven Floorplan using the MAHA Schedule

Description	*Dimensions* ($\mu m \times \mu m$)	*Area* (μm^2)
LADS Prediction	3320.94 × 3329.75	11057903.83
ChipCrafter Placement I	3696.75 × 4961.79	18342497.18
ChipCrafter Placement II	3506.25 × 3312.04	11612840.25
ChipCrafter Placement III	3001.75 × 4753.79	14269689.13
LADS Placement I	3642.75 × 3873.00	14108370.75
LADS Placement II	3089.50 × 3533.00	10915203.50

Description	*D.P. Delay*†	*C.P. Delay*‡	*Exec. Delay*§
LADS Prediction	96.10 ns	51.56 ns	147.66 ns
ChipCrafter Placement I	77.13 ns	22.92 ns	100.05 ns
ChipCrafter Placement II	78.25 ns	19.58 ns	97.83 ns
ChipCrafter Placement III	75.86 ns	20.13 ns	95.99 ns
LADS Placement I	73.18 ns	16.72 ns	89.90 ns
LADS Placement II	71.48 ns	16.74 ns	88.22 ns

Table 6.2: The 10-Time-Step Non-Pipelined FIR Filter Designs using the LADS Schedule

variations in the design characteristics. However, the variations among different cooling schedules are far less than the variations among schedules produced by different scheduling techniques. Therefore, we can conjecture that the design performance in our system is more dominated by schedules than placements.

We also compared the placement generated by LADS to the ones using the simulated annealing algorithm (i.e. the placements created by ChipCrafter). Figures 6.12 shows the comparisons of the area, data path delay, control path delay and execution delay with different scheduling techniques. The *LADS/Manual Placement I* and *LADS/Manual Placement II* of the designs using the LADS schedule represent the layouts using *LADS Placement* without and with manual pin assignment, respectively. The *LADS/Manual Placement I* and *LADS/Manual Placement II* of the designs using the FDS/MAHA schedule represent the layouts using the performance-driven placement shown in Figure 6.7/6.10 without and with pin assignment, respectively. Since there are three layouts whose

†The D.P. delay represents data path delay.

‡The C.P. delay represents control path delay.

§The execution delay includes the delays in the data path and the control path.

Description	*Dimensions* ($\mu m \times \mu m$)	*Area* (μm^2)
ChipCrafter Placement I	3089.50 × 3533.00	10915203.50
ChipCrafter Placement II	3628.50 × 5592.29	20291624.27
ChipCrafter Placement III	3953.75 × 3830.33	15144167.24
Manual Placement I	4298.00 × 2917.00	12537266.00
Manual Placement II	4103.75 × 2939.25	12061947.19

Description	*D.P. Delay*	*C.P. Delay*	*Exec. Delay*
ChipCrafter Placement I	91.77 ns	33.38 ns	125.15 ns
ChipCrafter Placement II	103.58 ns	35.29 ns	138.87 ns
ChipCrafter Placement III	93.79 ns	32.43 ns	126.22 ns
Manual Placement I	89.94 ns	32.27 ns	122.21 ns
Manual Placement II	92.17 ns	32.41 ns	124.58 ns

Table 6.3: The 10-Time-Step Non-Pipelined FIR Filter Designs using the FDS Schedule

Description	*Dimensions* ($\mu m \times \mu m$)	*Area* (μm^2)
ChipCrafter Placement I	3301.00 × 3332.25	10999757.25
ChipCrafter Placement II	3485.79 × 3657.75	12750148.37
ChipCrafter Placement III	3708.50 × 3290.50	12202819.25
Manual Placement I	3982.00 × 3473.50	13831477.00
Manual Placement II	4005.25 × 3448.00	13810102.00

Description	*D.P. Delay*	*C.P. Delay*	*Exec. Delay*
ChipCrafter Placement I	99.97 ns	45.86 ns	145.83 ns
ChipCrafter Placement II	116.65 ns	36.96 ns	153.61 ns
ChipCrafter Placement III	116.18 ns	37.24 ns	153.42 ns
Manual Placement I	102.91 ns	48.43 ns	151.34 ns
Manual Placement II	100.11 ns	49.05 ns	149.16 ns

Table 6.4: 10-Time-Step Non-Pipelined FIR Filter Designs using the MAHA Schedule

placements produced by the ChipCrafter placement program for each schedule, the layouts with minimal area are chosen to represent the placements produced by ChipCrafter in these charts. It is found that the design characteristics (i.e. the performance and area) are improved in the layouts using the LADS placement. The performance of the layout with pin assignment using the LADS schedule is similar to the one without pin assignment but the area is improved. However, the design characteristics do not vary much for the layouts with and without pin assignment using a Force-Directed schedule and MAHA schedule due to the complex interconnections.

As for comparing the design characteristics, the results of the layouts using LADS schedule are favorable. The execution delay of the LADS design schedule is about 40% less than the designs using other scheduling techniques. The data path delay of the LADS design schedule is about 30% less than the designs using other scheduling techniques. It is found from this set of experiments that a scheduling program may produce very poor schedules in the register-transfer level if the effects of wiring and interconnections are completely ignored during the scheduling process. The 3D scheduling technique using the interconnection information fed back from the floorplanner is able to produce better schedules than traditional scheduling techniques.

6.3 A Pipelined FIR Filter Example

For the pipelined FIR filter design example, we used **carry select adders** for all additions in the control/data flow graph. An initiation-interval-4 9-time-step design in which four adders and two multipliers were allocated was created by LADS. The schedule and a tentative floorplan are shown in Figures 6.13 and 6.14, respectively.

In order to compare the results using the 3D scheduling technique to other scheduling techniques, we applied the same performance constraint to Sehwa and a Force-Directed scheduler. Both programs produced the same schedule result, which is a 6-time-step design, as shown in Figure 6.15. In this example, LADS produced a design with a longer pipeline length because LADS took advantage

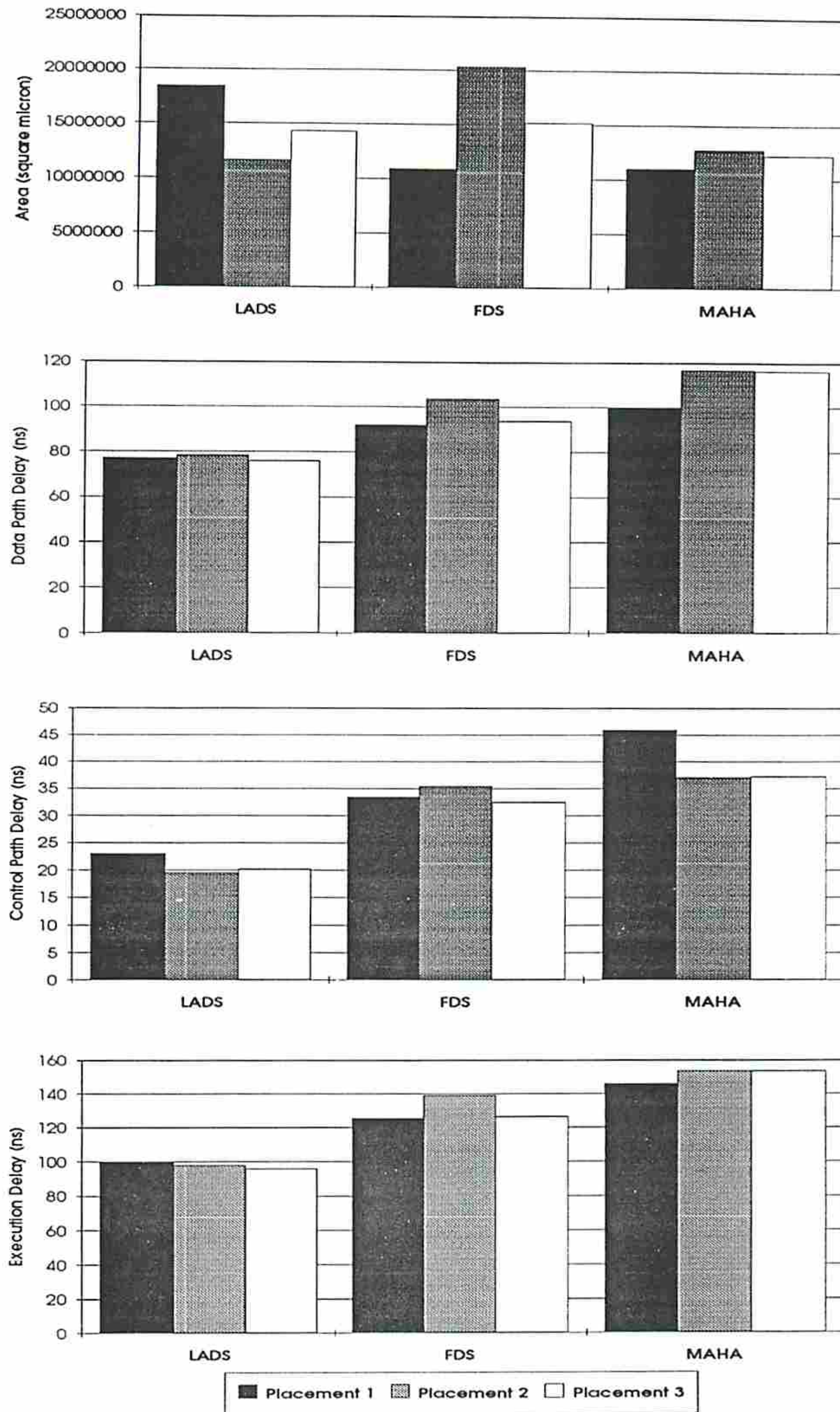


Figure 6.11: Comparisons with Different ChipCrafter Placements

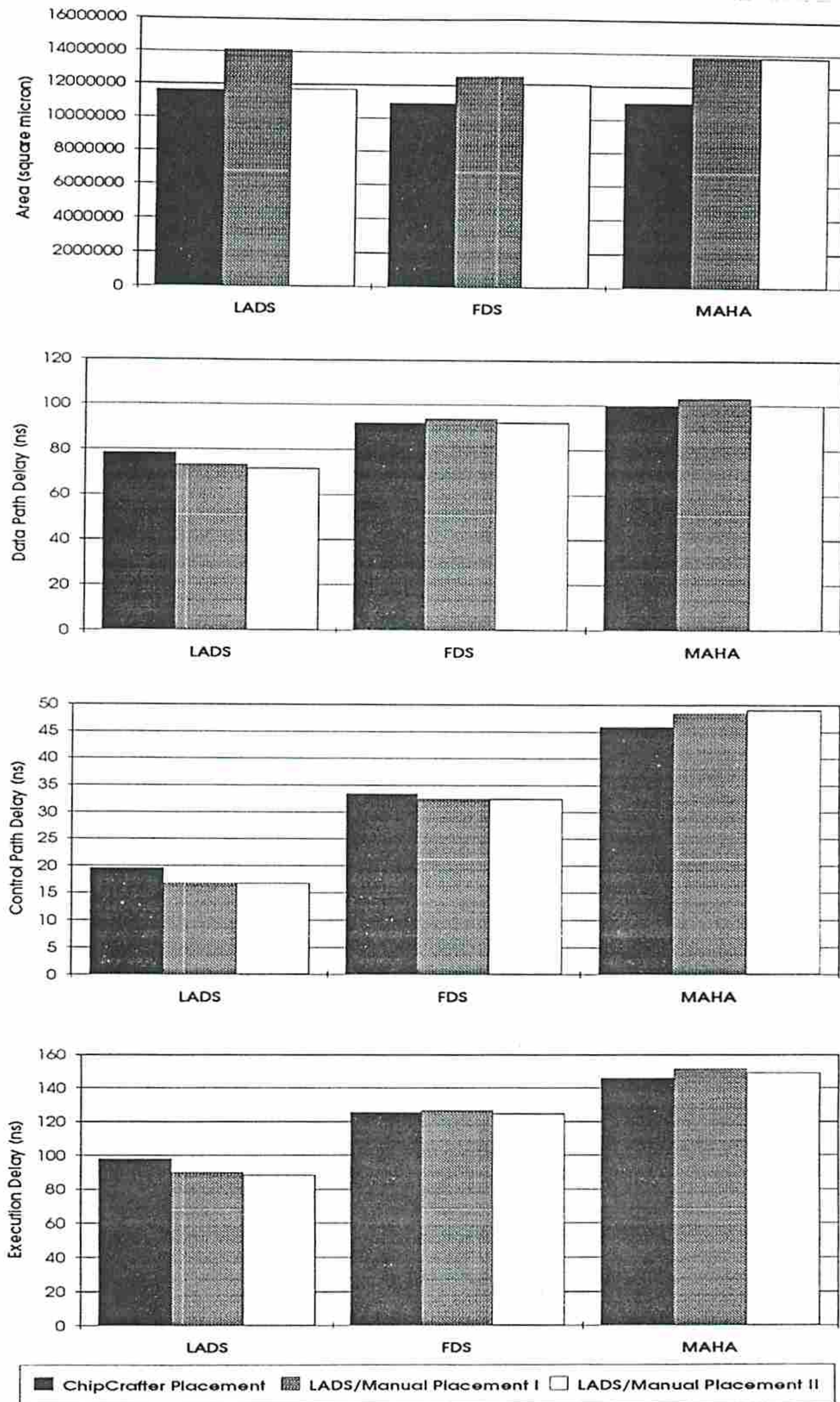


Figure 6.12: Comparisons with Different Placement Strategies

of the fact that chaining two adders into one time step will cause longer clock cycles in the final implementation step due to wiring delays. From the previous non-pipelined FIR filter example, we found that chaining two adders into one time step versus no operator chains results a 30–40% longer clock cycle (e.g. the execution delay of the design *LADS Placement II* is 88.22 ns versus 125.15 ns in the design *ChipCrafter Placement I* using the FDS schedule). Since there are two adders chained into one time step (i.e. the tentative critical paths) in the Sehwa schedule (or Force-Directed schedule) and there are no operator chains in the LADS schedule, it is expected that the clock cycle of the design using the Sehwa schedule (or Force-Directed schedule) is 30–40% longer than the design using LADS schedule. The through-put of the design using the Sehwa schedule (or Force-Directed schedule) is expectantly 30–40% slower. Therefore, LADS schedule can produce a better performance design in a low resynchronization rate case.

6.4 A Non-Pipelined Differential Equation Solver Example

In this example, we are going to show that the floorplan produced by LADS is able to reduce the wiring delays along the critical path. Figure 6.16 shows the schedule of a 4-time-step non-pipelined differential equation solver example produced by LADS. We used a **ripple carry adder** for all additions in this experiment. The complete register-transfer level design synthesized by MABAL is shown in Figure 6.17. Figure 6.18 shows the floorplan produced LADS. The modules shown with solid lines are the functional units along the critical path in the data flow graph. The modules shown with lightly hatched patterns are the functional units not on the critical path. It is found that in this example LADS successfully groups the functional units along the critical path in the data flow graph together and places them in the center of the floorplan to reduce the wiring delays along the critical path in the data flow graph. The modules on the off-critical paths in contrast are placed to corners of the floorplan because the execution delays are not critical along these paths.

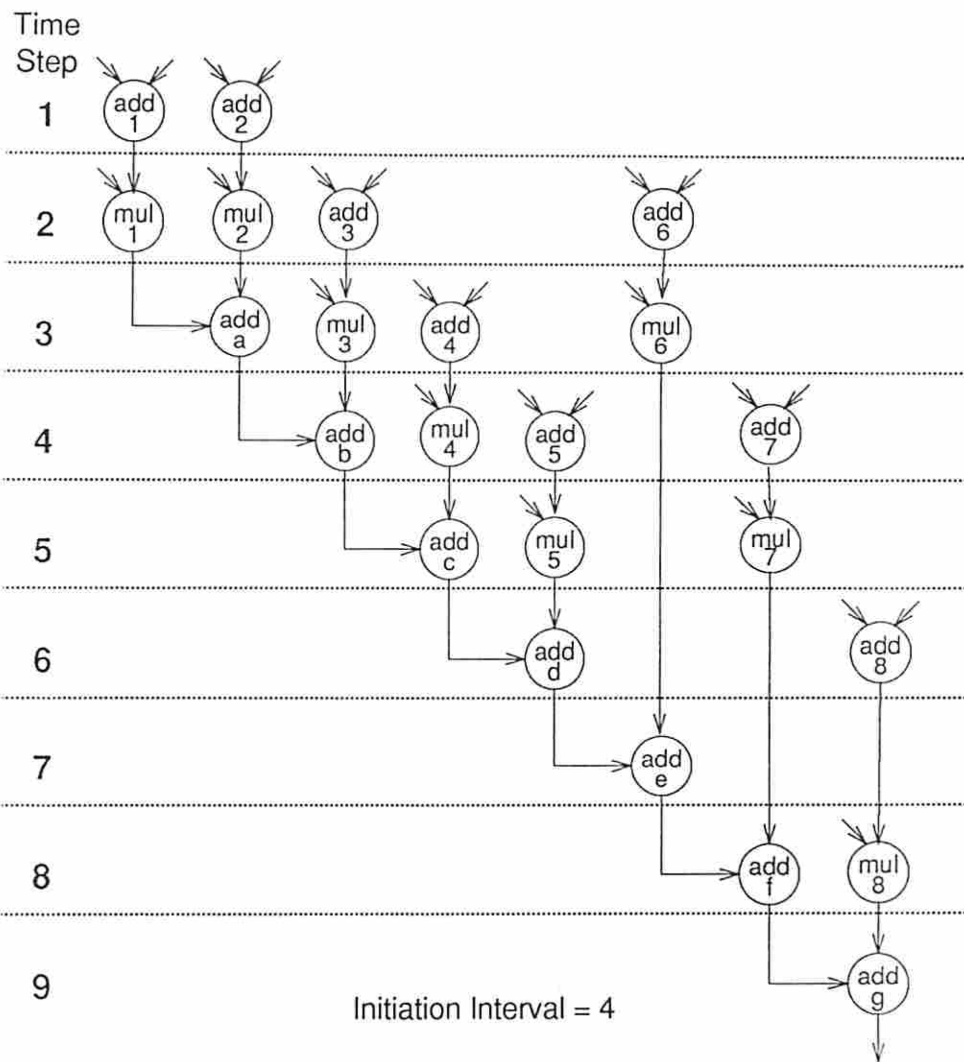
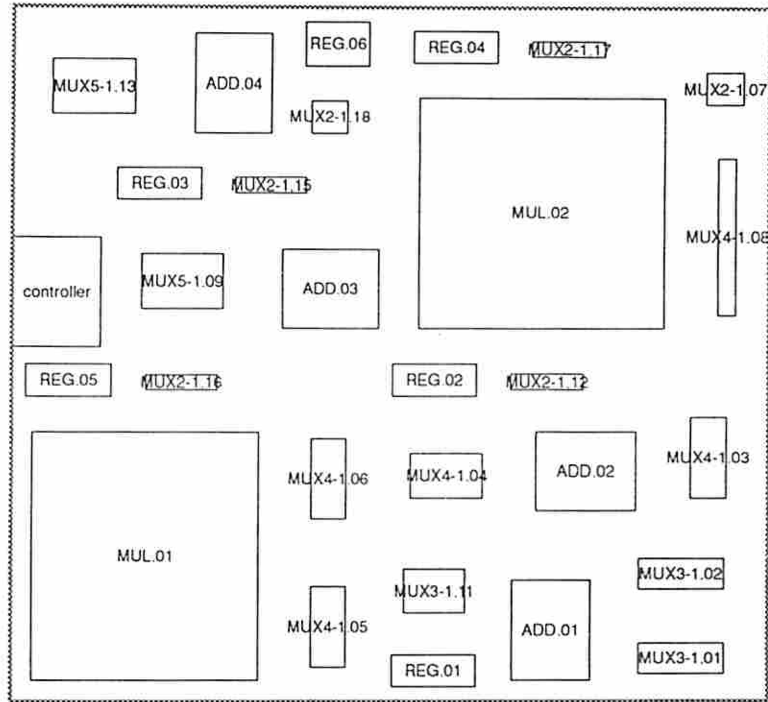


Figure 6.13: The Pipelined FIR Filter Schedule by LADS (Init.I. = 4, P.Len. = 9)



PD-4-9 (est. D.P. delay 85.90, C.P. delay 34.44)
Size: 4322.65 x 3894.25, Area: 16833469.87 (A.R. 1.11)

Figure 6.14: The Floorplan for Pipelined FIR Filter Design Created by LADS

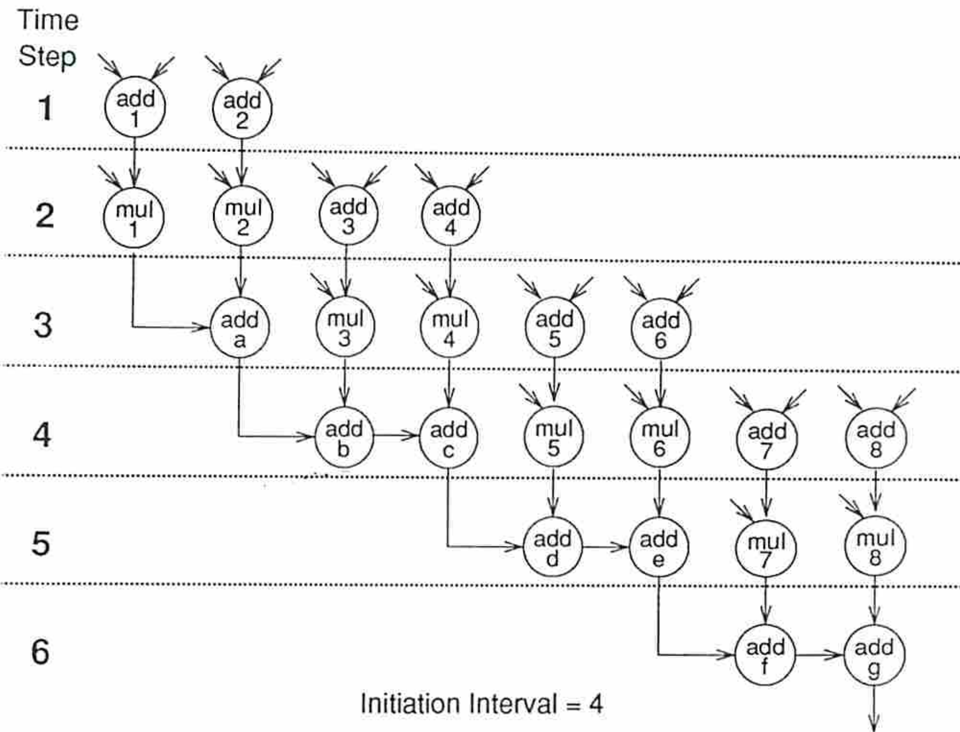


Figure 6.15: The Pipelined FIR Filter Schedule by Sehwa (Init.I.=4, P.Len.=6)

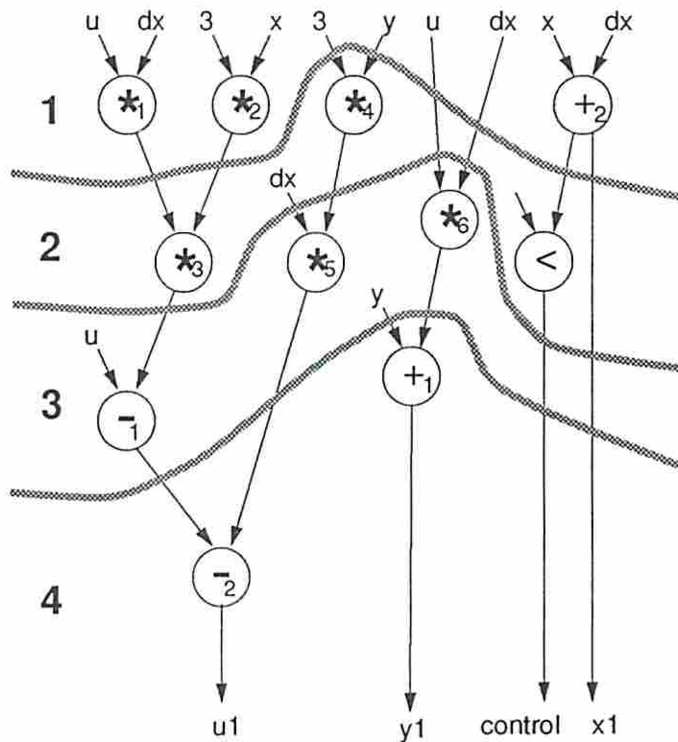


Figure 6.16: The 4-Time-Step Non-Pipelined Differential Equation Solver Schedule

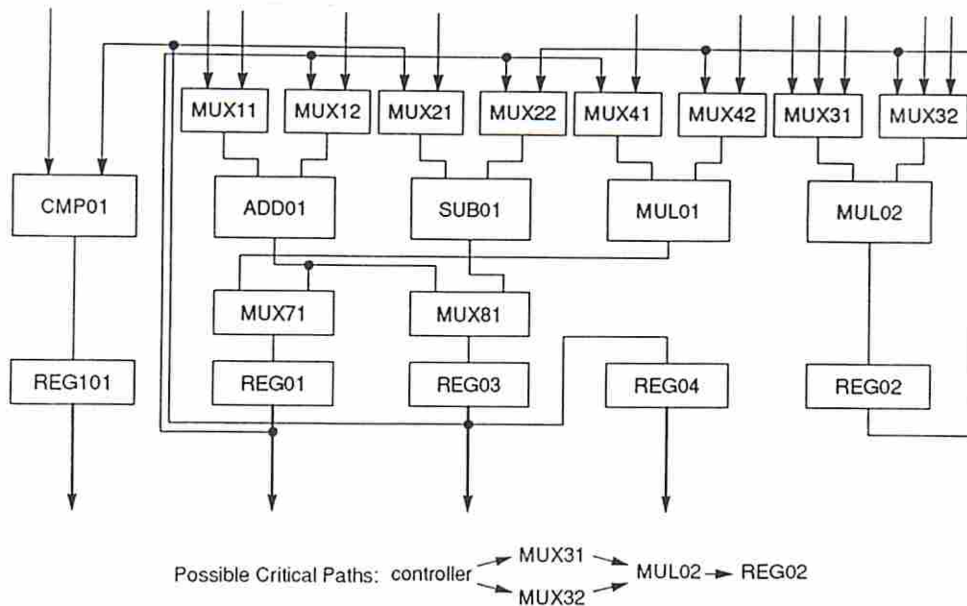
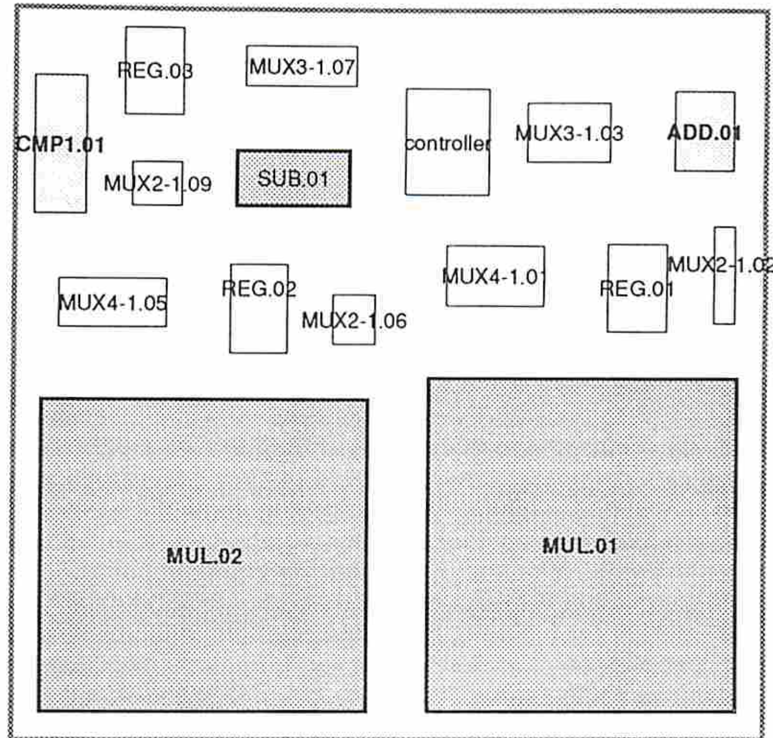


Figure 6.17: The RTL Design for the 4-Time-Step Differential Equation Solver

Figures B.1 and B.2 show the layouts using the placements produced by the simulated annealing placement program in ChipCrafter with different cooling schedules. Figure B.3 shows the layout produced by ChipCrafter using the LADS floorplan shown in Figure 6.18. The results of these layouts are summarized in Table 6.5. Note that the areas of these layouts are close. However, the layout using the LADS floorplan has a shorter critical path than the layouts using the placements produced by ChipCrafter. Since the placement program in ChipCrafter does not pay special attention to the wiring delays along the critical path, it results in modules not on the critical path being placed between the modules along the critical path. Due to the insertion of these modules along the critical path, an approximate 6% performance degradation is found in the layouts using the ChipCrafter placements.

6.5 Non-Pipelined Elliptic Filter Examples

Two non-pipelined elliptic filters using **ripple carry adders** to execute additions in a data flow graph were experimented with. Although a ripple carry adder has a longer propagation delay ($\approx$ 35ns) than a carry select adder ($\approx$ 25ns), two cascaded ripple carry adders have a shorter critical path delay ($\approx$ 30ns) than two



Estimated data path delay 74.69, control path delay 23.40
Size: 3188.50 x 3057.25, Area: 9748041.62
Aspect Ratio = 1.04

Figure 6.18: The Floorplan for the 4-Time-Step Differential Equation Solver Produced by LADS

Description	*Dimensions* ($\mu m \times \mu m$)	*Area* (μm^2)
LADS Prediction	3188.50 × 3057.25	9748041.62
ChipCrafter Placement I	3795.29 × 3358.00	12744583.82
ChipCrafter Placement II	3114.50 × 3601.25	11216093.12
LADS Placement	3442.50 × 3506.50	12071126.25

Description	*D.P. Delay*	*C.P. Delay*	*Exec. Delay*
LADS Prediction	74.69 ns	23.40 ns	98.09 ns
ChipCrafter Placement I	82.71 ns	49.63 ns	132.34 ns
ChipCrafter Placement II	82.64 ns	50.13 ns	132.77 ns
LADS Placement	78.12 ns	48.86 ns	126.98 ns

Table 6.5: The 4-Time-Step Non-Pipelined Differential Equation Solver Example

cascade carry select adders ($\approx$ 50ns). 3D scheduling takes into account the delays introduced by cascaded modules to give the scheduler more degrees of freedom to perform the scheduling. Due to the possibilities of adder chaining in the data flow graph, the elliptic filter design provides a good example to demonstrate the use of extra degrees of freedom to produce better schedules than traditional approaches.

6.5.1 A 12-Time-Step Non-Pipelined Elliptic Filter Example

Figure 6.19 shows the schedule of a 12-time-step non-pipelined elliptic filter design produced by LADS. The register-transfer level design produced by MABAL is shown in Figure 6.20. A respective tentative floorplan is shown in Figure 6.21. From the register-transfer level design, the floorplan for final implementation was modified manually from the floorplan produced by LADS to include additional registers added by MABAL. Note that the sizes of some multiplexers were also changed to be compatible with the design produced by MABAL as shown in Figure 6.22.

Seven layouts (shown in Figures C.1-C.7) were produced by ChipCrafter for the schedule shown in Figure 6.19. The placements of the first five layouts were produced by ChipCrafter's simulated annealing based placement program with different cooling schedules. Two placements, *LADS Placement I* and *LADS Placement II* as shown in Figures 6.22 and 6.23, using the LADS floorplan but with different manual placements for the modules added by MABAL were experimented with. The respective layouts are shown in Figures C.6 and C.7. Due to the more complicated interconnections between modules in this register-transfer level design (see Figure 6.20) and the lack of automatic pin assignment tools, we are unable to produce pin assignments manually. Therefore, the effects of pin assignments were not experimented with in this set of experiments. However, we expect an average of 7.5% reduction in the layout area and 25% reduction in the total interconnection length with proper pin assignments [PMSK90].

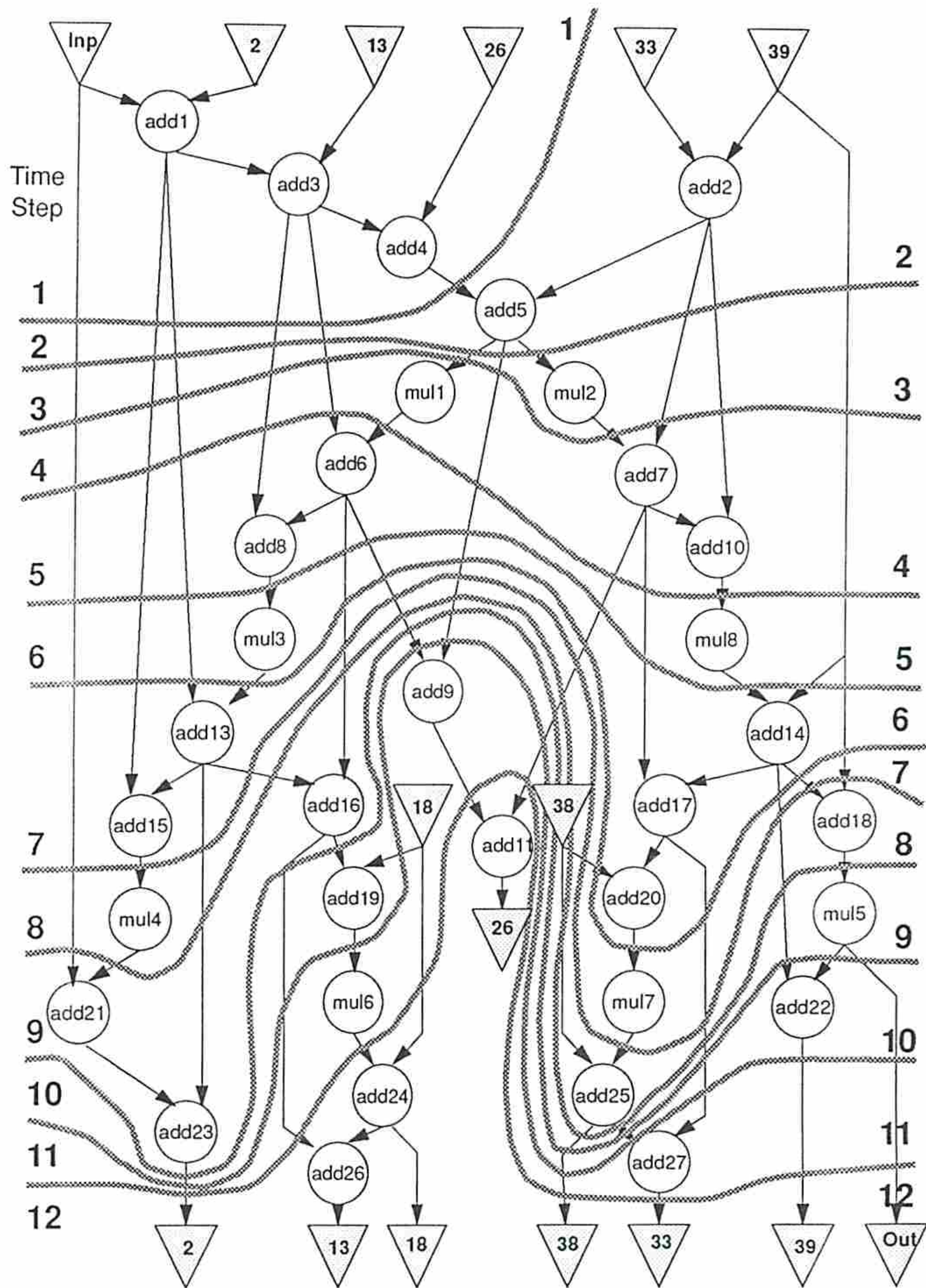


Figure 6.19: The 12-Time-Step Non-Pipelined Elliptic Filter Schedule by LADS

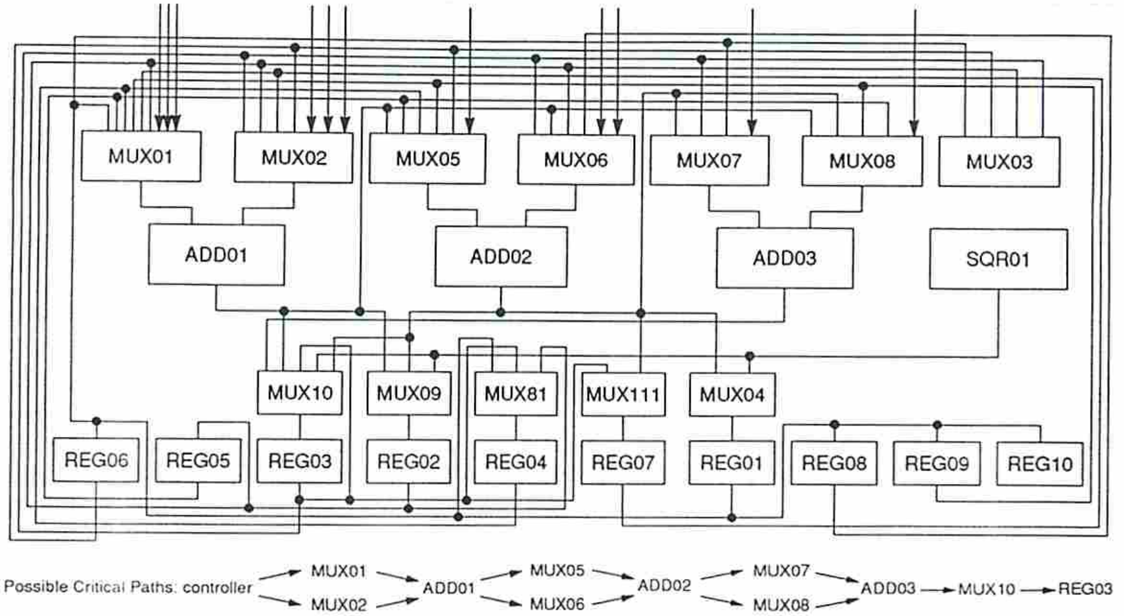
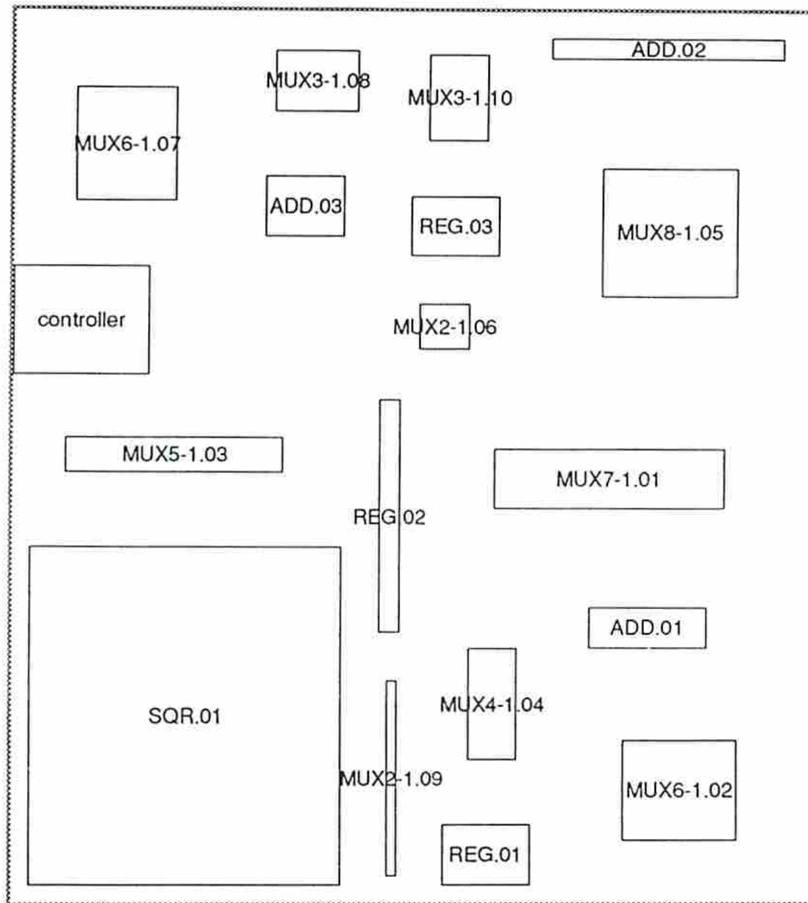


Figure 6.20: The RTL Design of 12-Time-Step Non-Pipelined Elliptic Filter by MABAL

Table 6.6 shows the design characteristics of the layouts. The area of layouts are larger than we predicted because an essential amount of the additional interconnections (e.g. two multiplexers, seven registers and respective interconnection wirings) are needed to store the values between time steps. The delays of all the layouts are also longer than the scheduler estimated (about 30% longer). From the results of the timing analysis, we found that the signal propagation from the registers to the input of multiplexers has much longer delays than we predicted. For example, the signal propagation from register REG06 to multiplexer MUX02 took approximately $10ns$ in the *LADS Placement I* layout.

After further investigation of the results of the timing analysis and the register-transfer level design, we noticed that unexpectedly long delays are caused by large fanouts and improper sizes of the output buffers of the datapath registers and functional modules in this design example. Since we are not able to assign the buffer sizes individually in ChipCrafter, an automatic buffer resizing tool provided within ChipCrafter is used to reduce wiring delays by changing the sizes of output buffers. The buffer resizing program does not change the placement of layouts; only the sizes of output buffers are altered according to their loads. Figures C.8-C.14



NPD-12 (est. D.P. delay 97.55, C.P. delay 31.38)

Size: 3367.00 x 3684.25, Area: 12404869.75 (A.R. 0.91)

Figure 6.21: The Floorplan for 12-Time-Step Non-Pipelined Elliptic Filter Design by LADS

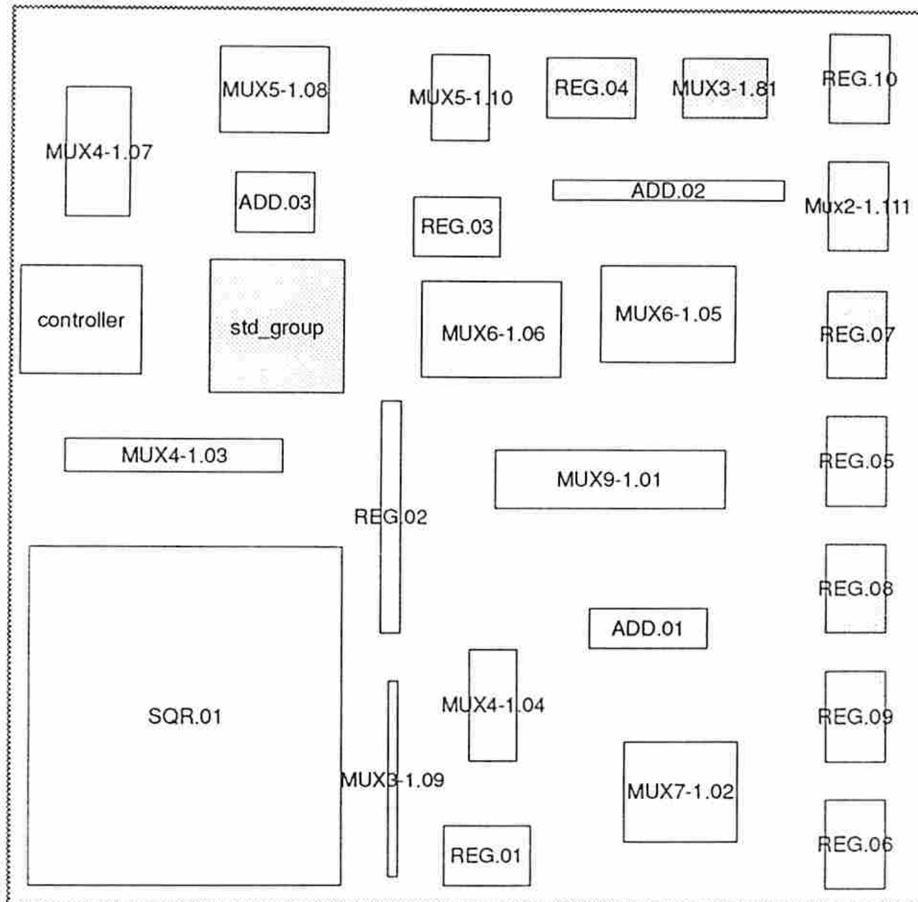


Figure 6.22: The Final Implementation Floorplan *LADS Placement I* for 12-Time-Step Non-Pipelined Elliptic Filter Design

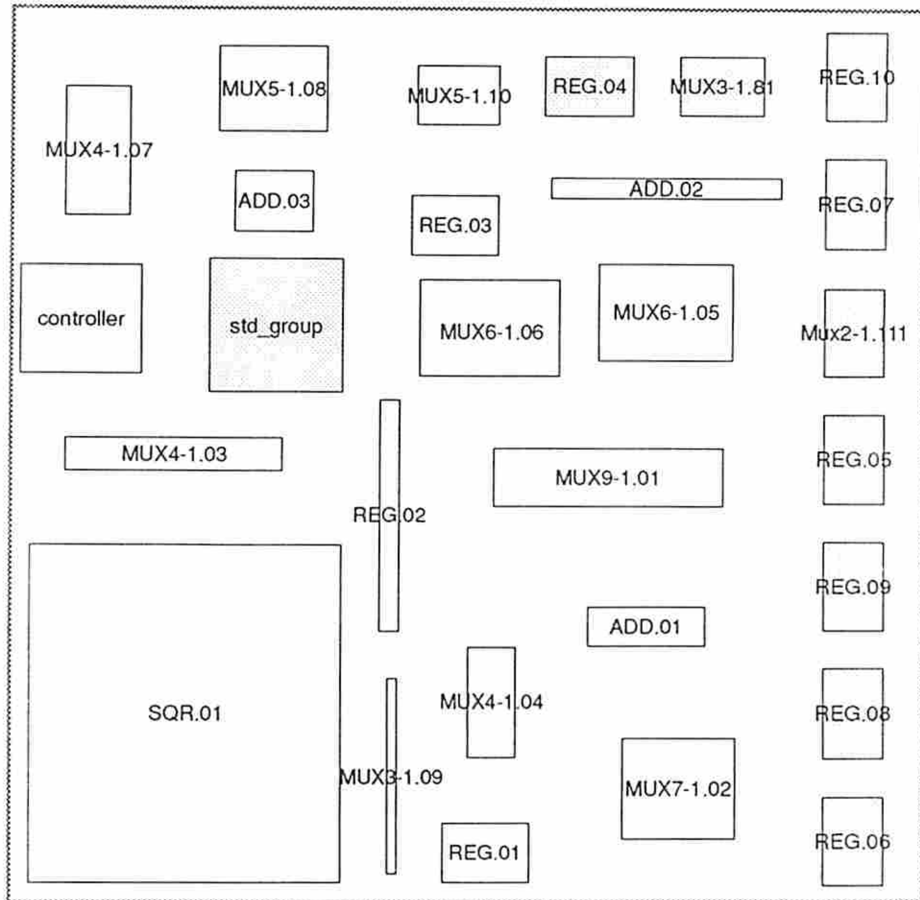


Figure 6.23: The Final Implementation Floorplan *LADS Placement II* for 12-Time-Step Non-Pipelined Elliptic Filter Design

Description	*Dimensions* ($\mu m \times \mu m$)	*Area* (μm^2)
LADS Prediction	3367.00 × 3684.25	12404869.75
ChipCrafter Placement I	4240.25 × 4981.71	21123695.83
ChipCrafter Placement II	4367.50 × 4932.00	21540510.00
ChipCrafter Placement III	4352.75 × 6419.54	27942652.73
ChipCrafter Placement IV	2936.75 × 8148.04	23928756.47
ChipCrafter Placement V	3923.75 × 5454.96	21403899.30
LADS Placement I	4067.00 × 4733.00	19249111.00
LADS Placement II	4038.75 × 4577.50	18487378.12

Description	*D.P. Delay*	*C.P. Delay*	*Exec. Delay*
LADS Prediction	97.55 ns	31.38 ns	128.93 ns
ChipCrafter Placement I	135.48 ns	10.90 ns	146.58 ns
ChipCrafter Placement II	138.35 ns	11.70 ns	150.04 ns
ChipCrafter Placement III	151.47 ns	13.87 ns	165.34 ns
ChipCrafter Placement IV	156.90 ns	13.87 ns	170.77 ns
ChipCrafter Placement V	136.23 ns	10.82 ns	147.05 ns
LADS Placement I	133.93 ns	17.70 ns	151.63 ns
LADS Placement II	130.05 ns	16.83 ns	146.88 ns

Table 6.6: The 12-Time-Step Elliptic Filter Designs before Buffer Resizing

show the layouts and simulation critical paths after running the buffer resizing program. The layout characteristics after running the buffer resizing program are summarized in Table 6.7.

The design performance after running the buffer resizing program is quite satisfactory. Not only the delays are reduced in all the layouts, but the area of the some layouts is also reduced. Note that the data path delay estimated by LADS is longer than the actual layouts after running the buffer resizing program. This result is the same as the one that we obtained in the non-pipelined FIR filter example. We believe that this is due to the use of worst-case wire length in estimating wiring delay.

The data path delays of *LADS Placement* I_b and *LADS Placement* II_b are comparable or slightly longer than the data path delays of some layouts using the placement program within ChipCrafter. This is probably because a substantial amount of interconnections ($\approx$ 50% more modules) are allocated by MABAL after the scheduling is done. The additional interconnections are then manually placed

Description	*Dimensions* ($\mu m \times \mu m$)	*Area* (μm^2)
ChipCrafter Placement I_b	4168.25 × 4941.50	20597407.38
ChipCrafter Placement II_b	4330.75 × 4927.25	21338687.94
ChipCrafter Placement III_b	4352.75 × 6435.75	28013210.81
ChipCrafter Placement IV_b	2935.25 × 8114.75	23818819.94
ChipCrafter Placement V_b	3922.25 × 5290.25	20749683.06
LADS Placement I_b	4038.00 × 4771.75	19268326.50
LADS Placement II_b	4032.50 × 4644.25	18727938.12

Description	*D.P. Delay*	*C.P. Delay*	*Exec. Delay*
ChipCrafter Placement I_b	77.94 ns	4.28 ns	82.22 ns
ChipCrafter Placement II_b	78.38 ns	4.35 ns	82.73 ns
ChipCrafter Placement III_b	81.77 ns	4.24 ns	86.01 ns
ChipCrafter Placement IV_b	81.28 ns	4.58 ns	85.86 ns
ChipCrafter Placement V_b	79.11 ns	4.25 ns	83.36 ns
LADS Placement I_b	81.22 ns	4.53 ns	85.75 ns
LADS Placement II_b	78.35 ns	4.29 ns	82.64 ns

Table 6.7: The 12-Time-Step Elliptic Filter Designs after Buffer Resizing

along the boundary of the floorplan produced by LADS. However, in the placement produced by the placement program within ChipCrafter, all the modules are considered and placed at the same time which allows the placement program within ChipCrafter to produce better placements. Also notice that the simulation critical paths are changed before and after running the buffer resizing program. Simulation critical paths are also different among the layouts. By inspecting the register-transfer level design shown in Figure 6.20, many possible critical paths are found in this elliptic filter example. This fact leads to the conclusion that the LADS floorplanner cannot effectively reduce the delays along all the critical paths in this example to produce a good performance-driven floorplan.

To compare the results of 3D scheduling to traditional scheduling approaches, we ran MAHA on the elliptic filter example with the same functional unit area constraint. The schedule produced by MAHA is shown in Figure 6.24 which is a 17-time-step non-pipelined design (versus a 12-time-step non-pipelined design produced by LADS). It is obvious that MAHA is unable to utilize the knowledge about the actual delay caused by chaining ripple carry adders in a time step.

Therefore, only a 17-time-step non-pipelined schedule was produced by MAHA, which uses 5 more time steps than the schedule produced by LADS. A longer data path delay and execution delay in this 17-time-step non-pipelined design is expected.

6.5.2 A 10-Time-Step Non-Pipelined Elliptic Filter Example

A 10-time-step non-pipelined schedule for an elliptic filter was generated using LADS as shown in Figure 6.25. A tentative floorplan created by LADS is also shown in Figure 6.26. For comparison purposes, we ran MAHA for the elliptic filter using the same resource constraints. The schedule produced by MAHA (shown in Figure 6.27) is a 14-time-step non-pipelined design. This experiment shows that a shorter schedule was produced by LADS under the same resource constraint. We conjecture that a shorter schedule may be produced (i.e. a better performance design) when the scheduler is able to accurately estimate the delays of operation chains during scheduling.

The ability to estimate path delays accurately helps a scheduler to explore more possible schedules and potentially produce a shorter schedule. A shorter schedule in a non-pipelined design results in better performance. (i.e. a shorter execution period.) In a pipelined design, a shorter schedule means a shorter pipeline length. For a pipelined design, the performance is determined by its initial interval if resynchronization does not occur. In this case, a shorter pipeline length for a pipelined design does not change the performance. However, a pipelined design having resynchronization, which often occurs in practice, with a shorter pipeline length is able to reestablish pipeline execution faster when resynchronizations occur [PP88].

6.6 Robot Arm Controller Examples

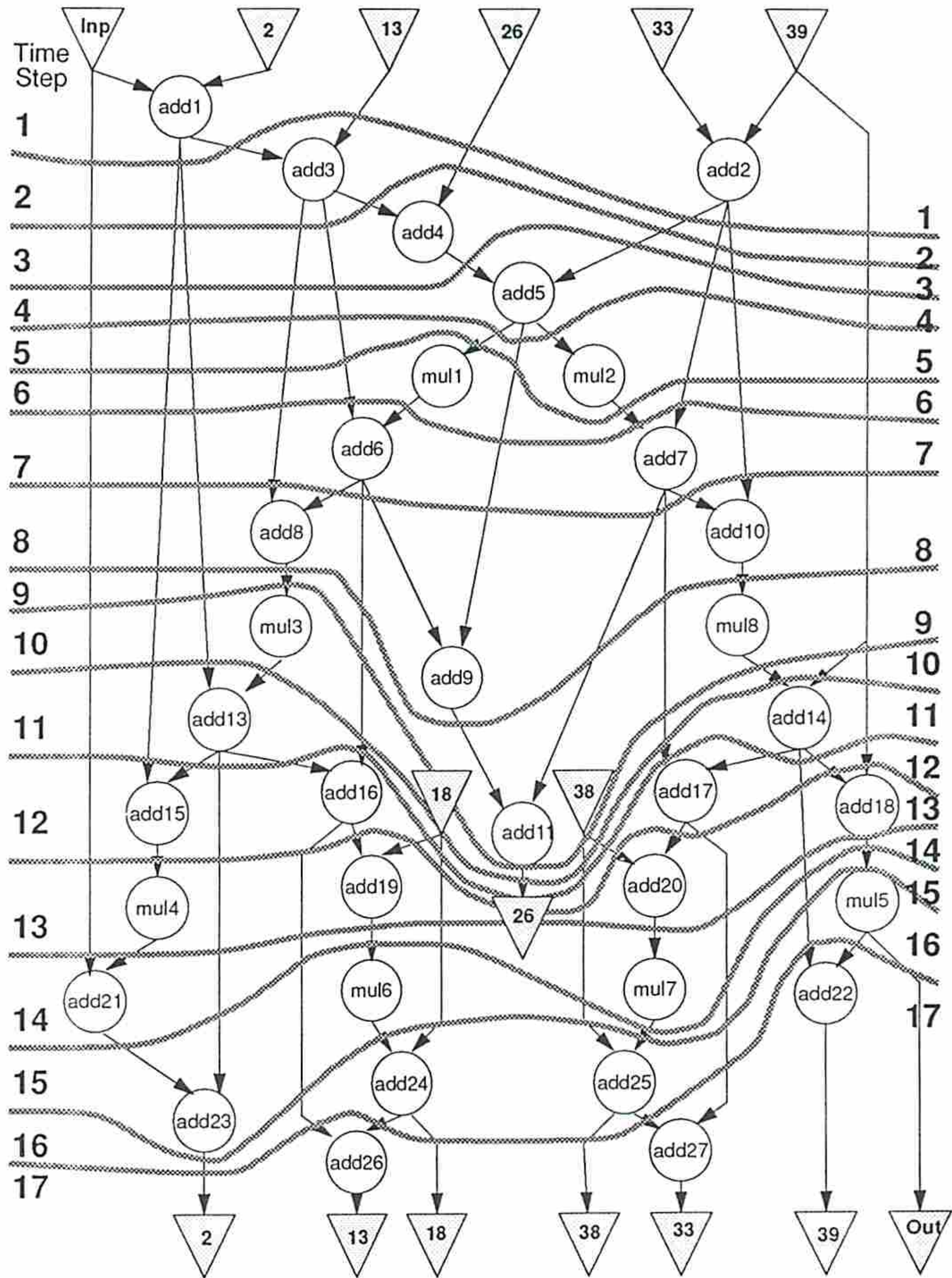


Figure 6.24: The 17-Time-Step Non-Pipelined Elliptic Filter Schedule by MAHA

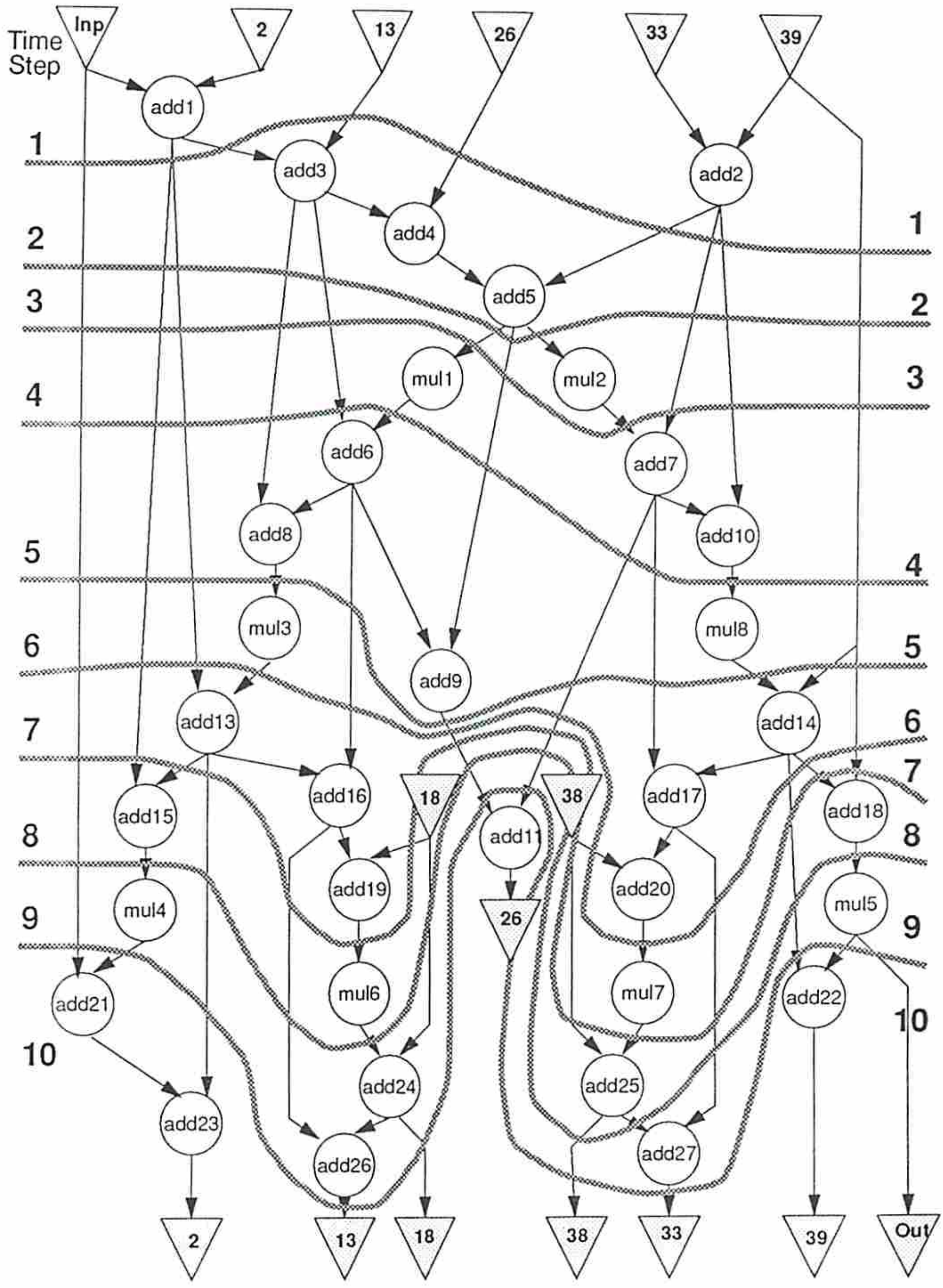
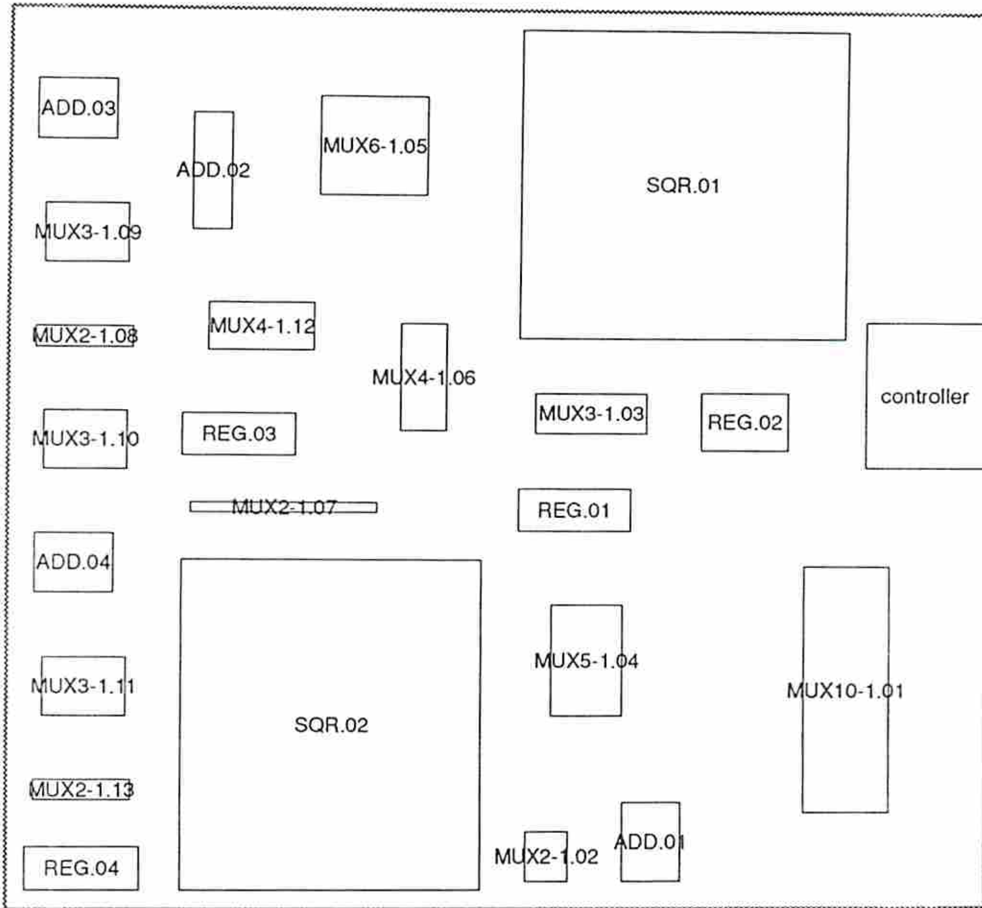


Figure 6.25: The 10-Time-Step Non-Pipelined Elliptic Filter Schedule by LADS



NPD-10 (est. D.P. delay 108.47, C.P. delay 33.99)
Size: 4184.25 x 3786.50, Area: 15843662.62 (A.R. 1.11)

Figure 6.26: The Floorplan for 10-Time-Step Elliptic Filter Design Created by LADS

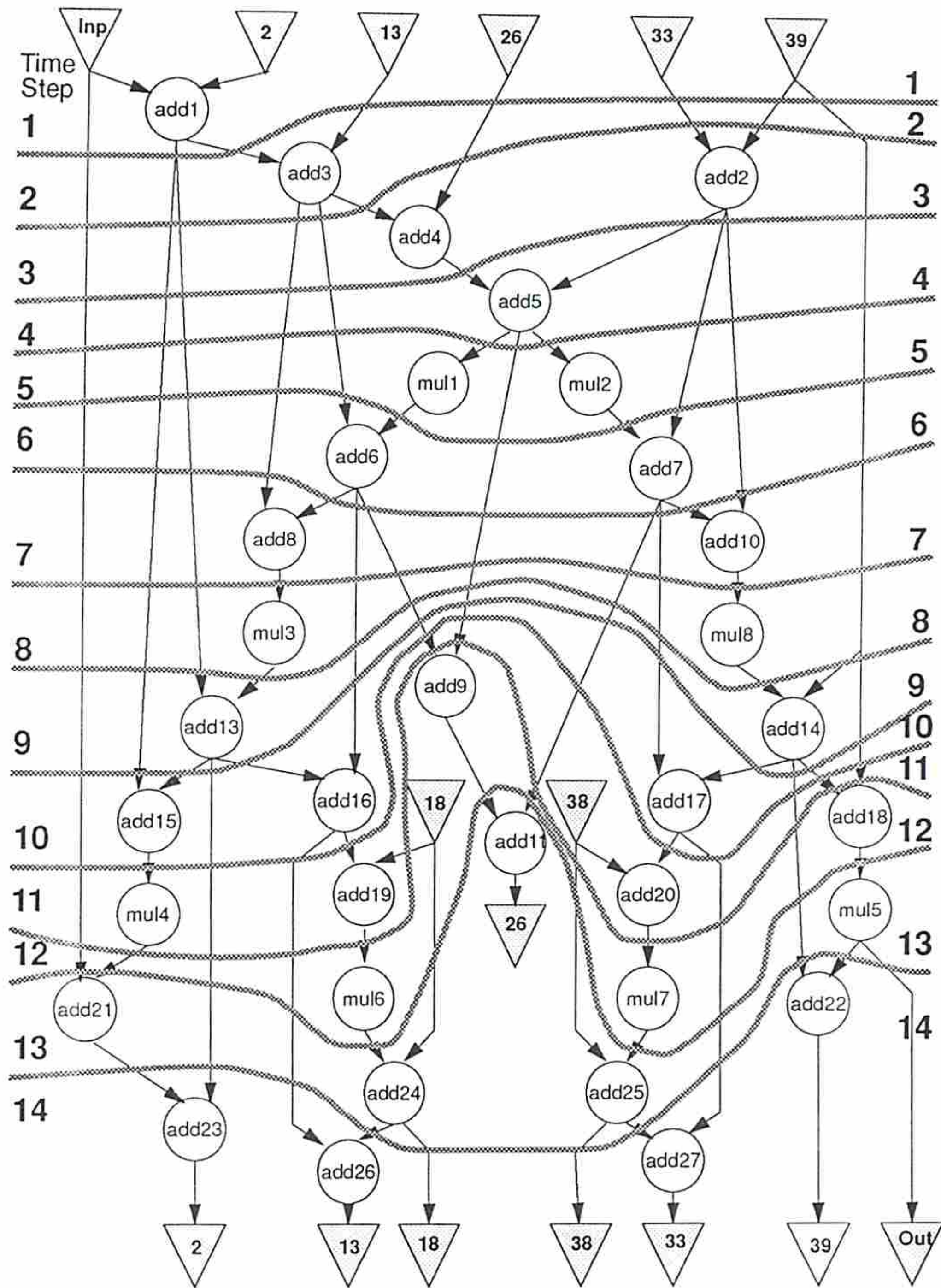


Figure 6.27: The 14-Time-Step Non-Pipelined Elliptic Filter Schedule by MAHA

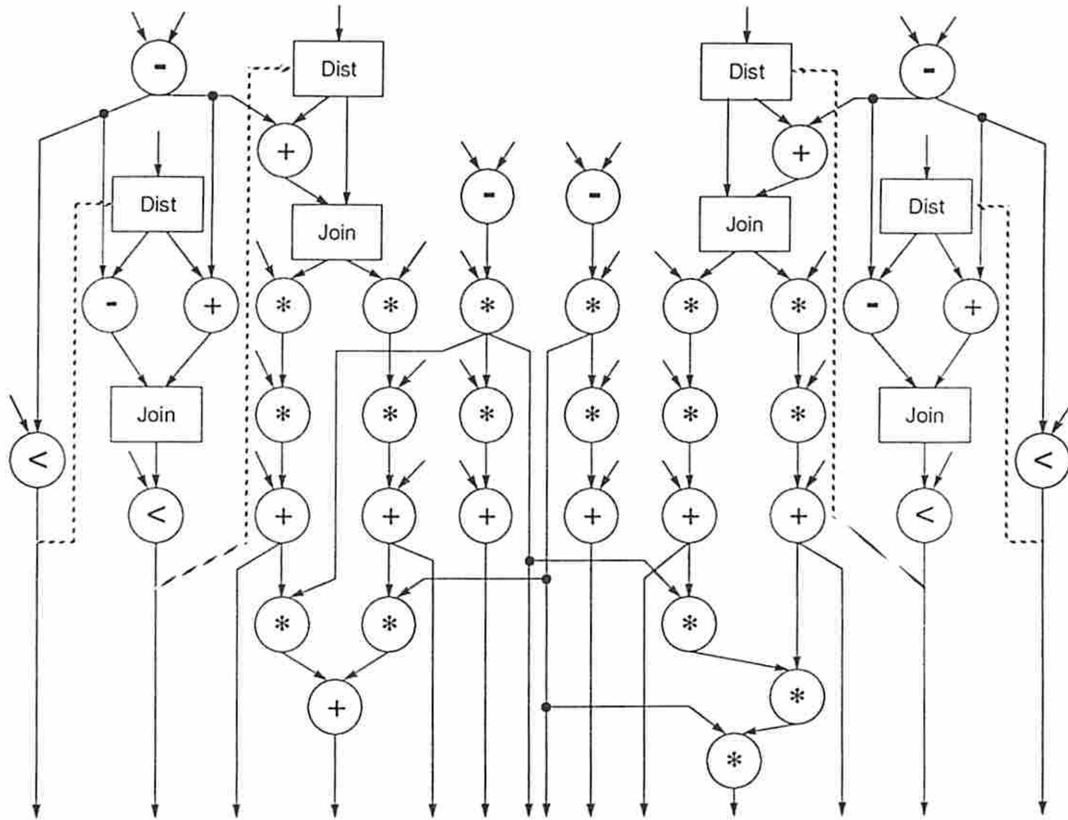


Figure 6.28: The Control/Data Flow Graph for the Robot Arm Controller

Non-Pipelined Design Name[1]	*Functional Units*				*Estimated Characteristics*	
	Cmp	*Add*	*Sub*	*Mul*	*D.P. Delay (ns)*	*Area* (μm^2)
Robot.11	1	2	3	4	93.42	22894387.66
Robot.12	2	2	2	3	95.85	21153578.12
Robot.13	1	2	2	3	90.76	19545892.38
*Robot.14**	1	2	2	3	90.68	19519360.88
*Robot.15**	1	2	2	3	95.65	20246519.36
Robot.16	1	1	1	2	90.54	16468172.00

Table 6.8: Non-Pipelined Robot Arm Controller Examples by LADS

For another experiment, a portion of the robot arm controller example obtained from UC-Berkeley served as our example. The robot arm controller was originally written in the C language. The C code was then translated into a VHDL description to obtain an internal control/data flow graph representation as shown in Figure 6.28. **Ripple carry adders** are used for all additions in the data flow graph.

Six non-pipelined designs and fourteen pipelined designs were experimented with as shown in Tables 6.8 and 6.9, respectively. The schedule results are similar to the ones using traditional scheduling approaches. This is probably because not many additions or subtractions exist and functional chainings are not possible in this example.

6.7 An Inner Loop Example

A portion of the robot arm controller serves as our inner loop example. The VHDL description of the robot arm controller example shown in Figure 6.29 has two nested inner loops. The translated data flow graph and timing graph are shown in Figures 6.30 and 6.31, respectively. The NOP operations are dummy operations which are introduced for the feedback of loop values [Che91].

[1]The naming convention for the non-pipeline robot arm controller designs is *Robot.[time steps]* for feasible designs and *Robot.[time steps]** for infeasible designs.

[2]The naming convention for the pipeline robot arm controller designs is *Robot.[time steps].[initial interval]* for feasible designs and *Robot.[time steps].[initial interval]** for infeasible designs; *Robot.[time steps].[initial interval]a* represents an alternative design.

```
--
-- This is the portion of a robot arm controller.
-- This was modified from the C code of the controller.
--

entity robotc2 is
 port( kp1, kv1, km11, km12, T1, K, L, mh11i,
       mh12i, ref1, xp1, xv1 : in Integer;
       q1 : out Integer);
end robotc2;

architecture Behaviour of robotc2 is

begin process
variable   mh110, mh111, mh120, mh121, ev1, xvh1,
           ep1, uv1, u10, u11, em1 : Integer;
variable   I, J : Integer;

begin

-- initialize states

      mh111 := mh11i;
      mh121 := mh12i;

      u11 := 0;   -- to set SubValuePath.

      xvh1 := 0;

      q1 <= 0;   -- to set SubValuePath.

-- BEGIN_OUTERLOOP

--- for J in 1 to K loop
      J := 0;
      while J < K loop
          J := J + 1;

-- I-control

          ep1 := ref1 - xp1;
          uv1 := kp1 * ep1;

-- BEGIN_INNERLOOP

--          for I in 1 to L loop
            I := 0;
            while I < L loop
                I := I + 1;

-- D-control

                ev1 := uv1 - xv1;
                u10 := kv1 * ev1;
                em1 := xvh1 - xv1;

-- ref. model

                xvh1 := u10 * T1 + xv1;

-- inertia estimation

                mh110 := km11 * em1 * u11 + mh111;
                mh120 := km12 * em1 * u11 + mh121;

-- torque estimation

                q1 <= mh110 * u10 + mh120 * u10;

-- update states and refresh constants in ram

                mh111 := mh110;
                mh121 := mh120;
                u11 := u10;

            end loop;

      end loop;

end process;

end Behaviour;
```

Figure 6.29: The VHDL Description of the Robot Arm Controller Example

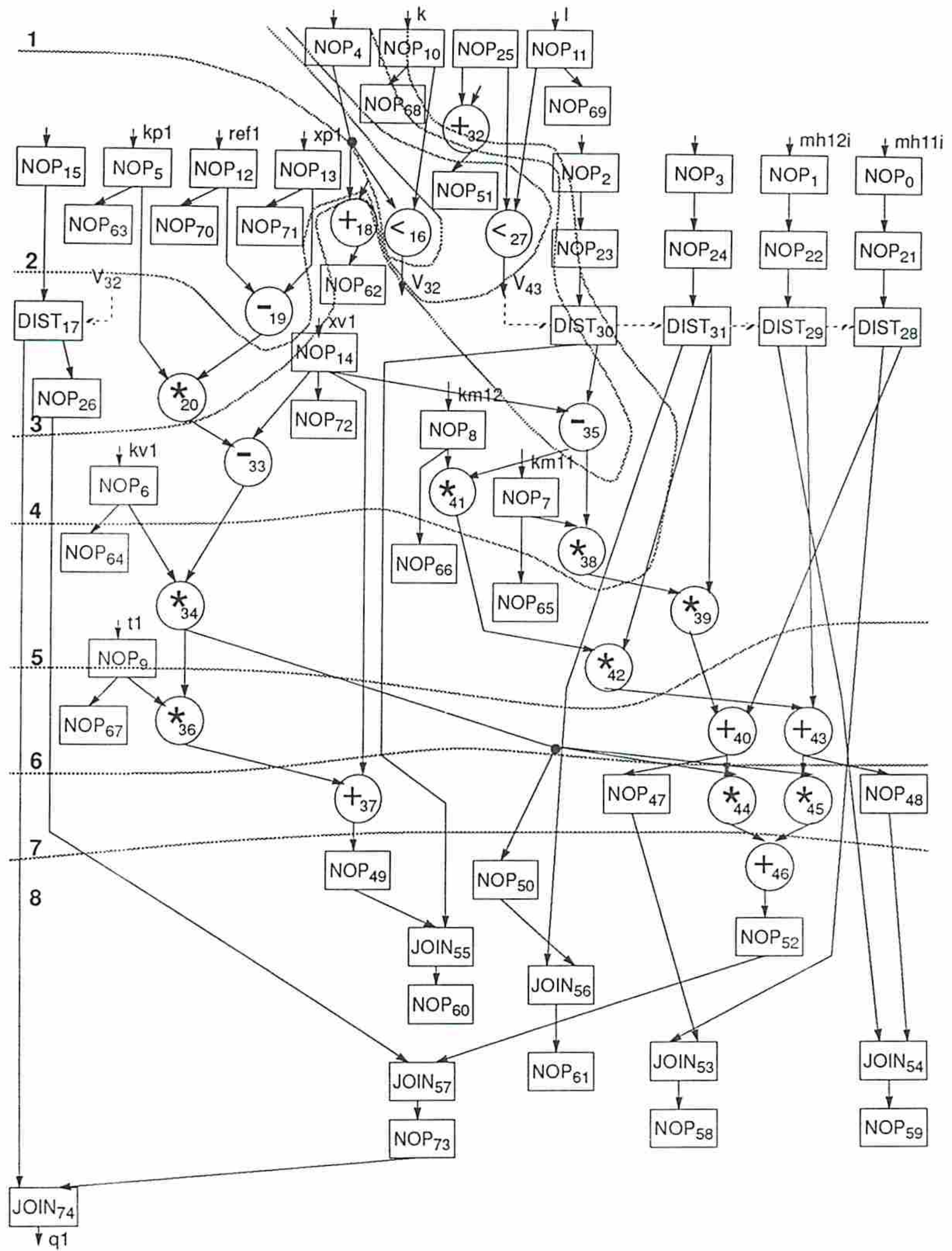


Figure 6.30: The Schedule and Data Flow Graph of the Robot Arm Controller

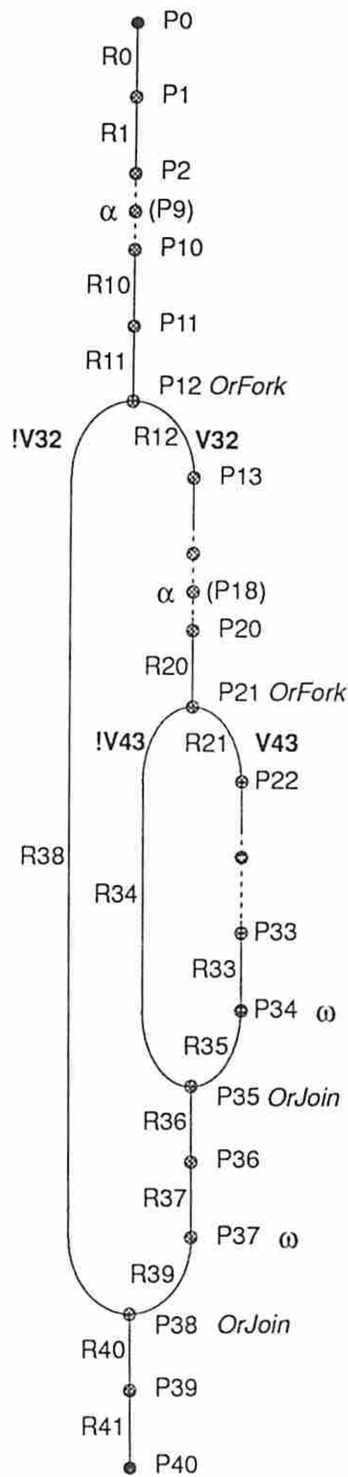


Figure 6.31: The Timing Graph of the Robot Arm Controller Example

Pipelined Design Name[2]	*Functional Units*				*Estimated Characteristics*	
	Cmp	*Add*	*Sub*	*Mul*	*D.P. Delay (ns)*	*Area* (μm^2)
Robot.11.4	1	4	2	5	93.45	28839741.56
Robot.12.4	1	3	2	5	96.44	28687950.00
*Robot.13.4**	1	3	2	5	97.85	29756338.12
Robot.14.4	1	4	2	5	92.46	29354080.88
Robot.15.4	1	3	2	5	103.02	28462010.00
Robot.16.4	1	3	2	5	92.75	28537831.88
Robot.11.9	1	2	2	4	96.81	22284067.88
Robot.12.9	2	2	2	3	93.10	21395779.43
*Robot.13.9**	1	2	1	3	94.95	21651528.00
Robot.13.9a	1	2	1	3	95.48	20675078.69
Robot.15.9	1	2	1	2	94.99	18002333.22
*Robot.15.9a**	1	2	1	2	99.30	18593615.34
Robot.16.9	1	2	1	2	95.14	16808586.25
Robot.17.4	1	2	1	2	90.56	18213024.50

Table 6.9: Pipelined Robot Arm Controller Examples by LADS

Ripple carry adders are used for all additions in the data flow graph. A schedule with 8 time steps using LADS is shown in Figure 6.30. Note that operations, such as $*_{34}$, $*_{39}$, $*_{42}$, $+_{40}$, $+_{43}$, $*_{44}$, $*_{45}$ and $+_{46}$, are closely coupled together in this example. Due to the limitations of the force-directed scheduling technique, we are unable to get more serialized design for the robot arm controller example.

Chapter 7

Control Path Synthesis

Many design problems of practical interest exhibit conditional branching behavior. However, most existing high-level synthesis systems only synthesize controllers for non-pipelined designs. ATOMICS system for Cathedral-II [ZSRM90] synthesizes ROM-based microprogrammed controllers for the data paths produced by Cathedral-II. Three pipeline stages can be identified in the control critical path of designs produced by ATOMICS. Pipelined designs that contain conditional branches were not given serious thought by high-level synthesis researchers until the mid '80's, when such pipelined data paths were first synthesized [PP88]. The issue of controllers for such data paths were only recently addressed (e.g. by Kim and Kurdahi [KK91]).

In this chapter, we present algorithms which generate a controller specification from a description of system behavior and the respective data path. The function of a controller is to issue control signals to the data path. Control signals select operations to be performed at specific time steps and route the processed data to appropriate functional units. Here, we are particularly interested in automatically synthesizing controllers for both pipelined and non-pipelined data paths produced by data-path dominated high-level synthesis tools, such as MABAL. Controllers for designs with or without conditional branches are discussed.

Data path scheduling, module allocation and bindings are assumed to have been performed to provide the exact schedule and data path information before control path synthesis. Three aspects of a controller need to be determined at this point, namely, the number of states, next state transitions of a given state

and activated control signals in a given state. The controller design process first determines a set of states that is sufficient to represent the control behavior, and then proceeds by determining outputs and next state transitions for each state, according to the input behavior. The design of a controller is usually a complex task since the controller must retain the current execution status to determine next state transitions and accordingly provide the proper control signals. This becomes more difficult when the design is pipelined.

The rest of this chapter presents a method for the synthesis of controllers for pipelined and non-pipelined data paths. We first discuss the assumptions made in our control path synthesis. The next section deals with the control signal generation for a scheduled design. The implementation techniques of controllers for designs with and without conditional branches are then given. Controller designs for both pipelined and non-pipelined designs are discussed.

7.1 Controller Assumptions

Historically, there are two basic controller design approaches—hardwired and microprogrammed [Hay88]. The former views the controller as a sequential circuit with a number of logic components, such as random logic and PLAs. Once a controller is constructed, changes in behavior can only be implemented by redesigning and physically rewiring the unit. Therefore, it is called a hardwired controller. On the other hand, a microprogrammed controller is designed around a control memory in which sets of micro instructions are stored. In general, a microprogrammed controller is often more costly and slower than a hardwired controller due to the presence of the control memory and its access circuitry. In this research, we will focus on hardwired controllers characterized by a state table of a finite state machine. The controller is assumed to be implemented by a PLA circuit. However, if a microprogrammed implementation is preferred, the finite state machine we synthesize can also be implemented by a microprogrammed controller.

The controller is modeled as a *Mealy-style* finite state machine in which state memory holds the present state and a combinational circuit decides next state transitions and output signals. Depending on the implementation style selected,

status registers may or may not be used in the controller synthesis. The purpose of introducing status registers into the controller is to store binary condition values in a design with conditional branches. If the controller design does not use status registers, condition values are stored in state memory.

7.2 Control Signal Generation

Control signals are organized as a set of tuples. A *tuple* has several attributes, such as the activated device name, the activated time step and activated condition values. For a multiplexer control signal, an extra attribute, *viz.* the selected port number, is added.

In order to determine execution conditions of condition values for operations along conditional branches, a mutually-exclusive condition analysis of the control/data flow graph is required. We analyze mutual exclusion conditions of a control/data flow graph using an algorithm similar to the one proposed by Park [PP88]. The algorithm assigns to every operation a label consisting of a sequence of one or more integer codes. Using these labels, we can test mutual exclusion between any pair of nodes (operations) and obtain activated condition values for operations inside conditional execution paths.

Control signals are obtained by combining the information from the input behavior and register-transfer level data path. A set of control signals is issued by the controller to achieve activities specified in the binding list under proper timing sequences. Basically, there are two classes of bindings in the list. An *operation binding* binds an operation to an operator and a *value binding* binds a data value to a register.

Three types of control signals are defined in our approach, namely, *register write signals*, *multiplexer port select signals* and *tri-state-driver enable signals*. These control contents of registers, the data transferred through multiplexers and the data values on buses, respectively. Inputs for an operation are expected to come from *external inputs* or *register outputs* which are specified in the operation bindings; and the input of a value binding should come from *an external input* or

operation's output. Control signals are then constructed after data transfer paths are identified from operation bindings and value bindings.

7.3 Designs without Conditional Branches

The controller for a design without conditional branches is simple and its state transition diagram is a ring. Time steps, to which operations are scheduled, are a continuous sequence of numbers in a scheduled control/data flow graph. The first time step starting of the execution is either time step 0 or time step 1 depending on whether the synthesized design has its input data latched before the execution of the process or not [KP90]. Without loss of generality, we assume that the schedule of a control/data flow graph is started at time step 1 unless specified explicitly. With a simple extension, the procedures discussed here can also handle designs starting at time step 0.

7.3.1 Controllers for Non-Pipelined Designs

The controller for a non-pipelined design without conditional branches can be realized by a simple finite state machine. The problem can be formally stated as follows: Given a scheduled n-time-step control/data flow graph $\mathcal{G}_n$, vertices are grouped to a set of *executions* $\mathcal{E} = \{\mathcal{E}_1, \mathcal{E}_2, \ldots, \mathcal{E}_n\}$. An *execution* $\mathcal{E}_i$ is defined as a set of control/data flow graph operations which are scheduled to be executed in time step i where $i = 1 \ldots n$. $\mathcal{E}_i$ and $\mathcal{E}_j$ are disjoint where $i, j = 1 \ldots n$ and $i \neq j$. Note that vertices of a control/data flow graph are basically the union of *executions* and distribute/join nodes.[1] The task of a controller is then to generate the set of control signals to ensure that all operations in $\mathcal{E}_i$ are executed according to the behavior specified in time step i, $i = \ldots n$.

A *state transition diagram* $\mathcal{T}_n$ is a graph with a set of *states* $\mathcal{S} = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n\}$ as its vertices and *transition arcs* as its edges. A *state* consists of *a set of control signals* which is activated when the state is being visited.

[1]Distribute/join nodes in a control/data flow graph are not assigned to any time step in the current ADAM system.

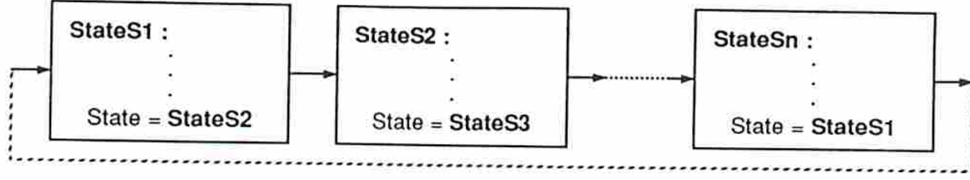


Figure 7.1: A Ring State Transition Diagram

Definition 7.1 *A* ring state transition diagram $\mathcal{R}_n$ *is a state transition diagram* $\mathcal{T}_n$ *whose vertex set is* $V = \{\mathcal{S}_1, \mathcal{S}_2, \ldots, \mathcal{S}_n\}$ *and whose edges are* $\{\mathcal{S}_1, \mathcal{S}_2\}, \{\mathcal{S}_2, \mathcal{S}_3\}, \ldots, \{\mathcal{S}_{n-1}, \mathcal{S}_n\}, \{\mathcal{S}_n, \mathcal{S}_1\}$.

An example of a *ring state transition diagram* is shown in Figure 7.1. We deduce the following lemma from the definition of the ring state transition diagram.

Lemma 7.1 *A* ring state transition diagram $\mathcal{R}_n$ *is sufficient to specify the control behavior of a n-time-step non-pipelined design without conditional branches.*

Proof: For an n-time-step non-pipelined design, a new data set is processed every n clock cycles (i.e. the behavior is repeated every n clock cycles). Under the proposed two-phase non-overlapping clocking scheme (see Section 3.2 for more details), operations being scheduled in time step i (where $1 \leq i \leq n$) should be executed and finished within a data-path clock cycle. In a design without conditional branches, activated control signals are merely dependent on the clock cycle being executed (or the state being visited).

Each time step in a non-pipelined design without conditional branches requires only one state to specify the activated control signals, so only n states are thus needed for an n-time-step non-pipelined design without conditional branches. Since the behavior is repeated every n clock cycles, a *ring state transition diagram* $\mathcal{R}_n$ is sufficient to specify the controller for an n-time-step non-pipelined design without conditional branches. □

Lemma 7.1 reveals that the number of states required to specify the controller for a non-pipelined design is equal to the number of time steps in the scheduled data flow graph. The controller can thus be realized by defining a *bijective* function, $\mathcal{F} : \mathcal{E}_i \rightarrow \mathcal{S}_i$ where $i = 1 \ldots n$. The *control signal generating function* $\mathcal{F}$ produces a

set of control signals which are activated during the execution of operations within *execution* $\mathcal{E}_i$, and the set of control signals is assigned to *state* $\mathcal{S}_i$. By assigning the first state to state $\mathcal{S}_1$, the controller design for a non-pipelined design without conditional branches can be completed.

7.3.2 Controllers for Pipelined Designs

The controller design for a pipelined design is similar to the procedure for a non-pipelined design. The basic idea is to "*fold*" the scheduled control/data flow graph every initiation interval. Therefore, the number of time steps in the folded control/data flow graph is equal to the length of the initiation interval, and operations in the folded control/data flow graph are executed every initiation interval.

We now present a formal model of the folded control/data flow graph for a pipelined design. Vertices in the *folded* scheduled n-time-step control/data flow graph $\mathcal{G}_{pn}$ for a pipelined design with an initiation interval I can be represented by a set of *p-executions*, $\mathcal{E}_p = \{\mathcal{E}_{p1}, \mathcal{E}_{p2}, \ldots, \mathcal{E}_{pI}\}$. A *p-execution* $\mathcal{E}_{pi}$ is defined as the union of executions $\{\mathcal{E}_i, \mathcal{E}_{i+I}, \mathcal{E}_{i+2I}, \ldots, \mathcal{E}_{i+mI}\}$ where $i + mI \leq n$. Note that *p-executions*, $\mathcal{E}_{pu}$ and $\mathcal{E}_{pv}$, are disjoint where $u, v = 1 \ldots I$ and $u \neq v$.

Lemma 7.2 *A* ring state transition diagram $\mathcal{R}_I$ *is sufficient to specify the control behavior for an initiation-interval-I pipelined design without conditional branches.*

Proof: The proof is similar to Lemma 7.1. For an initiation-interval-I pipelined design, a new data set is processed every I clock cycles. Under the proposed two-phase non-overlapping clocking scheme, operations being scheduled in time step $i, i + I, \ldots, i + mI$ (i.e. the folded time step i, where $1 \leq i \leq I$, $1 \leq i + mI \leq n$ and n is the pipeline length) should be executed and finished within a data-path clock cycle. In a design without conditional branches, activated control signals are merely dependent on the clock cycle being executed (or the state being visited).

Each folded time step in a pipelined design without conditional branches requires only one state to specify activated control signals, I states are thus needed for an initiation-interval-I pipelined design without conditional branches. Since the behavior is repeated every I clock cycles, a *ring state transition diagram* $\mathcal{R}_I$

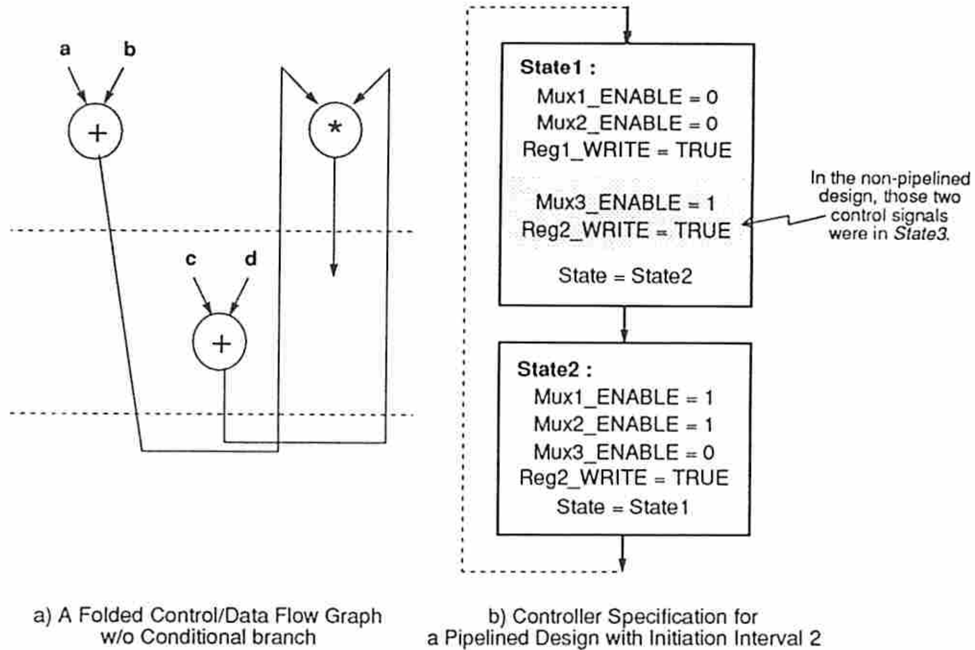


Figure 7.2: A Simple Pipelined Design Synthesis Example

is sufficient to specify the controller for an initiation-interval-I pipelined design without conditional branches. □

The previous lemma shows that the number of states required to specify the controller for a pipelined design is independent of the pipeline length (the number of time steps) in a scheduled control/data flow graph. The number of states required is equal to the length of the initiation interval in a pipelined design. For example, two states are sufficient to specify the controller for an initiation-interval-2 pipelined design without conditional branches (i.e. no matter what the number of time steps is in the scheduled control/data flow graph).

To realize the controller for a pipelined design without conditional branches, we again define a *bijective* function, $\mathcal{F}_p : \mathcal{E}_{pi} \rightarrow \mathcal{S}_i$ where $i = 1 \ldots I$. The *control signal generating function* $\mathcal{F}_p$ produces a set of control signals associated with the *p-execution* $\mathcal{E}_{pi}$ which is the union of control signals activated in time steps $i, i+I, i+2I, \ldots, i+mI$ where $i+mI \leq n$, and the set of control signals is assigned to state $\mathcal{S}_i$. Finally, by assigning the first state to state $\mathcal{S}_1$, the controller design for a pipelined design without conditional branches is done.

Example: We use the example shown in Figure 3.9 to illustrate this procedure. Assume that the scheduled control/data flow graph shown in Figure 3.9a represents a pipelined design with an initiation interval of 2 time units. The control/data flow graph after being folded is shown in Figure 7.2a. The register-transfer level data path shown in Figure 3.9b implements the function correctly even if the design is pipelined. The controller for this pipelined design is shown in Figure 7.2b. It has been found that the controller for this pipelined design is similar to the controller shown in Figure 3.9c, except that activated control signals of *state1* in Figure 3.9b are the union of activated control signals of *state1* and *state3* in Figure 3.9c. □

7.4 Controllers Using Status Registers

For a design with conditional branches, condition values should be "remembered" by the controller until no more conditional operations are dependent on these condition values. In this section, we present a controller design that uses status registers to store condition values. In general, a controller using status registers generates a simpler state transition diagram than a controller without status registers. However, this may be at the possible cost of increasing chip area. Before going further, we define the following terminology:

Definition 7.2 *The first time step in which the condition value $\mathcal{V}_c$ is created and stored into a register is defined as $\mathcal{B}_v$ where $\mathcal{B}_v \geq 1$, and the condition value $\mathcal{V}_c$ is said to be* born *in time step $\mathcal{B}_v$.*

Definition 7.3 *The last time step in which the execution of operations are dependent on the condition value $\mathcal{V}_c$ is defined as $\mathcal{D}_v$ where $\mathcal{D}_v \geq \mathcal{B}_v$, and the condition value $\mathcal{V}_c$ is said to be* dead *in time step $\mathcal{D}_v$.*

Definition 7.4 *The* lifetime *of the condition value $\mathcal{V}_c$ is defined as a span of time from time step $\mathcal{B}_v + 1$ to time step $\mathcal{D}_v$. The condition value $\mathcal{V}_c$ is said to be* alive *within the lifetime.*

Definition 7.5 *The* reserved period *of the condition value $\mathcal{V}_c$ is defined as a span of time from time step $\mathcal{B}_v + 2$ to time step $\mathcal{D}_v$ if $(\mathcal{B}_v + 1) < \mathcal{D}_v$. Otherwise, the reserved period of the condition value $\mathcal{V}_c$ is defined as* NIL. *The condition value $\mathcal{V}_c$ is said to be* reserved *within the reserved period.*

Note that the death of a condition value is determined not only by the schedule but also by the module allocation and bindings. The death of a condition value can be obtained correctly from the *condition-value lifetime analysis.* The condition-value lifetime analysis inspects all control signal tuples and the death of the condition value $\mathcal{V}_c$ is the last time step in which the activation of a control signal is dependent it. It is obvious that a condition value $\mathcal{V}_c$ must be "remembered" by the controller if and only if the reserved period of the condition value $\mathcal{V}_c$ is not NIL. The number of status registers required to implement a controller using status registers is no less than the number of condition values reserved in any clock cycle during execution.

7.4.1 Controllers for Non-Pipelined Designs

For an n-time-step non-pipelined design controller using status registers, a *ring state transition diagram* $\mathcal{R}_n$ is sufficient to specify the controller behavior because status registers are used to "remember" reserved condition values. The controller specification for a state in which no condition value is alive is the same as the controller specification for a design without conditional branches (i.e. a set of activated control signals followed by the next state assignment). Figure 7.3a shows a typical controller specification for a state without conditional branches.

For the state with only one condition value alive, an *if-else* statement is used to specify the conditional branch. Otherwise, a *case* statement is used to specify a state with more than one condition value alive. Figures 7.3b and 7.3c show typical controller specifications for states with one and more than one condition values alive, respectively. Note that a state with k condition values alive has at most 2^k possible conditional branches.

We determine the status register assignment of reserved condition values using the Left Edge algorithm that is also used for register assignment during data path

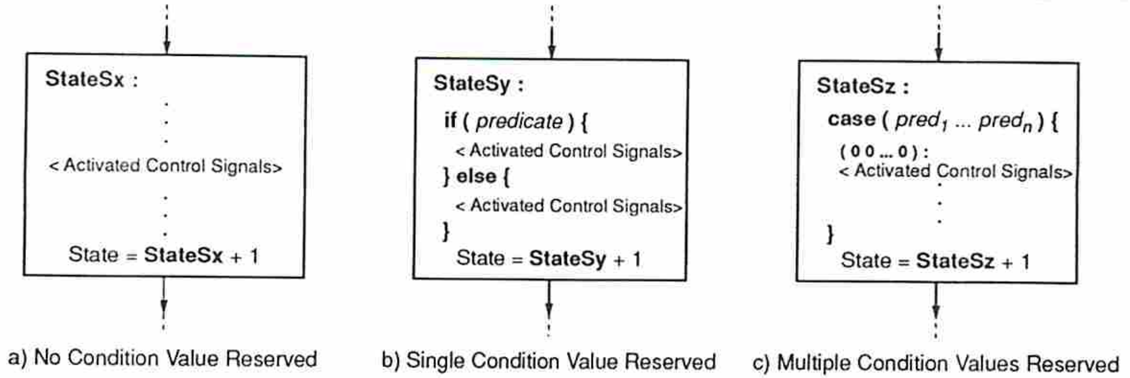


Figure 7.3: Control Specifications Using Status Registers

synthesis [KP87]. The following procedure assigns a reserved condition value to a non-conflicting status register for every time step.

Procedure 1

> *status_register_queue* ← $\emptyset$.
> for time step i from 1 to the last time step **do**
> begin
>   *status_register_queue* ← *allocated status registers.*
>   for the condition value $\mathcal{V}_c$ reserved in time step i **do**
>   begin
>   if *status_register_queue* $\neq \emptyset$
>   remove a status register from *status_register_queue* and assign it to the condition value $\mathcal{V}_c$ for time step i.
>   else
>   allocate a new status register and assign it to the condition value $\mathcal{V}_c$ for time step i.
>   end
> end □

Procedure 3 synthesizes the controller for a non-pipelined design using status registers by calling Procedures 2 and 1. Procedure 2 generates the control signals for each time step using results from the condition-value lifetime analysis and Procedure 1. Obviously, the controller specification produced by Procedure 3 is

a *ring state transition diagram* $\mathcal{R}_n$ for an n-time-step non-pipeline design using status registers.

Procedure 2

```
/* Subroutine to generate the controller specification for time step i. */
if no condition value is alive in time step i
begin
   print out activated control signals in time step i.
   return
end
/* Otherwise, at least one condition value is alive. */
if only one condition value V_c is alive in time step i
begin
   /* Use an "if-else" statement for the single condition value alive cases */
   if the condition value V_c is not reserved
      /* V_c is a newly born condition value */
      use the external input register value as the condition value.
   else
      use the assigned status-register value as the condition value.
   print out activated control signals in time step i which satisfy the
condition
   value alive is "TRUE" for the "if" part first.
   /* No condition value must propagate in the last time step */
   if time step i is not the last time step
   begin
      /* Propagate the reserved condition value. */
      if the condition value V_c is reserved in time step i + 1
         assign "1" to the status register assigned to the condition value V_c in
         time step i + 1.
   end
   /* The "if" part is done here. */
   print out activated control signals in time step i which satisfy the
condition
```

```
   value alive is "FALSE" for the "else" part.
   if time step i is not the last time step
      if the condition value V_c is reserved in time step i + 1
         assign "0" to the status register assigned to the condition value V_c in
         time step i + 1.
   return
end
/* For multiple condition values alive, we use a "case" statement. */
for each condition value V_j is alive in time step i do
begin
   if the condition value V_j is not reserved
      /* V_c is a newly born condition value */
      use the external input register value as the condition value.
   else
      use the assigned status-register value as the condition value.
end
for each possible condition value combination P_k in time step i do
begin
   print out activated control signals in time step i which satisfy the
condition
   value combination P_k.
   if time step i is not the last time step
   begin
      /* Propagate the reserved condition value. */
      for each condition value V_j reserved in time step i + 1 do
      begin
         assign "0"/"1" to the status register assigned to the condition value
         V_c in time step i + 1 using the condition value combination P_k.
      end
   end
end
return
```
□

Procedure 3

generate all possible control signal tuples first.

perform the condition-value lifetime analysis to determine the lifetime and reserved period for each condition value.

execute Procedure 1 to assign status registers for reserved condition values.

/ Now, generate the controller specification for each time step. */*

for time step i from 1 to the last time step **do**

begin

use Procedure 2 to generate controller specification for time step i.

if time step i is the last time step

print the next state transition to state $\mathcal{S}_1$.

else

print the next state transition to state $\mathcal{S}_{i+1}$.

end

□

7.4.2 Controllers for Pipelined Designs

The controller design for a pipelined design using status registers is similar to the design for a non-pipelined design using status registers. The idea of a *folded* scheduled control/data flow graph is used to model a pipelined design. We view the folded control/data flow graph as a set of *p-executions.* The function of the controller for an initiation-interval-I pipelined design is to issue respective control signals that are sufficient to correctly execute operations in the *p-execution* $\mathcal{E}_{pi}$, where $i = 1, \ldots, I$, $t = mI + i$ and m is a non-negative integer.

A *ring state transition diagram* $\mathcal{R}_I$ is sufficient to specify a controller for a pipelined design with an initiation interval I even though there are conditional branches, if status registers are used. The set of activated control signals for state $\mathcal{S}_i$ is the union of control signals activated in time step $i, i + I, i + 2I, \ldots, i + mI$

where $i + mI \leq n$, which is derived in a similar manner to the controller for pipelined designs without conditional branches. A simple example illustrates this procedure.

Example: The scheduled control/data flow graph shown in Figure 3.10a is assumed to be a pipelined design with an initiation interval of 2 time units. The register-transfer level design for the pipelined design is the same as the one for the non-pipelined design shown in Figure 3.10b. The folded control/data flow graph and controller are shown in Figures 7.4a and 7.4b, respectively. Note that the register-transfer level data path is the same as the non-pipelined one, so activated control signals of *state*1 in this example are simply the union of activated control signals of *state*1 and *state*3 in Figure 3.11b. Similarly, activated control signals of *state*2 in Figure 7.4b are the union of activated control signals of *state*2 and *state*4 in Figure 3.11. □

Since reserved condition values are stored in status registers, the controller specification associated with each time step can be viewed as an individual block. Activated control signals of a state in the controller for a pipelined design are actually the union of activated control signals of overlapped time steps. However, due to the overlapped execution with time steps in a pipelined design, a condition value is distinguished by being associated with its working time step. A condition value with two instances reserved in two different time steps should be separately stored in two status registers. A new status register assignment procedure for reserved condition values is thus needed, as described below in Procedure 4.

Procedure 4

> *status_register_queue* ← $\emptyset$.
> **for** state l from 1 to the initiation interval **do**
> **begin**
>   **for** time step i from l to the last time step **step** initiation interval **do**
>   **begin**
>   *status_register_queue* ← *allocated status registers*.
>   **for** the condition value $\mathcal{V}_c$ reserved in time step i **do**

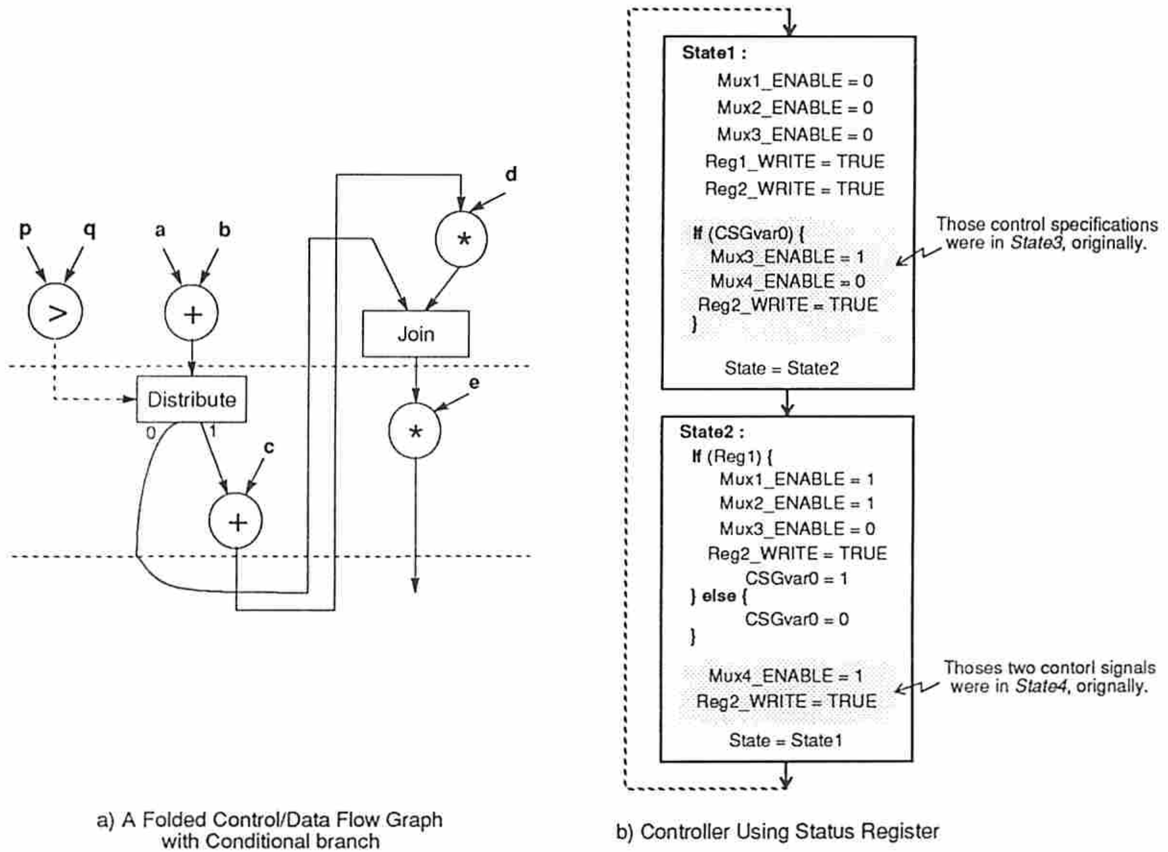


Figure 7.4: A Pipelined Design Synthesis Example with Conditional Branches

begin

if *status_register_queue* $\neq \emptyset$

remove a status register from *status_register_queue* and assign it to the condition value $\mathcal{V}_c$ for time step i.

else

allocate a new status register and assign it to the condition value $\mathcal{V}_c$ for time step i.

end

end

end □

Procedure 4 produces non-conflicting status register assignments for reserved condition values in a pipelined design. The following procedure modified from Procedure 3 generates the controller for a pipelined design.

Procedure 5

generate all possible control signal tuples first.

perform the condition-value lifetime analysis to determine the lifetime and reserved period for each condition value.

execute Procedure 4 to assign status registers for reserved condition values.

/* *Now, generate the controller specification for each state.* */

for state l from 1 to the initiation interval **do**

begin

for time step i from l to the last time step **step** initiation interval **do**

begin

use Procedure 2 to generate the controller specification for time step i.

if state l is equal to initiation interval

print the next state transition to state $\mathcal{S}_1$.

else

print the next state transition to state $\mathcal{S}_{l+1}$.

end

end

□

7.5 Controllers without Status Registers

We now present a controller design that uses state memory to store reserved condition values in a design with conditional branches. As compared to the controller using status registers, the controller without status registers usually has a more complicated state transition diagram. To realize a state transition diagram, three subtasks need to be determined, namely, the number of states required, next state transitions of a given state and activated control signals in a given state. In the implementation of a controller without status registers, the number of states required to specify the controller is dependent on the lifetimes of condition values. Possible next state transitions of a state are determined by the status of reserved condition values (born, reserved or dead). Obviously, more work is needed to design the controller without status registers. However, this type of controllers does not usually produce smaller PLA circuits than controllers using status registers from our experiments.

7.5.1 Controllers for Non-Pipelined Designs

For a design with conditional branches, a condition value should be "remembered" by the controller for the reserved period. The reserved period and the birth and death time steps of a condition value are derived from the condition-value lifetime analysis which is identical to the one performed in designing the controller using status registers. Some interesting characteristics useful for synthesizing a controller without status registers are found from results of the condition-value lifetime analysis:

- For an n-time-step non-pipelined design, at most 2^k states are required to specify the controller behavior for any time step i if there are k condition values reserved in time step i, where $1 \leq i \leq n$.

- A state $\mathcal{S}_{i,p}$ in time step i has exactly one possible next state transition $\mathcal{S}_{i+1,q}$ in time step $i+1$, if there is no condition value born in time step i or condition values born in time step i are also dead in time step i. It should be noted that if i is equal to n then time step 1 is next instead of time step $i+1$.

- A state $\mathcal{S}_{i,j}$ in time step i has two possible next state transitions, $\mathcal{S}_{i+1,p}$ and $\mathcal{S}_{i+1,q}$, in time step $i+1$ if and only if there is one condition value $\mathcal{V}_c$ which is born in time step i and reserved in time step $i+1$. Also note that if i is equal to n then time step 1 is next instead of time step $i+1$.

- A state $\mathcal{S}_{i,j}$ in time step i has at most 2^k possible next state transitions in time step $i+1$ if there are k condition values born in time step i and reserved in time step $i+1$. Note that if i is equal to n then time step 1 is next instead of time step $i+1$.

Lemma 7.3 *Only one state is sufficient to specify the controller behavior in time step 1 for a non-pipelined design.*

Proof: For a non-pipelined design without conditional branches, one state is sufficient to represent the controller behavior for a time step in a scheduled control/data flow graph (Lemma 7.1). Thus, one state is sufficient to represent the controller behavior in time step step 1 for a non-pipelined design without conditional branches.

For a non-pipelined design with conditional branches, a set of states is used to represent the controller behavior for a time step in a scheduled control/data flow graph. Since only reserved condition values are required to be stored by the controller, the number of states required to represent the controller behavior for a time step are dependent on the number of condition values reserved in that time step. The upper bound of states required to represent a time step with k reserved condition values is 2^k states. However, no condition value is reserved in time step 1 from the definition of the reserved condition value, so one state is also sufficient

to represent the controller behavior in time step 1 for a non-pipelined design with conditional branches. □

Other characteristics such as one state and two states transiting to a state, and the upper bound of states transiting to a state could also be obtained in a similar manner:

- A state $\mathcal{S}_{i+1,q}$ in time step $i+1$ has exactly one state $\mathcal{S}_{i,p}$ in time step i transiting to it if no condition values die in time step $i+1$, where $1 \leq i \leq n-1$ for an n-time-step non-pipelined design.

- Two states, $\mathcal{S}_{i,p}$ and $\mathcal{S}_{i,q}$, in time step i have the same next state transition $\mathcal{S}_{i+1,j}$ in time step $i+1$ if and only if there is one condition value $\mathcal{V}_c$ that is reserved and dies in time step i, where $1 \leq i \leq n-1$ for an n-time-step non-pipelined design.

- At most 2^k states in time step i have the same next state transition $\mathcal{S}_{i+1,j}$ in time step $i+1$ if there are k condition values that are reserved and die in time step i, where $1 \leq i \leq n-1$ for an n-time-step non-pipelined design.

Activated control signals in a given state are obtained by excluding non-activated control signals from the set of all possible control signal tuples, according to the condition value combination assigned to the given state and the working time step. The following procedures synthesize the controller for a non-pipelined design without status registers. Procedure 6 produces the controller specification for a state based on the status of condition values (born, reserved or dead). Procedure 7 generates a set of *states* which is sufficient to represent the controller for a given time step, and assigns an unique condition value combination to each state generated. Next state transitions are then determined using the above characteristics. Finally, assigning the first state to the state corresponding to time step 1 completes the controller design. Note that only one state is needed to specify the controller behavior in time step 1 from Lemma 7.3.

Procedure 6

```
/* Subroutine to create the controller specification for state S_{i,j} */
if no condition value is born in time step i
begin
   print out activated control signals in time step i.
   if time step i is not the last time step
   begin
      let the condition value combination P'_{i,j} equal to P_{i,j}.
      remove condition values which die in time step i from P'_{i,j}.
      use state S_{i+1,p} in time step i + 1 with the same condition value
      combination as P'_{i,j} to be the next state transition.
   end else
      use State1 in time step 1 as the next state transition.
   return
end
/* Otherwise, at least one condition value is born */
if only one condition value V_c is born in time step i
begin
   /* Use an "if-else" statement for single condition value born cases */
   print out activated control signals in time step i which satisfy the new
   born condition value V_c = "TRUE" and the condition value combination
   P_{i,j} for the "if" part first.
   if time step i is not the last time step
   begin
      let the condition value combination P'_{i,j} equal to P_{i,j}.
      add the new born condition value V_c = 1 to P'_{i,j}.
      remove condition values which die in time step i from P'_{i,j}.
      use state S_{i+1,p} in time step i + 1 with the same condition value
      combination as P'_{i,j} to be the next state transition.
   end else
      use State1 in time step 1 as the next state transition.
   /* Thus, the "if" part is done */
```

print out activated control signals in time step i which satisfy the new born condition value $\mathcal{V}_c$ = "FALSE" and the condition value combination $\mathcal{P}_{i,j}$ for the "else" part.

if time step i is **not** the last time step

begin

let the condition value combination $\mathcal{P}'_{i,j}$ equal to $\mathcal{P}_{i,j}$.

add the new born condition value $\mathcal{V}_c = 0$ to $\mathcal{P}'_{i,j}$.

remove condition values which die in time step i from $\mathcal{P}'_{i,j}$.

use state $\mathcal{S}_{i+1,p}$ in time step $i+1$ with the same condition value combination as $\mathcal{P}'_{i,j}$ to be the next state transition.

end else

use $State1$ in time step 1 as the next state transition.

return

end

/* *For multiple newly born condition values, we use a "case" statement* */

for each newly born condition value combination $\mathcal{P}_{bi}$ in time step i **do**

begin

print out activated control signals in time step i which satisfy condition value combinations $\mathcal{P}_{bi}$ and $\mathcal{P}_{i,j}$.

if time step i is **not** the last time step

begin

let the condition value combination $\mathcal{P}'_{i,j}$ equal to $\mathcal{P}_{i,j}$.

add the new born condition value combination $\mathcal{P}_{bi}$ to $\mathcal{P}'_{i,j}$.

remove condition values which die in time step i from $\mathcal{P}'_{i,j}$.

use state $\mathcal{S}_{i+1,p}$ in time step $i+1$ with the same condition value combination as $\mathcal{P}'_{i,j}$ to be the next state transition.

end else

use $State1$ in time step 1 as the next state transition.

end

return □

Procedure 7

/* *Generate a set of states $\mathcal{S}_i^*$ for each time step* */
for time step i from 1 to the last time step **do**
begin
generate a set of states, $\mathcal{S}_i^* = \{\mathcal{S}_{i,1}, \mathcal{S}_{i,2}, \ldots, \mathcal{S}_{i,2^{k_i}}\}$, where k_i is the number of condition values reserved in time step i.
for condition value combination $\mathcal{P}_{i,j}$ from $\mathcal{P}_{i,1}$ to $\mathcal{P}_{i,2^{k_i}}$ in time step i **do**
assign state $\mathcal{S}_{i,j}$ to the condition value combination $\mathcal{P}_{i,j}$.
end
/* *Now, produce the controller specification for each generated state* */
for time step i from 1 to the last time step **do**
for each state $\mathcal{S}_{i,j} \in \mathcal{S}_i^*$ in time step i **do**
use Procedure 6 to generate the controller specification for state $\mathcal{S}_{i,j}$.

□

7.5.2 Controllers for Pipelined Designs

The controller design for a pipelined design without status registers is analogous to the design for a non-pipelined design without status registers. A *folded* scheduled control/data flow graph is used to model the execution of a pipelined design. The task of the controller in a pipelined design with an initiation interval I is to produce control signals that carry out the execution of operations in the *p-execution* $\mathcal{E}_{pi}$ at clock cycle $t = mI + i$, where $i = 1, \ldots, I$ and m is a non-negative integer. An approach similar to the approach that we used in the controller design for a non-pipelined design without status registers is applied here. A set of states which is sufficient to specify the controller behavior is first created. Next state transitions of the states created are then determined. Activated control signals in a given state are finally produced to complete the controller design.

For an initiation-interval-I pipelined design, the control behavior of a folded time step is repeated every I clock cycles. I sets of states, $\mathcal{S}_{p1}^*, \mathcal{S}_{p2}^*, \ldots, \mathcal{S}_{pI}^*$, are used to specify the controller for a pipelined design without status registers. In the controller design for a pipelined design using status registers, a condition value

within the reserved period is treated as a different condition value instance in each time step and should be "remembered" distinctly by the controller. Therefore, the instance of a reserved condition value is actually a function of *the condition value itself* as well as *its working time step*. In order to differentiate the instances of a reserved condition value in different time steps, the time step i is added as a second subscript in the condition value $\mathcal{V}_c$, making it $\mathcal{V}_{c,i}$.

In designing a controller without status registers, each possible condition value combination must correspond to a state in the controller in order to accomplish all possible execution cases. Due to the concurrent execution of multiple data sets in a pipelined design and the mutual exclusion of condition values, the set of states of a folded time step can be derived as the *Cartesian products* of the sets of states which are mapped to this folded time step. To define the *Cartesian product* of two sets of states, we first define the *product* of two states. A *product* of two states produces a new state whose condition value combination is the union of the condition value combinations of the two states. The *product* of two states and *Cartesian product* of two sets of states are defined as follows.

Definition 7.6 *The* product *of two states $\mathcal{S}_{i,p}$ and $\mathcal{S}_{j,q}$ is a new state $\mathcal{S}_{i\times j,r}$ whose condition value combination is the union of condition value combinations of $\mathcal{P}_{i,p}$ and $\mathcal{P}_{j,q}$ associated with states $\mathcal{S}_{i,p}$ and $\mathcal{S}_{j,q}$, respectively, where index $r = (p - 1)|\mathcal{S}_j^*| + q$.*

Definition 7.7 *The* Cartesian product *of two sets of states*

$$\mathcal{S}_i^* = \{\mathcal{S}_{i,1}, \mathcal{S}_{i,2}, \ldots, \mathcal{S}_{i,p}\}$$

and

$$\mathcal{S}_j^* = \{\mathcal{S}_{j,1}, \mathcal{S}_{j,2}, \ldots, \mathcal{S}_{j,q}\}$$

is a new set of states

$$\mathcal{S}_{i\times j}^* = \{\mathcal{S}_{i\times j,1}, \mathcal{S}_{i\times j,2}, \ldots, \mathcal{S}_{i\times j,pq}\}$$

where $\mathcal{S}_{i\times j,k}$ is the product of two states $\mathcal{S}_{i,\lfloor \frac{k}{q} \rfloor+1}$ and $\mathcal{S}_{j,(k \bmod q)+1}$, $1 \leq k \leq pq$.

Example: The Cartesian product of the sets of states

$$\mathcal{S}_2^* = \{\mathcal{S}_{2,1} \leftarrow (\mathcal{V}_{1,2} : 1), \mathcal{S}_{2,2} \leftarrow (\mathcal{V}_{1,2} : 0)\}$$

and

$$\mathcal{S}_4^* = \{\mathcal{S}_{4,1} \leftarrow (\mathcal{V}_{1,4} : 1, \mathcal{V}_{2,4} : 1), \mathcal{S}_{4,2} \leftarrow (\mathcal{V}_{1,4} : 1, \mathcal{V}_{2,4} : 0), \mathcal{S}_{4,3} \leftarrow (\mathcal{V}_{1,4} : 0, \mathcal{V}_{2,4} : 1)\}$$

is a new set of states

$$\begin{aligned} \mathcal{S}_{2\times 4}^* \quad = \quad & \{\mathcal{S}_{2\times 4,1} \leftarrow (\mathcal{V}_{1,2} : 1, \mathcal{V}_{1,4} : 1, \mathcal{V}_{2,4} : 1), \mathcal{S}_{2\times 4,2} \leftarrow (\mathcal{V}_{1,2} : 1, \mathcal{V}_{1,4} : 1, \mathcal{V}_{2,4} : 0), \\ & \mathcal{S}_{2\times 4,3} \leftarrow (\mathcal{V}_{1,2} : 1, \mathcal{V}_{1,4} : 0, \mathcal{V}_{2,4} : 1), \mathcal{S}_{2\times 4,4} \leftarrow (\mathcal{V}_{1,2} : 0, \mathcal{V}_{1,4} : 1, \mathcal{V}_{2,4} : 1), \\ & \mathcal{S}_{2\times 4,5} \leftarrow (\mathcal{V}_{1,2} : 0, \mathcal{V}_{1,4} : 1, \mathcal{V}_{2,4} : 0), \mathcal{S}_{2\times 4,6} \leftarrow (\mathcal{V}_{1,2} : 0, \mathcal{V}_{1,4} : 0, \mathcal{V}_{2,4} : 1)\}. \end{aligned}$$

□

In the controller design using status registers, the controller uses status registers to store reserved condition values. However, in the controller design without status registers, different states are used to distinguish and "remember" different instances of a reserved condition value. For an n-time-step pipelined design, the set of states $\mathcal{S}_{pi}^*$ required to specify the controller without status registers in a folded time step t can be derived by Cartesian products of the sets of states $\mathcal{S}_i^*, \mathcal{S}_{i+I}^*, \ldots, \mathcal{S}_{i+mI}^*$, where $i + mI \leq n$, $t = i + m'I$ and m' is a non-negative integer. The following theorem proves this statement.

Theorem 7.1 *The set of states $\mathcal{S}_{pi}^*$ produced by Cartesian products of the sets of states*

$$\mathcal{S}_i^*, \mathcal{S}_{i+I}^*, \ldots, \mathcal{S}_{i+mI}^*$$

is sufficient to specify the control behavior for an initiation-interval-I pipelined design in the folded time step t, where $t = i + m'I$, $i + mI \leq n$, n is the pipeline length and m, m' are non-negative integers, if the set of states $\mathcal{S}_j$ is sufficient to specify the control behavior in time step j, where $j = i, i + I, \ldots, i + mI$.

Proof: In the controller design without status registers for a pipelined design with an initiation interval I, a set of states $\mathcal{S}_{pi}^*$, which is sufficient to represent the control behavior in a given folded time step t, should be able to represent all possible condition value combinations for time step $i, i+I, \ldots, i+mI$ in a scheduled control/data flow graph. From the definition of the *Cartesian product* of two sets of states, $\mathcal{S}_i^*$ and $\mathcal{S}_j^*$, the *Cartesian product* $\mathcal{S}_{i \times j}^*$ includes all possible condition value combinations from these two sets of states $\mathcal{S}_i^*$ and $\mathcal{S}_j^*$. Since the set of states $\mathcal{S}_{pi}^*$ is produced by *Cartesian products* of $\mathcal{S}_i^*, \mathcal{S}_{i+I}^*, \ldots, \mathcal{S}_{i+mI}^*$, and the sets of states $\mathcal{S}_i^*, \mathcal{S}_{i+I}^*, \ldots, \mathcal{S}_{i+mI}^*$ are sufficient to specify the control behavior in time step $i, i+I, \ldots, i+mI$, respectively, the set of states $\mathcal{S}_{pi}^*$ is sufficient to represent the control behavior in the folded time step t for an initiation-interval-I pipelined design. □

To determine next-state transitions, the characteristics used to determine next state transitions for a non-pipelined design controller without status registers are applied here. Note that due to the use of the *folded* control/data flow graph, a time step in a non-pipelined design controller is now interpreted as a folded time step in a pipelined design controller which generally includes a set of time steps being executed at the same clock cycle.

Activated control signals for a given state are derived in a similar manner as the non-pipelined design controller without status registers. Finally, arbitrarily assigning one of the states $\mathcal{S}_{p1,i} \in \mathcal{S}_{p1}^*$ in the folded time step 1 as the first execution state completes the design of the controller without status registers. Note that any compound state in the set of states $\mathcal{S}_{p1}^*$ corresponding to the folded time step 1 is sufficient to specify the control behavior in the first execution clock cycle. This arises from the proof of Theorem 7.1. We now use a simple pipelined example to illustrate the design process of the controller.

Example: In the example shown in Figure 7.4, one state is required for time step 1, 2 and 4 and two states are used for time step 3 before the scheduled control/data flow graph is folded. Note that "$\leftarrow$ ()" after each state shows the possible condition value combinations associated with this state. `NIL` means that no condition value

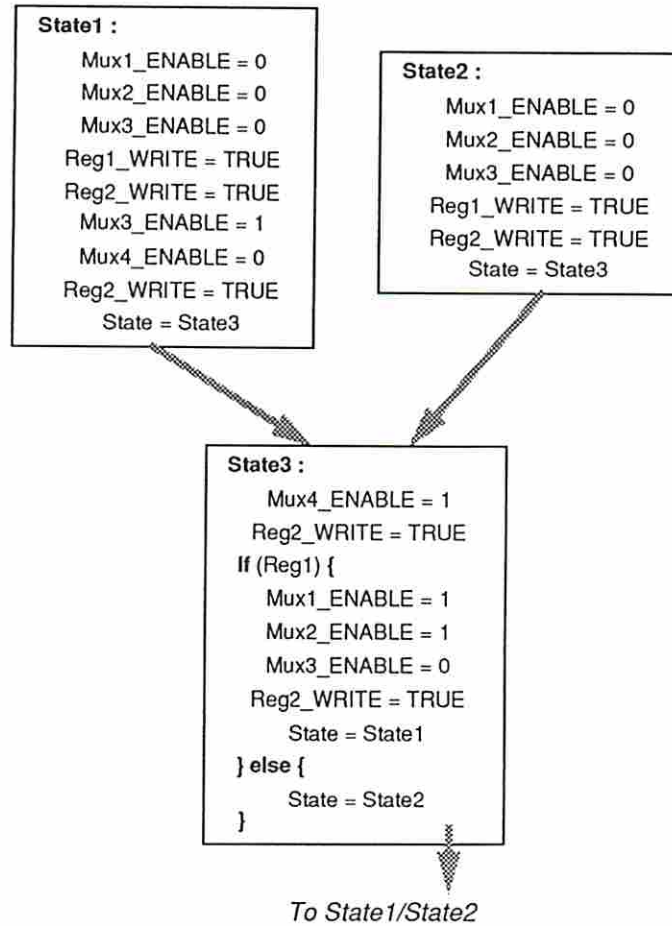


Figure 7.5: The Controller for the Pipelined Example Shown in Figure 7.4

combination is associated with this state. The sets of states for time steps before the control/data flow graph is folded are

$$\begin{aligned}
\mathcal{S}_1^* &= \{State1 \leftarrow (\texttt{NIL})\} \\
\mathcal{S}_2^* &= \{State2 \leftarrow (\texttt{NIL})\} \\
\mathcal{S}_3^* &= \{State3 \leftarrow (\mathcal{V}_{c,3} : 1), State4 \leftarrow (\mathcal{V}_{c,3} : 0)\} \\
\mathcal{S}_4^* &= \{State5 \leftarrow (\texttt{NIL})\}
\end{aligned}$$

where $\mathcal{V}_c$ represents the condition value of "$p > q$".

For an initiation-interval-2 pipelined design, two sets of states are needed which are derived from Cartesian products of $\mathcal{S}_1^*$, $\mathcal{S}_3^*$ and $\mathcal{S}_2^*$, $\mathcal{S}_4^*$, respectively.

$$\begin{aligned}
\mathcal{S}_{p1}^* &= \mathcal{S}_1^* \times \mathcal{S}_3^* = \{State1 \leftarrow (\mathcal{V}_{c,3} : 1), State2 \leftarrow (\mathcal{V}_{c,3} : 0)\} \\
\mathcal{S}_{p2}^* &= \mathcal{S}_2^* \times \mathcal{S}_4^* = \{State3 \leftarrow \texttt{NIL})\}
\end{aligned}$$

Next state transitions between the two sets of states $\mathcal{S}_{p1}^*$ and $\mathcal{S}_{p2}^*$ are determined in a similar manner as for the non-pipelined design controller without status registers. The complete state transition diagram is shown in Figure 7.5. The first state can be arbitrarily assigned to either $State1$ or $State2$. □

The following procedures synthesize the controller for an initiation-interval-I n-time-step pipelined design without status registers.

Procedure 8

/* *Subroutine to create the controller specification for state* $\mathcal{S}_{pi,j}$ */
if no condition value is born in the folded time step i
begin
print out activated control signals in the folded time step i.
let the condition value combination $\mathcal{P}'_{pi,j}$ equal to $\mathcal{P}_{pi,j}$.
remove condition values which die in the folded time step i from $\mathcal{P}'_{pi,j}$.
increase the time step of condition value instances in $\mathcal{P}'_{pi,j}$ by 1.
if i is **not** equal to the initiation interval I
use state $\mathcal{S}_{pi+1,t} \in \mathcal{S}_{pi+1}^*$ with the same condition value combination as $\mathcal{P}'_{pi,j}$ to be the next state transition.
else
use state $\mathcal{S}_{p1,t} \in \mathcal{S}_{p1}^*$ with the same condition value combination as $\mathcal{P}'_{pi,j}$ to be the next state transition.
return
end
/* *Otherwise, at least one condition value is born* */
if only $\mathcal{V}_{c,s}$ is born in the folded time step i where $s = i + mI$ and $i + mI \leq n$

begin

/ Use an "if-else" statement for single condition value born cases */*

print out activated control signals in the folded time step i which satisfy the new born condition value $\mathcal{V}_{c,s}$ = "TRUE" and the condition value combination $\mathcal{P}_{pi,j}$ for the "if" part first.

let the condition value combination $\mathcal{P}'_{pi,j}$ equal to $\mathcal{P}_{pi,j}$.

add the new born condition value $\mathcal{V}_{c,s} = 1$ to $\mathcal{P}'_{pi,j}$.

remove condition values which die in the folded time step i from $\mathcal{P}'_{pi,j}$.

increase the time step of condition value instances in $\mathcal{P}'_{pi,j}$ by 1.

if i is **not** equal to the initiation interval I

use state $\mathcal{S}_{pi+1,t} \in \mathcal{S}^*_{pi+1}$ with the same condition value combination as $\mathcal{P}'_{pi,j}$ to be the next state transition.

else

use state $\mathcal{S}_{p1,t} \in \mathcal{S}^*_{p1}$ with the same condition value combination as $\mathcal{P}'_{pi,j}$ to be the next state transition.

/ Thus, the "if" part is done */*

print out activated control signals in the folded time step i which satisfy the new born condition value = "FALSE" and the condition value combination $\mathcal{P}_{pi,j}$ for the "else" part.

let the condition value combination $\mathcal{P}'_{pi,j}$ equal to $\mathcal{P}_{pi,j}$.

add the new born condition value $\mathcal{V}_{c,s} = 0$ to $\mathcal{P}'_{pi,j}$.

remove condition values which die in the folded time step i from $\mathcal{P}'_{pi,j}$.

increase the time step of condition value instances in $\mathcal{P}'_{pi,j}$ by 1.

if i is **not** equal to the initiation interval I

use state $\mathcal{S}_{pi+1,t} \in \mathcal{S}^*_{pi+1}$ with the same condition value combination as $\mathcal{P}'_{pi,j}$ to be the next state transition.

else

use state $\mathcal{S}_{p1,t} \in \mathcal{S}^*_{p1}$ with the same condition value combination as $\mathcal{P}'_{pi,j}$ to be the next state transition.

return

end

/ For multiple new born condition values, we use a "case" statement */*

for each new born $\mathcal{P}_{bi}$ in the folded time step i **do**

begin

print out activated control signals in the folded time step i which satisfy condition value combinations $\mathcal{P}_{bi}$ and $\mathcal{P}_{pi,j}$.

let the condition value combination $\mathcal{P}'_{pi,j}$ equal to $\mathcal{P}_{pi,j}$.

add the new born condition value combination $\mathcal{P}_{bi}$ to $\mathcal{P}'_{pi,j}$.

remove condition values which die in the folded time step i from $\mathcal{P}'_{pi,j}$.

increase the time step of the condition value instances in $\mathcal{P}'_{pi,j}$ by 1.

if i is **not** equal to the initiation interval I

use state $\mathcal{S}_{pi+1,t} \in \mathcal{S}^*_{pi+1}$ with the same condition value combination as $\mathcal{P}'_{pi,j}$ to be the next state transition.

else

use state $\mathcal{S}_{p1,t} \in \mathcal{S}^*_{p1}$ with the same condition value combination as $\mathcal{P}'_{pi,j}$ to be the next state transition.

end

return □

Procedure 9

/ Generate a set of states $\mathcal{S}^*_{pi}$ for each folded time step */*

for time step i from 1 to the last time step **do**

begin

generate a set of states, $\mathcal{S}^*_i = \{\mathcal{S}_{i,1}, \mathcal{S}_{i,2}, \ldots, \mathcal{S}_{i,2^{k_i}}\}$, where k_i is the number of condition values reserved in time step i.

for condition value combination $\mathcal{P}_{i,j}$ from $\mathcal{P}_{i,1}$ to $\mathcal{P}_{i,2^{k_i}}$ in time step i **do**

assigned state $\mathcal{S}_{i,j}$ to the condition value combination $\mathcal{P}_{i,j}$.

end

for the set of states $\mathcal{S}^*_{pi}$ from $\mathcal{S}^*_{p1}$ to $\mathcal{S}^*_{pI}$ **do**

$\mathcal{S}^*_{pi} \leftarrow$ *Cartesian products* of the sets of states, $\mathcal{S}^*_i, \mathcal{S}^*_{i+I}, \ldots, \mathcal{S}^*_{i+mI}$, $mI \leq n$.

/ Now, produce the controller specification for each generated state */*

for the set of states $\mathcal{S}^*_{pi}$ from $\mathcal{S}^*_{p1}$ to $\mathcal{S}^*_{pI}$ **do**

for each state $\mathcal{S}_{pi,j} \in \mathcal{S}^*_{pi}$ **do**

use Procedure 8 to generate the controller specification for state $\mathcal{S}_{pi,j}$.

$1.2\mu m$ *Technology*	*Delay* (ns)	*Area* (μm^2)
16-bit Comparator (>)	36	142087
16-bit Adder (+)	34	68825
16-bit Subtractor (−)	36	93646
16-bit Multiplier (*)	48	1838537

Table 7.1: Design Library Set Used by CSG (Obtained from ChipCrafter)

□

7.6 Control Path Experiments and Results

The algorithms presented for the control path synthesis have been implemented in a program called CSG using the C language. The experiments are focused on the comparisons of two different controller implementation styles, e.g. using status registers versus not using status registers. A portion of the robot arm controller example obtained from UC-Berkeley served as our example as shown in Figure 6.28. In this example, there is a total of 46 operations/nodes including 4 distribute-join pairs. The module library used for this experiment is shown in Table 7.1.

7.6.1 Non-Pipelined Design Experiments and Results

For the non-pipelined designs, we created nine designs using MAHA in the current ADAM synthesis system, as shown in Table 7.2. The non-pipelined design schedules generated by MAHA are further processed to create register-transfer level designs by the MABAL program. CSG then takes schedule, bindings and register-transfer level data paths to generate the controller specifications. In order to compare the number of control signals before the Finesse program synthesis and after the duplicated output signals are merged, Table 7.3 lists the number of allocated modules in each design. For each design, both types of controller specifications (with/without status registers) were produced and are shown in Tables 7.4 and 7.5, respectively. These designs are then synthesized using ChipCrafter to obtain the layouts.

Design Name	*Time Steps*	*Execution Delay*[†]*(ns)*	*Area* (μm^2)
MAHA.11	11	528	6193552
MAHA.12	12	576	4473482
MAHA.13	13	624	4237749
MAHA.14	14	672	4168924
MAHA.15	15	720	4050457
MAHA.19	19	912	2472474
MAHA.20	20	960	2378828
MAHA.21	21	1008	2236741
MAHA.22	22	1056	2143095

Table 7.2: Non-Pipelined Robot Arm Controller Examples by MAHA

Non-Pipelined	*Functional Units*				*Interconnection Units*	
Design Name	*Cmp*	*Add*	*Sub*	*Mul*	*Registers*	*Multiplexers*
MAHA.11	2	3	2	3	34	25
MAHA.12	2	2	4	2	34	21
MAHA.13	1	2	3	2	34	20
MAHA.14	1	1	3	2	34	15
MAHA.15	1	2	1	2	34	19
MAHA.19	2	1	3	1	34	14
MAHA.20	2	1	2	1	34	15
MAHA.21	1	1	2	1	34	12
MAHA.22	1	1	1	1	34	12

Table 7.3: Non-Pipelined Robot Arm Controller RTL Examples by MABAL

Figure 7.6 shows a typical controller layout produced by ChipCrafter which includes a PLA component and a set of D-type flip flops. It can be seen from this layout example that the area of the DFFs is almost the same as the PLA area. Wiring also takes a substantial amount of area in a controller. Therefore, in order to estimate the area of a PLA type controller accurately, the area of DFFs and wiring can not be ignored.

[†] The execution delay here includes the delays caused by functional units only.

[‡] Note that the number of states for a controller using status registers are always one more than the number of time steps. This is because the first state in the controller specification is used to latch the input values.

[§] The area of a controller includes the area of PLA, DFFs and wiring.

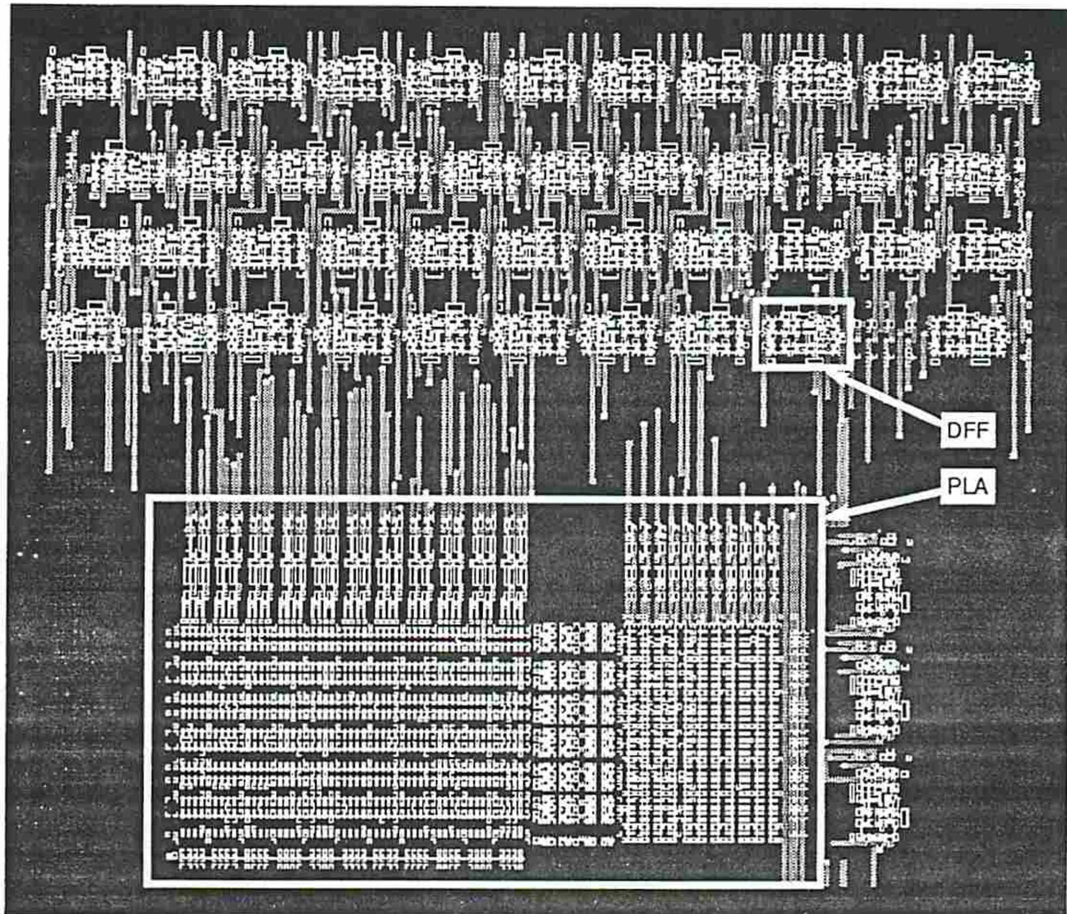


Figure 7.6: A PLA-Based Controller Synthesized by Finesse Program

Controller Specification			
Design Name	*No. of States*[‡]	*Status Registers*	*Outputs*
MAHA.11	12	2	79
MAHA.12	13	2	72
MAHA.13	14	1	76
MAHA.14	15	2	70
MAHA.15	16	2	74
MAHA.19	20	2	63
MAHA.20	21	2	66
MAHA.21	22	2	64
MAHA.22	23	2	63

PLA Implementation							
Design Name	*Rows*	*Ipts*	*Opts*	*PLA Area*	*M. Opts*	*DFFs*	*Area*[§](μm^2)
MAHA.11	17	10	39	153990	45	42	603325
MAHA.12	19	10	37	157954	41	38	575055
MAHA.13	18	9	44	162491	36	52	879981
MAHA.14	22	10	47	206672	29	53	919443
MAHA.15	24	10	48	220070	32	50	755715
MAHA.19	25	11	44	212503	26	45	753246
MAHA.20	26	11	49	244219	24	50	863212
MAHA.21	31	11	49	280061	22	49	858183
MAHA.22	30	11	47	260929	23	48	769165

Table 7.4: Controllers Using Status Registers for Non-Pipelined Examples

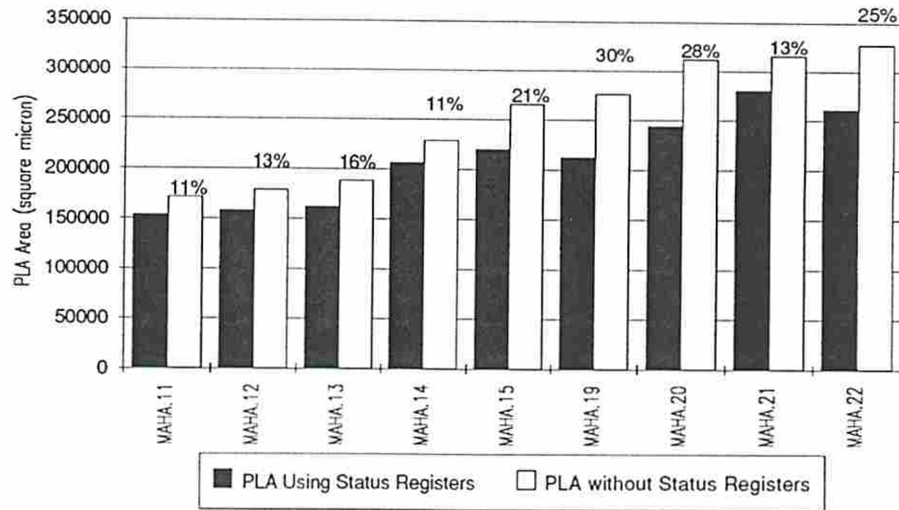


Figure 7.7: PLA Area Comparison of the Non-Pipelined Designs

In the non-pipelined design experiments, the size of the PLAs in controllers increased as the number of time steps increased (see Figure 7.7). This behavior is consistent with our prediction. However, the area of the controllers (see Figure 7.8) increased in the beginning, but then fluctuated with less than 10% variation. This phenomenon may be explained as follows. From Tables 7.4 and 7.5, the number of output signals decreased as the number of time steps increased in the controller specifications. Nevertheless, the number of merged outputs is reduced when the number of time steps is increased. This is probably because more output bits are encoded into a select word for more serialized non-pipelined designs. For example, two 3-to-1 multiplexers (4 control bits) may be replaced by one 6-to-1 multiplexer (3 control bits) in a more serial design. Even though the number of controller outputs after removing the merged outputs, which is similar to the number of PLA outputs, is not increased in the designs with more than 12 time steps, the area of PLAs is still increased. This is probably due to the fact that the numbers of PLA product terms(rows) and state encoding bits(columns) are increased as the number of time steps increases.

Figures 7.7 and 7.8 show the area comparisons of PLAs and control paths for the non-pipelined designs with/without using status registers. It is interesting to note that the controller specified using status registers usually creates a smaller PLA area as well as a smaller total control area for the non-pipelined designs,

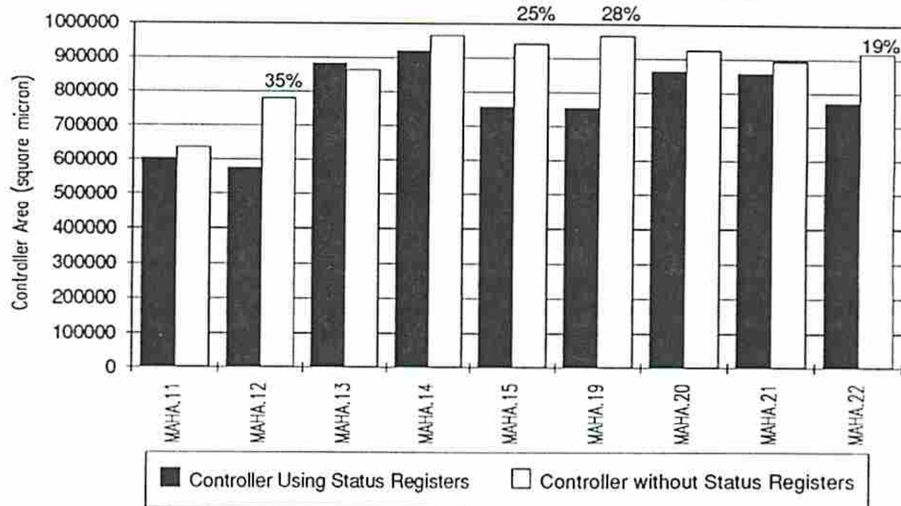


Figure 7.8: Controller Area Comparison of the Non-Pipelined Designs

although it uses more state encoding bits[5] than the one without status registers. In order to understand the reasons for those results, we examine the state transition diagrams for *MAHA.12* in both designs (with/without status registers) after the Finesse[6] synthesis. Notice that a twelve-time-step design needs at least thirteen clock cycles to complete the task because one extra clock cycle is used to latch input values.

Figures 7.9 and 7.10 depict the state transition diagrams for *MAHA.12* without/with status registers, respectively. Even though the two state transition diagrams look completely different, they represent exactly the same controller behavior. By comparing these two state transition diagrams, we found that some of the states in the state transition diagrams without status registers split into several states as compared to the state transition diagram with status registers. All the split states are shown by bold faces and cross hatch patterns in Figures 7.9 and 7.10. The splits of those states indeed indicate that a multiple-code state encoding is produced in the controller using status registers. Apparently, *Espresso* is able to exploit the don't-care bits in state codes in order to minimize the size of the binary

[5]The state encoding bits here include the bit to represent states and the bits used by status registers.

[6]Finesse is a finite state machine synthesis program in Cascade Design Automation ChipCrafter.

Controller Specification		
Design Name	*No. of States*	*Outputs*
MAHA.11	21	79
MAHA.12	25	72
MAHA.13	19	76
MAHA.14	25	70
MAHA.15	30	74
MAHA.19	40	63
MAHA.20	41	66
MAHA.21	41	64
MAHA.22	42	63

PLA Implementation							
Design Name	*Rows*	*Ipts*	*Opts*	*PLA Area*	*M. Opts*	*DFFs*	*Area* (μm^2)
MAHA.11	21	9	40	171527	44	40	635578
MAHA.12	25	9	36	179181	41	37	778786
MAHA.13	19	9	47	188595	34	50	863749
MAHA.14	25	9	48	228518	27	50	964412
MAHA.15	29	9	50	265571	29	51	942046
MAHA.19	40	10	42	275989	27	43	965367
MAHA.20	39	10	46	311978	26	49	923976
MAHA.21	39	10	47	315747	23	48	893265
MAHA.22	41	10	46	327286	23	47	915990

Table 7.5: Controllers without Status Registers for Non-Pipelined Examples

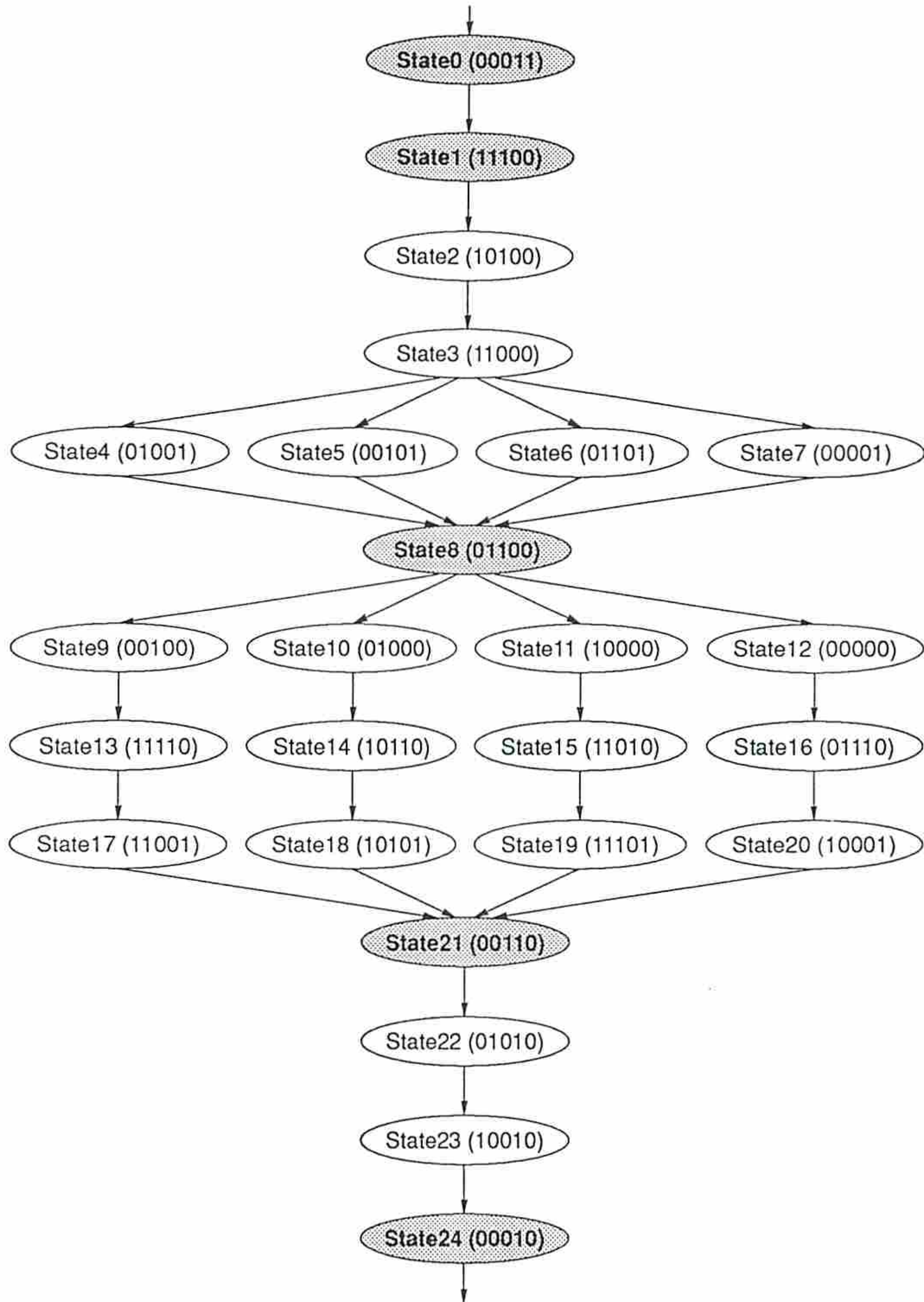


Figure 7.9: *MAHA.12* State Transition Diagram without Status Registers

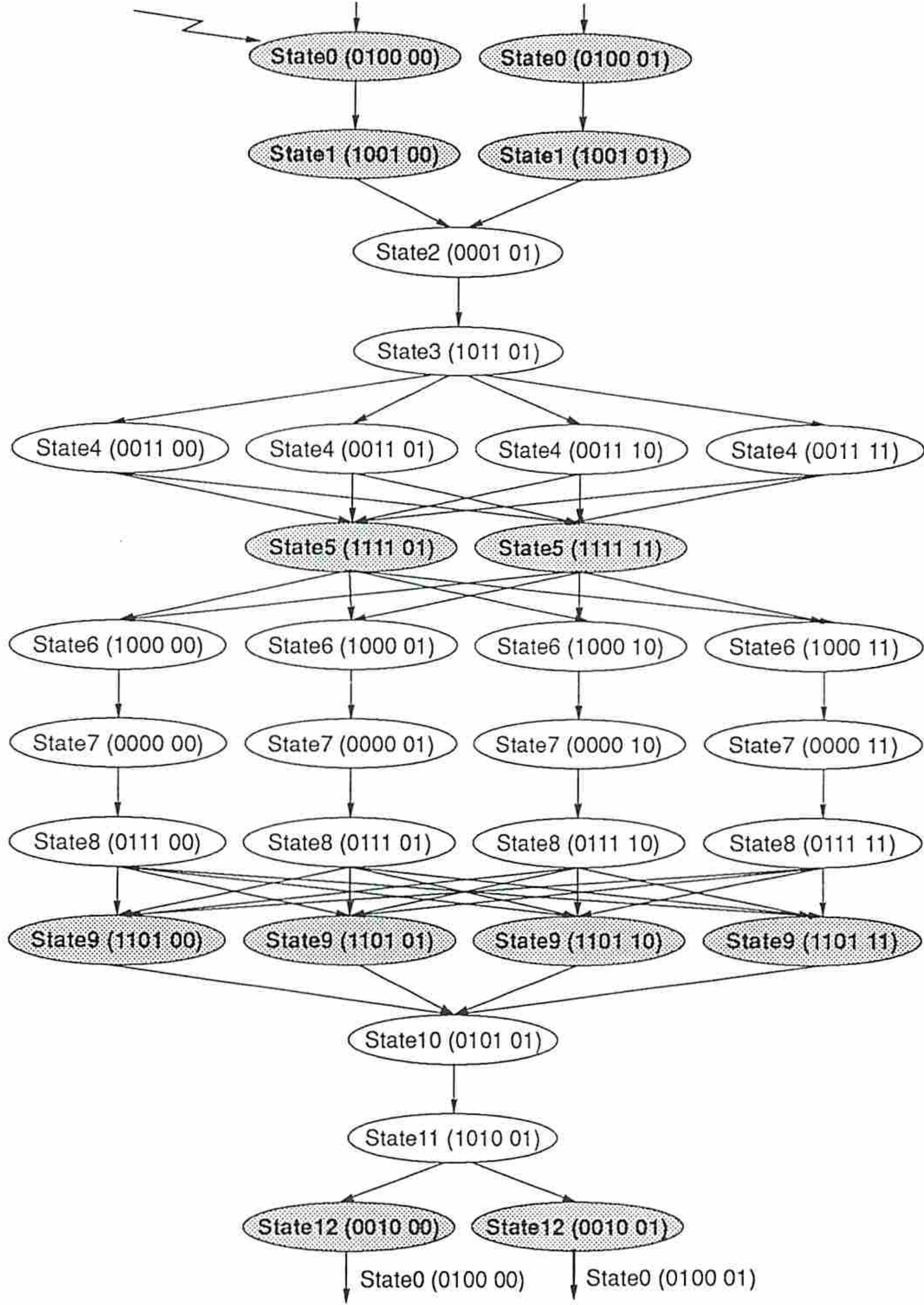


Figure 7.10: *MAHA.12* State Transition Diagram Using Status Registers

Design Name	*Init. Intvl*	*Pipeline Length*	*Init. Delay**(ns)*	*Area* (μm^2)
Sehwa.15.3	3	15	144	11777988
Sehwa.17.3	3	17	144	11777988
Sehwa.13.4	4	13	192	9728539
Sehwa.15.4	4	15	192	9728539
Sehwa.11.5	5	11	240	7890002
Sehwa.12.5	5	12	240	7890002
Sehwa.19.6	6	19	288	5888994
Sehwa.24.6	6	24	288	5888994
Sehwa.15.9	9	15	432	4050457
Sehwa.16.9	9	16	432	4050457
Sehwa.14.11	11	14	528	3981632
Sehwa.20.11	11	20	528	3981632
Sehwa.20.17	17	20	816	2143095
Sehwa.27.17	17	27	816	2143095

Table 7.6: Pipelined Robot Arm Controller Examples by Sehwa

encoded cover. For example, no status registers are used in *State0* for the controller using status registers so, in Finesse, the two don't-care bits and "0100" are assigned by the unicode state encoding algorithm. To reduce the product terms, *Espresso* assigned two don't-care bits to "00" and "01" and the state encodings of *State0* became "0100 00" and "0100 01" (a multiple-code state encoding). Note that most current state encoding algorithms can produce unicode encodings only. Our heuristic functionally separates the encoding task between states and condition values, which make the unicode state encoding algorithm produce a multiple-code state encoding.

7.6.2 Pipelined Design Experiments and Results

Fourteen pipelined designs with different initiation intervals and/or pipeline lengths were created by Sehwa as shown in Table 7.6. The respective register-transfer level designs produced by MABAL are shown in Table 7.7. Both controller styles are generated for all the designs which are shown in Tables 7.8 and 7.9.

**The initiation interval here includes the delays caused by functional units only.

Pipelined Design Name	*Functional Units*				*Interconnection Units*	
	Cmp	*Add*	*Sub*	*Mul*	*Registers*	*Multiplexers*
Sehwa.15.3	2	4	2	6	110	40
Sehwa.17.3	2	4	2	6	110	36
Sehwa.13.4	1	3	2	5	84	28
Sehwa.15.4	1	3	2	5	76	35
Sehwa.11.5	1	3	2	4	55	26
Sehwa.12.5	1	3	2	4	60	29
Sehwa.19.6	1	2	1	3	71	20
Sehwa.24.6	1	2	1	3	92	26
Sehwa.15.9	1	2	1	2	42	22
Sehwa.16.9	1	2	1	2	45	22
Sehwa.14.11	1	1	1	2	35	19
Sehwa.20.11	1	1	1	2	41	20
Sehwa.20.17	1	1	1	1	36	15
Sehwa.27.17	1	1	1	1	42	16

Table 7.7: Pipelined Robot Arm Controller RTL Examples by MABAL

For simplicity, a comparison between two implementation styles, the comparison bar charts for the PLA area and complete controller area are depicted in Figures 7.11 and 7.12, respectively. The results are slightly different than the non-pipelined designs. In the pipelined designs, the designs that used status registers were found larger than those not using the status registers. This may be due to the high utilization of status registers which leaves *Espresso* with little or no freedom to assign the don't-care bits to status registers. However, the number of cases that improved are greater than the number of cases that are degraded in our experiments. Notice that the percentage of area improvements on both the PLA area and total controller area is more dramatic than for non-pipelined designs, and the area increase percentage of the controller using the status registers is much less than the one not using the status registers.

Controller Specification			
Design Name	*No. of States*	*Status Registers*	*Outputs*
Sehwa.15.3	3	4	170
Sehwa.17.3	3	2	168
Sehwa.13.4	4	3	131
Sehwa.15.4	4	1	137
Sehwa.11.5	5	1	101
Sehwa.12.5	5	2	109
Sehwa.19.6	6	1	115
Sehwa.24.6	6	4	145
Sehwa.15.9	9	1	88
Sehwa.16.9	9	2	90
Sehwa.14.11	11	1	76
Sehwa.20.11	11	2	87
Sehwa.20.17	17	2	70
Sehwa.27.17	17	2	77

PLA Implementation							
Design Name	*Rows*	*Ipts*	*Opts*	*PLA Area*	*M. Opts*	*DFFs*	*Area* (μm^2)
Sehwa.15.3	11	10	13	62823	163	21	310456
Sehwa.17.3	6	6	11	37616	159	14	234454
Sehwa.13.4	10	8	14	54584	120	24	285479
Sehwa.15.4	8	7	14	47011	125	26	304104
Sehwa.11.5	9	8	26	73783	79	33	408352
Sehwa.12.5	11	9	22	73680	92	29	369543
Sehwa.19.6	11	7	31	83722	87	39	593705
Sehwa.24.6	22	10	40	175973	111	42	871637
Sehwa.15.9	13	9	45	152487	48	48	666393
Sehwa.16.9	17	10	50	192984	46	57	1029564
Sehwa.14.11	18	10	51	197505	30	53	826538
Sehwa.20.11	19	9	48	191454	44	54	868873
Sehwa.20.17	23	11	47	221892	29	50	877914
Sehwa.27.17	23	11	49	228270	35	51	783790

Table 7.8: Controllers Using Status Registers for Pipelined Examples

Controller Specification		
Design Name	*No. of States*	*Outputs*
Sehwa.15.3	36	170
Sehwa.17.3	9	168
Sehwa.13.4	18	131
Sehwa.15.4	8	137
Sehwa.11.5	9	101
Sehwa.12.5	14	109
Sehwa.19.6	10	115
Sehwa.24.6	52	145
Sehwa.15.9	14	88
Sehwa.16.9	23	90
Sehwa.14.11	19	76
Sehwa.20.11	16	87
Sehwa.20.17	25	70
Sehwa.27.17	28	77

PLA Implementation							
Design Name	*Rows*	*Ipts*	*Opts*	*PLA Area*	*M. Opts*	*DFFs*	*Area* (μm^2)
Sehwa.15.3	30	10	19	123853	157	26	461174
Sehwa.17.3	6	6	7	32234	163	10	155644
Sehwa.13.4	19	9	25	103717	111	32	429688
Sehwa.15.4	8	7	14	47200	125	26	316469
Sehwa.11.5	11	8	24	74366	81	32	429125
Sehwa.12.5	13	8	26	85107	87	32	431443
Sehwa.19.6	9	7	32	80207	86	38	570495
Sehwa.24.6	48	10	45	360937	106	51	994086
Sehwa.15.9	13	8	47	152167	45	50	742560
Sehwa.16.9	23	9	49	220554	46	53	811965
Sehwa.14.11	20	9	50	201343	31	51	821688
Sehwa.20.11	15	8	52	176593	39	54	979000
Sehwa.20.17	24	9	48	216030	27	49	754222
Sehwa.27.17	28	9	49	244170	33	50	767844

Table 7.9: Controllers without Status Registers for Pipelined Examples

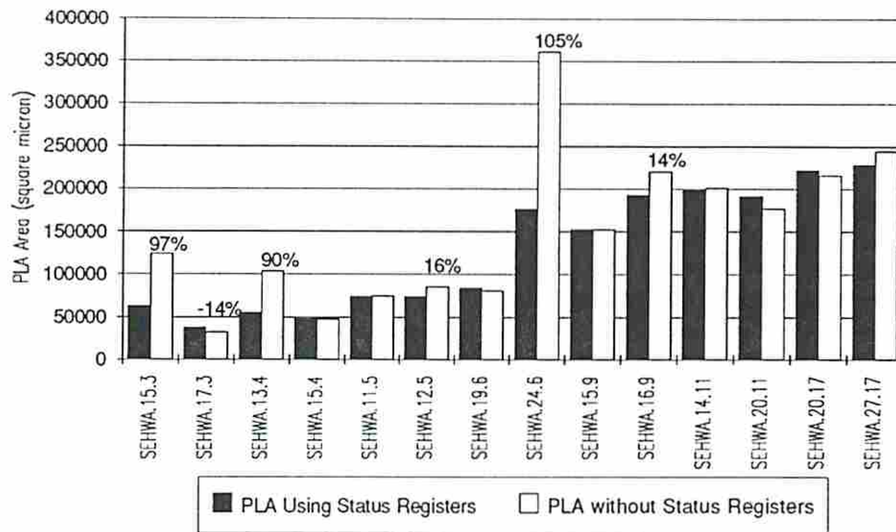


Figure 7.11: PLA Area Comparison of the Pipelined Designs

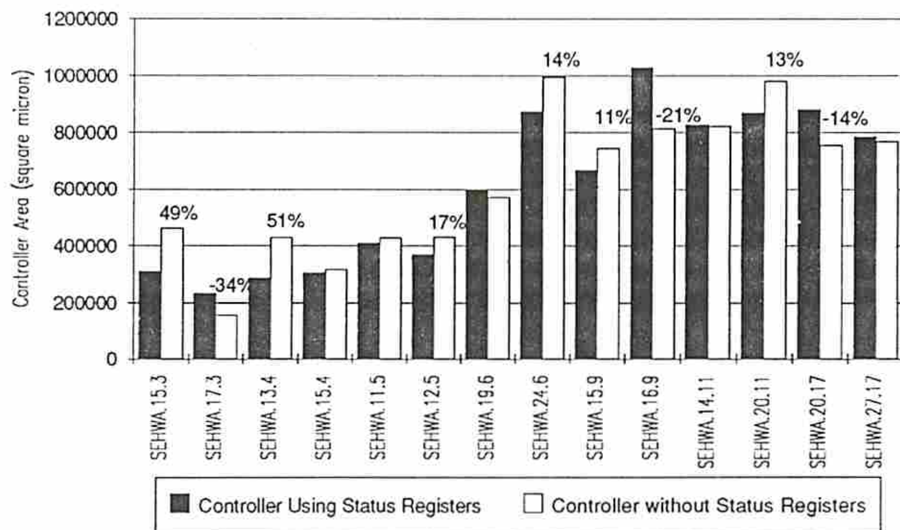


Figure 7.12: Controller Area Comparison of the Pipelined Designs

Chapter 8

Thermal Analysis and Simulation Results

As integrated circuit feature sizes have decreased, designers have been forced to consider physical effects at higher levels of the design process. An increase in power density results from reduced feature sizes, partly due to higher operating speeds, and partly due to the increased component density on integrated circuits. However, the operating temperature of a chip is limited to a certain range for acceptable system reliability. For silicon devices, this temperature is in the range of 75–85$^\circ C$ [Jr.83]. With increasing power density and with limits on the operating temperature, thermal limitations of the chip must be considered during the design process. Consequently, it is imperative to study not only the thermal properties of the device and package material but also the run-time thermal properties of the chip/die so that, during the design stage, a better thermal layout can be obtained to alleviate potentially high thermal stress. While thermal effects on-chip might be relatively inconsequential for many single MOS chips, the current thermal problems associated with multichip modules indicate that thermal effects cannot be ignored during multichip synthesis.

For the design of electronic circuits for reliability, there are a number of handbooks, specifications and guidelines which help us establish a common basis for comparing, evaluating and predicting related or competitive designs. The Military Handbook MIL-HDBK-217E [Mil90] was developed by the Department of Defense to unify reliability prediction methods for integrated circuits produced by the military. According to this handbook, surface temperature is one of the major factors affecting circuit reliability. Computer-aided design software should predict and

analyze thermal properties in circuits while they are under design, highlighting areas which are overused in order to prevent such failures. Engineers can then improve designs so that systems themselves operate at lower temperatures and more reliably.

Localized heat-concentration problems can appear when a high-level synthesis tool or designer uses a greedy approach to the allocation and binding synthesis subtasks which overuse some central resources. Along with the scheduling procedure, the schedule and topology of functional modules and the selection of the package material affect the design enormously. Since not all functional modules are active all the time, utilizations of functional modules, which are the result of scheduling, can also affect the reliability of the design.

Thermal effects thus can represent a limiting factor in the development of ASIC chips. An accurate model of the thermal behavior of the die structure is necessary in order to make reliable designs. Thermal analysis methods can be categorized into two general approaches, namely, analytical solutions and numerical solutions. The analytical approach involves the search for an exact analytical solution for structures with regular geometries. However, using an analytical approach, a solution is not always obtainable for a structure with complex geometries. Therefore, a numerical method is a better approach for a structure with irregular geometries. Numerical techniques such as finite-difference [MAD83], finite-element [WFM83], and boundary element [CPB88] have been widely used to analyze thermal profiles of electronic circuits.

The purpose of this chapter is to provide a novel method for improving overall thermal characteristics of circuits during the high-level synthesis process. The basic idea in this work is to alleviate the heat-concentration phenomenon by averaging work loads[1] between modules of the same functionality during scheduling and by balancing power dissipation during floorplanning. Averaging or spreading the power dissipation of functional modules allows the design to operate with a more balanced thermal profile. As compared to worst case, our simulation results indicate the rise in temperature can be reduced 20% by combining the efforts of improving the floorplan and the schedule.

[1]The amount of heat produced by a functional module is proportional to its work load.

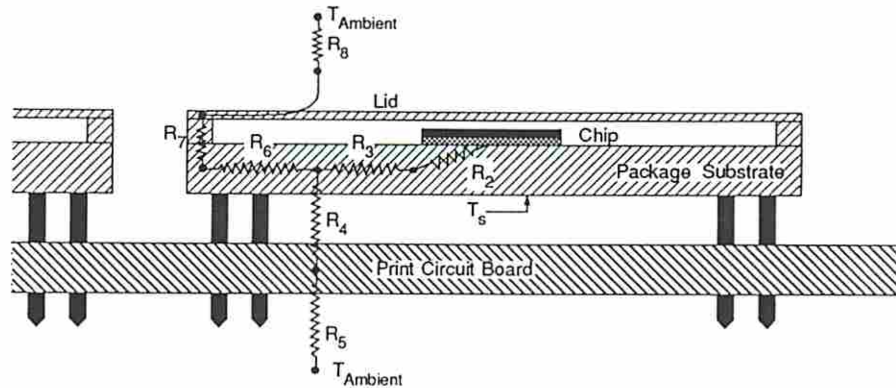


Figure 8.1: A Typical Single-Chip Heat Transfer Model

In the following sections, we first introduce thermal models for a chip under thermal consideration. An analytical solution is then derived. A numerical finite-difference method is also presented. An example is used to illustrate thermal impacts on floorplanning and scheduling results. Finally, some remarks on the thermal analysis are given.

8.1 Thermal Models

To calculate the induced thermal stress of a design, the temperature profile must be known. In our analysis, the heat conduction from the top of die surface to the working fluid[2] is assumed to be negligible, as compared to the heat conducted laterally through the doped substrate to the print circuit board. The thermal conductivities of the substrate and package material are constant at the steady state, and the surface temperature is primarily a function of functional module utilizations and the topology of functional modules.

For a steady-state conductive cooling environment, the single-chip heat transfer environment can be modeled by a network of thermal resistances as shown in Figure 8.1 [TR89]. In the thermal resistance model, the temperature and heat flux are analogous to the voltage and current in the analysis of an electronic circuit. The thermal resistances in Figure 8.1 are derived as

[2]Usually, the working fluid is air.

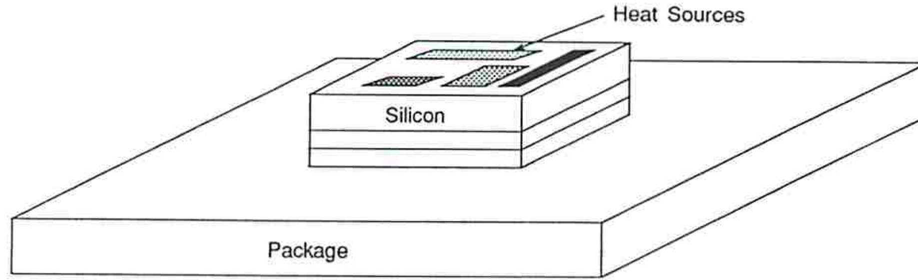


Figure 8.2: A Four-Layer Chip Thermal Model

$$R_t = \frac{W_c}{K \cdot A_c}$$

where W_c the distance between the conduction, K is the thermal conductivity of the conducted material and A_c is the cross-sectional area of the conduction. Since the amount of heat produced inside a chip and the ambient temperature are known, the temperature on the surface of package can then be derived from the network of thermal resistances.

In order to obtain the analytical solution of the temperature profile on the surface of the silicon substrate, we modified this thermal resistance model to a four-layer thermal model as shown in Figure 8.2 [LPM89]. The thermal characteristics are retained in this simplified model. For a common silicon device, the first (top) layer is usually silicon. The second layer is usually a bonding material, such as *epoxy*. The third layer represents metallization on the substrate (usually gold). The fourth (bottom) layer, the package material, uses material such as alumina. Several assumptions have been made in this four-layer thermal model:

- The surface temperature at any position on the silicon depends on the heat dissipated by the functional modules, the thermal conductivities of the silicon and package material and the temperature of the working fluid.
- The heat generated from each functional module on the substrate is assumed to be uniformly distributed. The heat dissipation of a functional module is dependent on the clock frequency and its utilization.

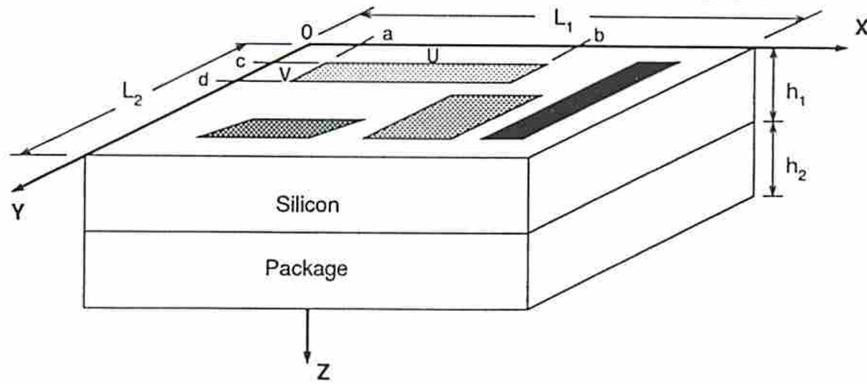


Figure 8.3: A Simplified Layered Chip Thermal Model

- The average power dissipation of a functional module is used in the thermal analysis, since the propagation speed of the heat dissipated is much slower than the clock frequency.

Note that the average power dissipation of a functional module is the product of the maximal power dissipated at the working clock frequency and its utilization. For example, the power dissipated by a typical adder circuit when operating at a frequency of 10 MHz and $V_{DD} = +5$ volts is about 2 mW. If the utilization of this adder on a given design is 80%, the average power dissipation of this adder is 1.6 mW.

8.2 Derivation of the Analytical Solution

Due to the small thickness of the bonding and metallization layers, only the first silicon layer and the fourth package layer are considered. Therefore, the four-layer thermal model shown in Figure 8.2 is further simplified to a two-layer thermal model as shown in Figure 8.3 [CA78]. In this model, the package size is assumed to be the same as the die size; only the thermal properties of the silicon and package layers are considered. To derive the temperature distribution of the die structure in the steady state, the classic heat-flow govern equation must be solved:

$$\nabla^2 T(x, y, z) = 0 \tag{8.1}$$

where the thermal conductivity of the medium is assumed not to be a function of temperature. The need for a three-dimensional solution is due to the fact that heat transfer in the die is three-dimensional. The following boundary conditions are considered:

- The bottom surface (the package surface) has an arbitrary but known temperature distribution $T_0(x, y)$.
- The lateral sides of the layer structure are considered to be adiabatic.[3]
- The heat dissipated on the top surface of the silicon chip is described by the function $P(x, y)$.
- The heat flux is continuous at the interfaces between layers.

Equation 8.1 can be solved by separation of variables or other equivalent techniques. The solution that calculates the temperature on the top silicon surface is thus derived as [CA78]

$$\begin{aligned} T(x,y,0) = & \; T_0(x,y,h_1+h_2) + \left(\frac{h_1}{K_1 L_1 L_2} + \frac{h_2}{K_2 L_1 L_2}\right) \cdot \int_0^{L_1}\int_0^{L_2} P(x,y)\,dx\,dy \\ & + \frac{4}{K_1 L_1 L_2} \cdot \sum_{m=0}^{\infty}\sum_{n=0}^{\infty} \frac{\int_0^{L_1}\int_0^{L_2} P(x,y)\cos(\frac{m\pi x}{L_1})\cos(\frac{n\pi y}{L_2})\,dx\,dy}{A_{mn}\cdot(\delta_m+1)\cdot(\delta_n+1)} \\ & \cdot \frac{\sinh A_{mn}h_1 + \frac{K_1}{K_2}\cdot\cosh A_{mn}h_1 \cdot \tanh A_{mn}h_2}{\frac{K_1}{K_2}\cdot\sinh A_{mn}h_1\cdot\tanh A_{mn}h_2 + \cosh A_{mn}h_1} \cdot \cos(\frac{m\pi x}{L_1})\cdot\cos(\frac{n\pi y}{L_2}) \end{aligned} \quad (8.2)$$

where

$$A_{mn} = \sqrt{\frac{m^2\pi^2}{L_1^2} + \frac{n^2\pi^2}{L_2^2}}$$

δ_m and δ_n are the Kronecker delta and K_1, K_2 are the thermal conductivities of the first layer and the second layer, respectively. Note that the exact solution of the four-layer structure can be derived in a similar manner [LPM89].

[3] Adiabatic means no heat transfer actions between the analyzed structure and the working environment.

In the case of ASIC design, the power distribution $P(x,y)$ can be modeled by a group of uniformly distributed functional modules. This fact suggests that we can represent the function $P(x,y)$ as a set of box functions[4] with the amplitude Q_{f_i} for each functional module. Due to the linearity of the heat distribution function $P(x,y)$, the solution for the structure with a single heat source is first derived. For the structure with multiple sources, the solutions are then obtained by the superposition of corresponding single-source solutions. Two integrals in Equation 8.2 are thus derived as

$$\int_0^{L_1}\int_0^{L_2} P(x,y)\,dx\,dy = \sum_{i\in M} Q_{f_i} \tag{8.3}$$

$$\int_0^{L_1}\int_0^{L_2} P(x,y)\cos(\frac{m\pi x}{L_1})\cos(\frac{n\pi y}{L_2})\,dx\,dy = \\ \sum_{i\in M} Q_{f_i}\cdot\{sinc(\frac{m\pi U}{2L_1})\cdot sinc(\frac{n\pi V}{2L_2})\cdot\cos(\frac{m\pi(a+b)}{2L_1})\cdot\cos(\frac{n\pi(c+d)}{2L_2})\} \tag{8.4}$$

where

$$sinc(g) = \frac{\sin(g)}{g}$$

Q_{f_i} is the heat dissipated by module i and M is the allocated module set.

Since the solution has the form of an infinite series, a criterion is needed to truncate the summation at a given required precision. Through a closer inspection of the infinite double Fourier cosine series, a rule of thumb, $m = 6L_1/U$, $n = 6L_2/V$, is used which generally allows the temperature to converge within 1 percent of the final value [LPM89].

[4] A box function, $B(x,y)$, is a function with zero-value everywhere, except a constant value inside a rectangle area.

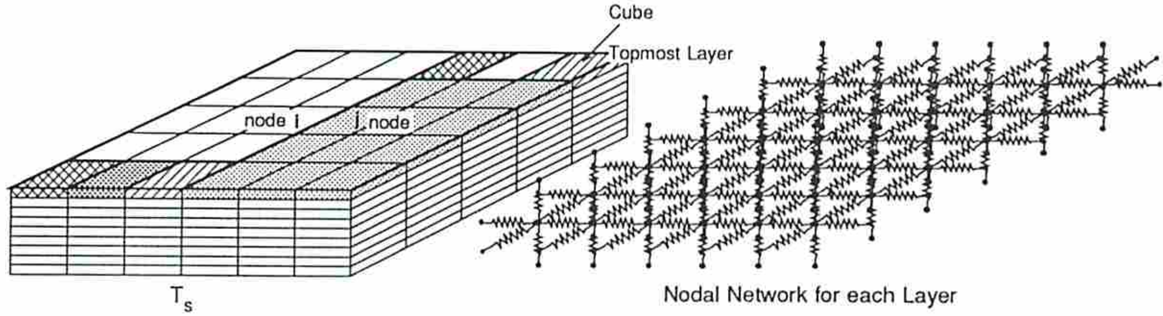


Figure 8.4: Finite Difference Approximation and The Corresponding Nodal Network

8.3 Derivation of the Numerical Solution

The basic principle of the finite difference approach is derived from differential equations via Taylor's expansion. Instead of using finite difference equations directly, these equations can be interpreted in a physical way to permit a more convenient application. Consider the die structure shown in Figure 8.4. The die is divided into a number of cubes. The cross-sectional area of each cube equals to the area of a unit cell,[5] so a larger functional module (such as a multiplier) is divided into several cells. In this case, the heat dissipated in the module is assumed to equally distribute among all topmost cubes which cover the module. The heat dissipation of a cube is assumed to be a point heat source on the geometric center of the cube. A nodal network is thus obtained, representing the structure under a steady-state condition. Each node i, which is the geometric center of cube i, must satisfy the equations

$$\sum_{j \in B_i} \frac{t_j - t_i}{R_{ij}} + q_i = 0 \tag{8.5}$$

where B_i is the set of all neighboring nodes adjacent to node i and R_{ij} is the thermal resistance between node i and node j which equals to $\frac{\delta_{ij}}{k_s A_{ij}}$. δ_{ij} denotes the conduction distance between node i and node j and A_{ij} is the cross-sectional area for heat conduction normal to δ_{ij}. q_i is the heat produced in the volume lump

[5] We use a constructive approach on floorplanning in the preliminary 3D scheduling research. We define a unit cell as a primitive block.

1.2 μm Technology	8 bit adder		8 bit multiplier	
	Delay	Area	Delay	Area
	13 ns	75 mil^2	32 ns	903 mil^2

Table 8.1: Library Set Created by ChipCrafter Silicon Compiler

at i (i.e. cube i) which is the average heat dissipated in the module divided by the number of topmost cubes covering of the module. Note that the heat produced by a module is assumed only coming from the topmost layer of cubes, since the power is mainly consumed by semiconductor circuits implanted on the surface of the silicon substrate, during signal switching.

The formulation so far is restricted to interior points of the structure in which the heat conduction is taking place. In order to solve the equation set represented by Equation 8.5, imposed boundary conditions must also be satisfied. The boundary condition here is the equilibrium temperature on the bottom of the structure surface which is the surface of the package [TR89]. The temperature is assumed to be uniformly distributed on the surface of the package due to the high heat conductivity of the package material. By solving the n-equation set simultaneously, the temperature profile can be obtained. The profile of the thermal stress can then be calculated.

8.4 Thermal Experiments and Results

The thermal model presented in this chapter was used to experiment with the ADAM high-level synthesis tools developed at USC. Figure 8.5 shows an example of a data flow graph which is used as an input for high-level synthesis. Table 8.1 lists the module library set, which was derived from the Cascade Design Automation ChipCrafter silicon compiler. The data path synthesis program used to experiment with the thermal model is based on the preliminary 3D scheduling research, which incorporates interconnection delays during scheduling using floorplanning. A two-time-step schedule for a non-pipelined FIR filter design is shown with the cross-hatched line in Figure 8.5.

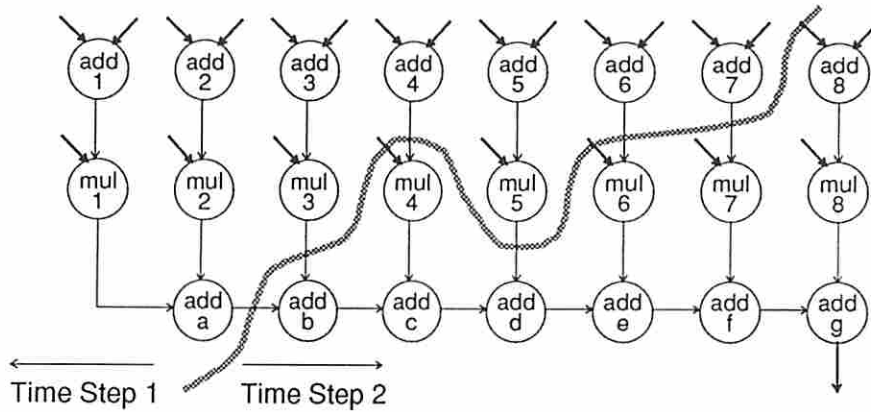


Figure 8.5: A 2-time-step Non-pipelined FIR Filter Design

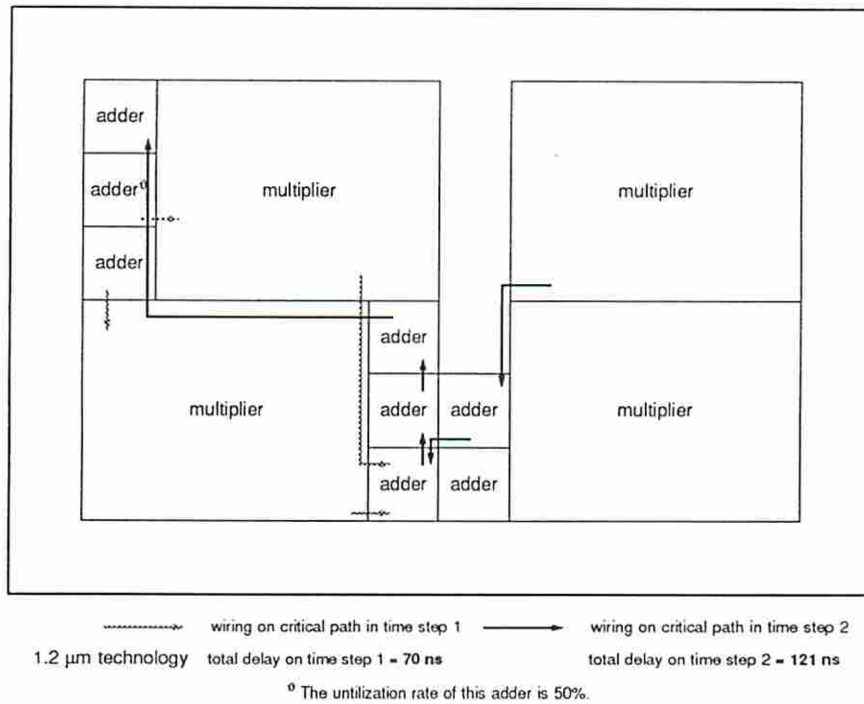


Figure 8.6: The Floorplan for the FIR Filter Design with Minimum Operators

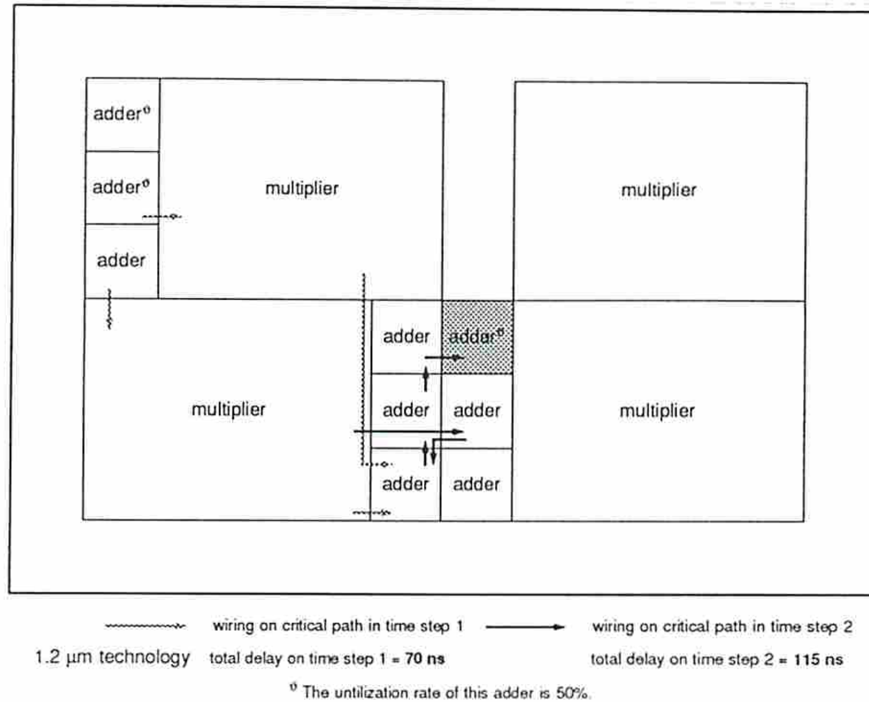


Figure 8.7: The Floorplan for the FIR Filter Design with a Redundant Adder

The 3D scheduling algorithm minimizes interconnection delays along critical paths to improve the design performance. The software tries to introduce *redundant operators*[6] to alleviate interconnection delays, when the synthesized design cannot achieve user specified timing constraints (see Section 4 for more details). For example, the floorplan of the two-time-step non-pipelined FIR filter design in Figure 8.5 is shown in Figure 8.6. In this example, the feasible design with minimum operators allocated contains 8 adders and 4 multipliers. The software found interconnection delays along critical paths are reduced by introducing one more adder, which is shown shaded in Figure 8.7, in this two-time-step non-pipelined FIR filter design.

The thermograms of the design with minimum operators and the design with a redundant adder are shown in Figures 8.8a and 8.8b, respectively. The temperature shown in thermograms is the temperature difference between the top surface of the silicon substrate and the bottom surface of the package. The thermal profiles were

[6]Redundant operators are operators not required for the feasible design with minimum operators.

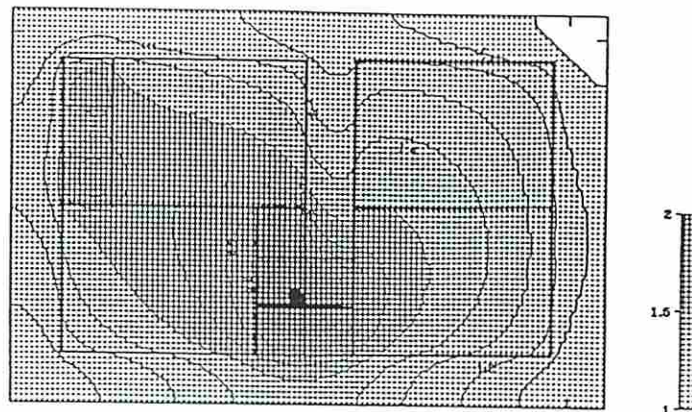
calculated by both analytical and numerical methods. The results of these two solutions match, but the analytical solution provides a more flexible and efficient way to compute simulation results. The complexity of the analytical solution is dependent on the number of heat sources and the number of terms in the infinite Fourier series to be summed. For example, in the design with minimum operators, it took 2.2 seconds of CPU time on a Solbourne 5e/900 machine (or 2.25 seconds of CPU time on a SUN 4/460 machine) in the average to compute one sample temperature when there are twelve heat sources on the top surface of the silicon substrate.

In order to demonstrate the problem of heat concentration, adders in our experiments are considered to be "hot-problem devices" which produce three times the amount of heat that multipliers produce when both adders and multipliers are fully utilized. The thermogram of the design with minimum operators in Figure 8.8a presents a smaller "hot-spot" as compared to the thermogram of the design with a redundant adder in Figure 8.8b. The reason was obvious after these two floorplans were compared and analyzed. The design with a redundant adder design does produce better system performance as compared to the design with minimum operators. However, the design with a redundant adder attempts to use modules around the central area intensively to shorten wiring delays, which results a large amount of heat being produced in the central area, causing heat dissipation problems.

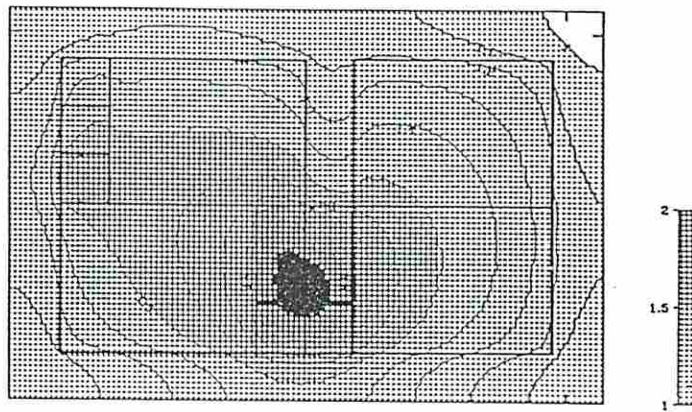
To resolve the heat-concentration problem, two solutions are proposed. First, by spreading adders around the problem area over the unused space on the floorplan, the heat-concentration problem can be alleviated. We applied this strategy to the FIR filter example. The resultant floorplan and thermogram are shown in Figures 8.9 and 8.8c, respectively. The results are successful. The temperature dropped about 20%.[7] This heat-balancing strategy resolved the heat-concentration problems at the cost of performance degradation.

In a design whose timing constraint is crucial, the synthesis program or designer does not allow any degradation of the system performance. In this case, we propose

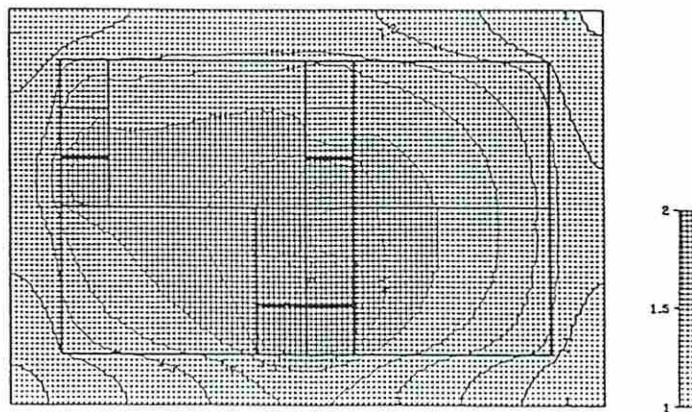
[7]The dropped percentage is the ratio of reduced temperature versus overall temperature difference.



a) Minimum Operator Design



b) One Redundant Adder Design



c) Thermal-Improved Design

Figure 8.8: The Thermograms of the FIR Filter Design

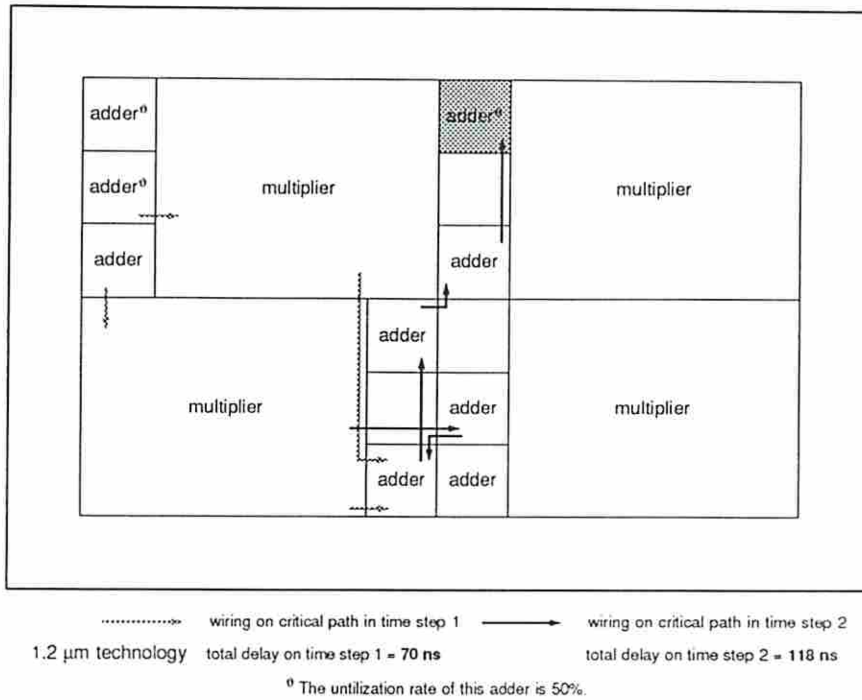


Figure 8.9: The Floorplan for the Heat-Balanced FIR with a Redundant Adder

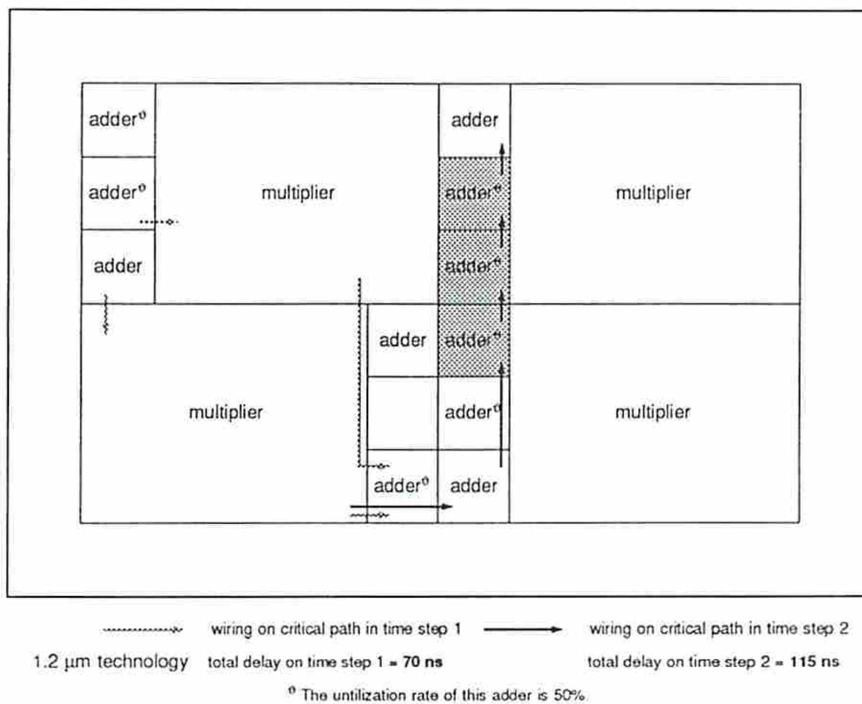


Figure 8.10: The Floorplan for the Heat-Balanced FIR with Redundant Adders

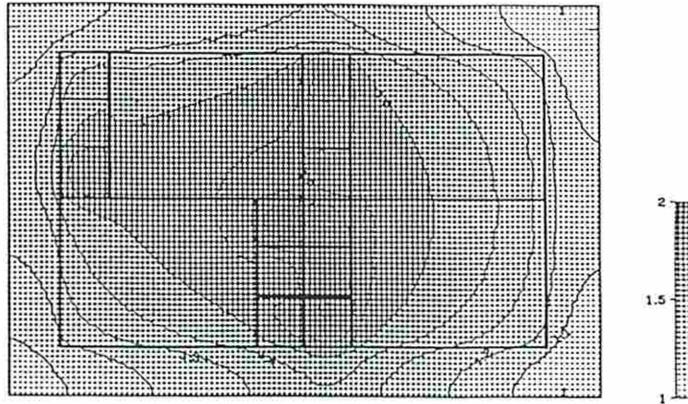


Figure 8.11: The Thermogram of the Heat-Balanced FIR with Redundant Adders

another strategy by allocating additional redundant adders to reduced utilizations of adders around the central area. The resulting floorplan is shown in Figure 8.10. Three redundant adders were allocated to reduce the heat-concentration problem in this example. Utilizations of some adders are reduced in this floorplan. The related thermogram shown in Figure 8.11 reveals that both the thermal constraint and performance constraint are achieved simultaneously. The thermal improvement achieved is the same as in the previous case, the temperature is reduced about 20% (Indeed, the thermal profile of this floorplan is slightly better than the previous one.). These two design cases whose thermal properties have been improved illustrate that design tradeoffs are possible between area, performance and reliability factors. We believe tradeoffs can be done much better if thermal effects are simultaneously considered during the data path synthesis process instead of considering them as two problems separately and/or sequentially.

Chapter 9

Conclusions and Future Research

This thesis has attempted to solve many design issues associated with high-level synthesis. Most design automation systems consist of a collection of tools which are individually focused on specific optimizations and do not have a global understanding of the problem due to the high complexity of synthesis. As a result, excessive post-design iterations may be required to produce good quality designs.

The work presented here describes an integrated high-level synthesis approach in which number of design issues such as effects of interconnections and thermal problems are concurrently being considered during data path synthesis. Both pipelined and non-pipelined designs with or without conditional branches are considered in our high-level synthesis system. The nested inner-loop problem for a non-pipelined design is also addressed. Different design strategies are proposed to tradeoff area against performance while the functionality of a circuit design is still preserved.

The research presented in this thesis can be classified into three major areas: data path synthesis, control path synthesis, and thermal analysis and synthesis. Below we describe the contributions made in each area by this thesis. We also list some open problems for future research, which in some cases are related problems that could not be fully addressed in this work, but in other cases arise as a direct consequence of the contributions in this work itself.

9.1 Data Path Synthesis

In this thesis, a new approach to data path synthesis problem is addressed. We consider scheduling, module allocation, module assignment and floorplanning simultaneously during data path synthesis. The main objective of our approach is to produce better schedules and maximize the sharing among allocated modules for register-transfer level designs. A floorplan is constructed incrementally during data path synthesis which feeds back effects of interconnections to the scheduling process in order to guide the scheduler to determine the next scheduling iteration.

We have shown the effects of interconnections in our experiments. Totally ignoring the existence of interconnections during the scheduling process may produce poor schedules in register-transfer level designs that lead to unnecessary performance degradation. Taking effects of interconnections into account during scheduling is thus necessary in order to produce better schedules (i.e. better designs), specially in designing submicron integrated circuits. In our experiments, the placement and pin assignment of a module greatly affect the area and performance of a circuit design. Also, the performance of a circuit design is dramatically influenced by its schedule.

In the future work, a more accurate delay estimation model, which considers various sizes of output buffers, is needed in order to provide more accurate timings to the scheduler. A more precise estimate of the wiring area and delay, information on pin locations of a module and a global routing procedure may be incorporated during floorplanning to increase the accuracy in estimating effects of interconnections in the 3D scheduling algorithm.

9.2 Control Path Synthesis

In the research on control path synthesis, two possible controller implementation strategies are presented: controllers with status registers and without status registers. Intuitively, controllers without status registers would seem better, because less state encoding bits are used. However, experimental results contradicted our initial expectation: controllers implemented using status registers are smaller in

all non-pipelined design cases tested and many pipelined design cases. After we traced one of the non-pipelined design cases, we believe that the improvement is possibly due to the use of a multiple-code state encoding (instead of a conventional unicode state encoding) in controllers using status registers. Therefore, in some design cases, controllers implemented using status registers are smaller than those without status registers after logic simplification.

The controller design which uses status registers provides an effective heuristic for producing multiple-code state encodings using an unicode state encoding algorithm. In general, a multiple-code state encoding, if done intelligently, is expected to produce better results than an unicode state encoding. However, to the best of our knowledge, there is no current reported work on multiple-code state encoding algorithms. One possible reason is the high complexity of computing optimal multiple-code state encodings.

There is still much work to be done here. More research on controller design with and without status registers may lead to more concrete results that could help us choose a better controller implementation style in an early design stage of control path synthesis. This research will also assist us to develop more powerful multiple-code state encoding algorithms by giving hints to unicode state encoding algorithms. We believe that with further research on multiple-code state encodings, we may be able to derive more powerful and efficient heuristics for the state encoding problem.

9.3 Thermal Analysis and Synthesis

The main objective of this approach is to floorplan heat-balanced designs by avoiding overuse of the functional modules around the central area. We have shown a localized heat concentration which induces a high thermal stress in the problem area and causes reliability problems. To calculate the thermal profile on the surface of an integrated circuit, a four-layer thermal model and a simplified two-layer thermal model were proposed. Both analytical and numerical solutions were investigated for the two-layer thermal model in our experiments and they produced

similar results. However, the analytical solution gives us more flexibility and efficiency during the computation of thermal profiles. We also found that thermal problems may be successfully alleviated by rearranging functional modules around the problem area or by introducing extra redundant operators.

In the future work, a more exact thermal model should be studied which allows more layers to be considered in the computation of thermal profiles. Computation time may be improved by using an infinite plate model [LMP89], when the dimension ratio of heat sources to die structure is large.

Reference List

[AHU74] A. V. Aho, J. E. Hopcroft, and J. D. Ullman, editors. *The Design and Analysis of Computer Algorithms*, chapter 2, pages 52–55. Addison Wesley Publishing Company, 1974.

[BG90] F. Brewer and D. Gajski. Chippe: A System for Constraint Driven Behavioral Synthesis. *IEEE Transactions on Computer-Aided Design*, 9(7):681–695, July 1990.

[CA78] R. Castello and P. Antognetti. Integrated-Circuit Thermal Modeling. *IEEE Journal of Solid-State Circuits*, SC-13(3):363–366, June 1978.

[Che91] C. Chen. VHDL2DDS: A VHDL Language to DDS Data Structure Translator. Technical Report 91-21, Department of Electrical Engineering, University of Southern California, July 1991.

[Con89] J. Cong. Pin Assignment with Global Routing. In *Proceeding of 1989 IEEE Int. Conf. on Computer-Aided Design*, pages 302–305, November 1989.

[CPB88] C. C. Chen, A. L. Palisoc, and J. M. Baynham. Thermal Analysis of Solid-State Devices Using Boundary Element Method. *IEEE Transactions on Electronic Devices*, 35(7):1151–1153, July 1988.

[CPTR89] C. Chu, M. Potkonjak, M. Thaler, and J. Rabaey. Hyper: An Interactive Synthesis Environment for High Performance Real Time Applications. In *Proceeding of 1989 IEEE Int. Conf. on Computer Design*, pages 432–435, October 1989.

[CT90] R. J. Cloutier and D. E. Thomas. The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm. In *Proceeding of 27th ACM/IEEE Design Automation Conference*, pages 71–76, July 1990.

[DK87] W. Dai and E. S. Kuh. Simultaneous Floor Planning and Global Routing for Hierarchical Building-Block Layout. *IEEE Transactions on Computer-Aided Design*, CAD-6(5):828–837, September 1987.

[Gaj88a] D. Gajski, editor. *Silicon Compilation*, chapter 8, pages 311–360. Addison-Wesley Publishing Company, Inc., 1988.

[Gaj88b] D. Gajski, editor. *Silicon Compilation*, chapter 7, pages 204–310. Addison-Wesley Publishing Company, Inc., 1988.

[Gir84] E. Girczyc. *Automatic Generation of Microsequenced Data Paths to Realize ADA Circuit Descriptions.* PhD thesis, Carleton University, July 1984.

[Gup91] P. Gupta. *PLA and Wire Delay Analysis.* Department of Electrical Engineering, University of Southern California, 1991. Internal Report.

[GVM87] G. Goossens, J. Rabaey J. Vandewalle, and H. De Man. An Efficient Microcode Compiler for Custom DSP Processors. In *Proceeding of 1987 IEEE Int. Conf. on Computer-Aided Design*, pages 24–27, November 1987.

[Hag91] J. W. Hagerman. A Fast and Accurate Technique for Function Unit Allocation Estimation. Technical Report CMUCAD-91-28, Carnegie Mellon University, April 1991.

[Hay88] J. Hayes. *Computer Architecture and Organization.* McGraw-Hill Book Company, second edition, 1988.

[HCLH90] C. Huang, Y. Chen, Y. Lin, and Y. Hsu. Data Path Allocation Based on Bipartite Weighted Matching. In *Proceeding of 27th ACM/IEEE Design Automation Conference*, pages 499–504, June 1990.

[HGF89] A. Herrigal, M. Glaser, and W. Fichtner. A Global Floorplanning Technique for VLSI Layout. In *Proceeding of 1989 IEEE Int. Conf. on Computer-Aided Design*, pages 92–95, November 1989.

[Jai89] R. Jain. *High-Level Area-Delay Prediction with Application to Behavioral Synthesis.* PhD thesis, University of Southern California, July 1989.

[JMP88] R. Jain, M. J. Mlinar, and A. C. Parker. Area-Time Model for Synthesis of Non-Pipelined Designs. In *Proceeding of 1988 IEEE Int. Conf. on Computer-Aided Design*, pages 48–51, November 1988.

[JPP87] R. Jain, A. C. Parker, and N. Park. Predicting Area-Time Tradeoffs for Pipelined Designs. In *Proceeding of 24th ACM/IEEE Design Automation Conference*, pages 35–41, June 1987.

[Jr.83] A. J. Boldgett Jr. Microelectronic Packaging. *Scientific American*, pages 86–96, July 1983.

[KK91] J. Kim and F. Kurdahi. Synthesis of Time-Stationary Controllers for Pipelined Data Paths. In *Proceeding of 1991 IEEE Int. Conf. on Computer-Aided Design*, pages 30–33, November 1991.

[Kna86] D. Knapp. *A Planning Model of the Design Process.* PhD thesis, University of Southern California, December 1986.

[Kna90] D. Knapp. Feedback-Driven Datapath Optimization in Fasolt. In *Proceeding of 1990 IEEE Int. Conf. on Computer-Aided Design*, pages 300–303, November 1990.

[KP87] F. Kurdahi and A. C. Parker. REAL : A Program for REgister ALlocation. In *Proceeding of 24th ACM/IEEE Design Automation Conference*, pages 210–215, June 1987.

[KP89] F. J. Kurdahi and A. C. Parker. Techniques for Area Estimation of VLSI Layouts. *IEEE Transactions on Computer-Aided Design*, 8(1):81–92, January 1989.

[KP90] K. Kucukcakar and A. Parker. Data Path Tradeoffs using MABAL. In *Proceeding of 27th ACM/IEEE Design Automation Conference*, pages 511–516, June 1990.

[KR91] F. J. Kurdahi and C. Ramachandran. LAST: Layout Area and Shape Function EsTimator. In *Proceeding of 1st ACM/IEEE European Design Automation Conference*, pages 351–355, February 1991.

[Lau79] U. Lauther. A Min-Cut Placement Algorithm for General Cell Assemblies Based on a Graphical Representation. In *Proceeding of 16th ACM/IEEE Design Automation Conference*, pages 1–10, 1979.

[Len90] T. Lengauer. *Combinatorial Algorithms for Integrated Circuit Layout.* Wiley-Teubner Series in Computer Science, 1990.

[LMP89] C. C. Lee, Y. J. Min, and A. L. Palisoc. A General Integration Algorithm for the Inverse Fourier Transform of Four-Layer Infinite Plate Structure. *IEEE Transactions on Components, Hybrids, Manufacturing Technology*, 12(4):710–716, December 1989.

[LPM89] C. C. Lee, A. L. Palisoc, and Y. J. Min. Thermal Analysis of Integrated Circuit Devices and Packages. *IEEE Transactions on Components, Hybrids, Manufacturing Technology*, 12(4):701–709, December 1989.

[MAD83] L. M. Mahalingam, J. A. Andrews, and J. E. Drye. Thermal Analysis on Pin Grid Array Packages for High Density LSI and VLSI Logic Circuits. *IEEE Transactions on Components, Hybrids, Manufacturing Technology*, CHMT-6:246–256, September 1983.

[McF86] M. McFarland. Using Bottom-up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions. In *Proceeding of 23th ACM/IEEE Design Automation Conference*, pages 474–480, July 1986.

[Mil90] Military Handbook MIL-HDBK-217E. *Reliability Prediction of Electronic Equipment.* Rome Air Development Center, February 1990.

[Mli91] M. Mlinar. *Control Path/Data Path Tradeoffs in VLSI Design.* PhD thesis, University of Southern California, May 1991.

[MPC88] M. McFarland, A. Parker, and R. Camposano. Tutorial on High-Level Synthesis. In *Proceeding of 25th ACM/IEEE Design Automation Conference*, pages 330–336, July 1988.

[NCP82] A. Nagle, R. Cloutier, and A. Parker. Synthesis of Hardware for the Control of Digital Systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-1(4):201–212, October 1982.

[OP90] M. Osterman and M. Pecht. Placement for Reliability and Routability of Convectively Cooled PWB's. *IEEE Transactions on Computer-Aided Design*, 9(7):734–744, July 1990.

[Ott83] R. H. J. M. Otten. Efficient Floorplan Optimization. In *Proceeding of 1983 IEEE Int. Conf. on Computer Design*, pages 499–502, October 1983.

[Pan88] B. M. Pangrle. Splicer: A Heuristic Approach to Connectivity Binding. In *Proceeding of 25th ACM/IEEE Design Automation Conference*, pages 536–541, June 1988.

[PD86] D. P. La Potin and S. W. Director. Mason: A Global Floorplanning Approach for VLSI Design. *IEEE Transactions on Computer-Aided Design*, CAD-5(4):477–489, October 1986.

[Ped91] M. Pedram. *An Integrated Approach to Logic Synthesis and Physical Design.* PhD thesis, University of California at Berkeley, August 1991.

[PK89] P. Paulin and J. Knight. Force-Directed Scheduling for the Behavioral Synthesis of ASIC's. *IEEE Transactions on Computer-Aided Design*, June 1989.

[PMSK90] M. Pedram, M. Marek-Sadowska, and E. Kuh. Floorplanning with Pin Assignment. In *Proceeding of 1990 IEEE Int. Conf. on Computer-Aided Design*, pages 98–101, November 1990.

[PP88] N. Park and A. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Transactions on Computer-Aided Design*, 7(3):356–370, March 1988.

[PPM86] A. Parker, J. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proceeding of 23th ACM/IEEE Design Automation Conference*, pages 461–466, July 1986.

[Sak83] T. Sakurai. Approximation of Wiring Delay in MOSFET LSI. *IEEE Journal of Solid-State Circuits*, SC-18(4):418–426, August 1983.

[Sto83] L. Stockmeyer. Optimal Orientations of Cells in Slicing Floorplan Designs. *Information and Control*, 57:91–101, 1983.

[TPL+86] C. Tseng, A. M. Prabhu, C. Li, Z. Mehmood, and M. M. Tong. A Versatile Finite State Machine Synthesizer. In *Proceeding of 1986 IEEE Int. Conf. on Computer-Aided Design*, pages 206–209, November 1986.

[TR89] R. Tummala and E. Rymaszewski. *Microelectronics Packaging Handbook*, chapter 4, pages 167–224. Van Nostrand Reinhold, 1989.

[TWR+88] C. Tseng, R. Wei, S. G. Rothweiler, M. M. Tong, and A. K. Bose. Bridge: A Versatile Behavioral Synthesis System. In *Proceeding of 25th ACM/IEEE Design Automation Conference*, pages 415–420, June 1988.

[UT86] S. Unger and C. Tan. Clocking Schemes for High-Speed Digital Systems. *IEEE Transactions on Computers*, C-35(10):880–895, October 1986.

[VHD88] The Institute of Electrical and Electronics Engineers, Inc. *IEEE Standard VHDL Language Reference Manual*, March 1988. IEEE Std 1076-1987.

[WFM83] D. L. Waller, L. R. Fox, and R. J. Mannemann. Analysis of Surface Mount Thermal and Thermal Stress Performance. *IEEE Transactions on Components, Hybrids, Manufacturing Technology*, CHMT-6:257–266, September 1983.

[WL86] D. F. Wong and C. L. Liu. A New Algorithm for Floorplan Design. In *Proceeding of 23th ACM/IEEE Design Automation Conference*, pages 101–107, June 1986.

[WTL91] W. Wolf, A. Takach, and T. Lee. *High-Level VLSI Synthesis*, chapter 10, pages 231–254. Kluwer Academic Publishers, 1991.

[WW90] T. Wang and D. F. Wong. An Optimal Algorithm for Floorplan Area Optimization. In *Proceeding of 27th ACM/IEEE Design Automation Conference*, pages 180–186, June 1990.

[Zim88] G. Zimmerman. A New Area and Shape Function Estimation Technique for VLSI Layouts. In *Proceeding of 25th ACM/IEEE Design Automation Conference*, pages 60–65, June 1988.

[ZSRM90] J. Zegers, P. Six, J. Rabaey, and H. De Man. CGE: Automatic Generation of Controllers in the CATHEDRAL-II Silicon Compiler. In *Proceeding of 1990 IEEE Int. Conf. on Computer-Aided Design*, pages 617–621, November 1990.

Appendix A

Layouts of the 10-Time-Step Non-Pipelined FIR Filter Design

This appendix shows the partial bindings produced by LADS and the layouts of the 10-time-step non-pipelined FIR filter design example presented in Chapter 6. The layouts were produced using ChipCrafter. Solid lines in the layouts show the critical paths obtained by the timing simulator in ChipCrafter. The figures are scaled in order to reflect the relative dimensions among the layouts. Five layouts were created for each schedule. ChipCrafter I, ChipCrafter II and ChipCrafter III use the placements produced by ChipCrafter with different cooling schedules.

Operation	*Operator*	*Register*	*Period*
add1	ADD01	REG01	3 - 3
add2	ADD01	REG01	2 - 2
add3	ADD01	REG01	4 - 4
add4	ADD01	REG01	5 - 5
add5	ADD01	REG01	6 - 6
add6	ADD01	REG01	7 - 7
add7	ADD01	REG01	8 - 8
add8	ADD01	REG01	9 - 9
mul1	MUL01	REG02	4 - 4
mul2	MUL01	REG02	3 - 3
mul3	MUL01	REG02	5 - 5
mul4	MUL01	REG02	5 - 6
mul5	MUL01	REG02	7 - 7
mul6	MUL01	REG02	8 - 8
mul7	MUL01	REG02	9 - 9
mul8	MUL01	REG01	10 - 10
adda	ADD02	REG03	5 - 5
addb	ADD02	REG03	6 - 6
addc	ADD02	REG03	7 - 7
addd	ADD02	REG03	8 - 8
adde	ADD02	REG03	9 - 9
addf	ADD02	REG02	10 - 10
addg	ADD02	None	N.A.

Table A.1: The Bindings for the 10-Time-Step Non-Pipelined FIR Filter Design Produced by LADS

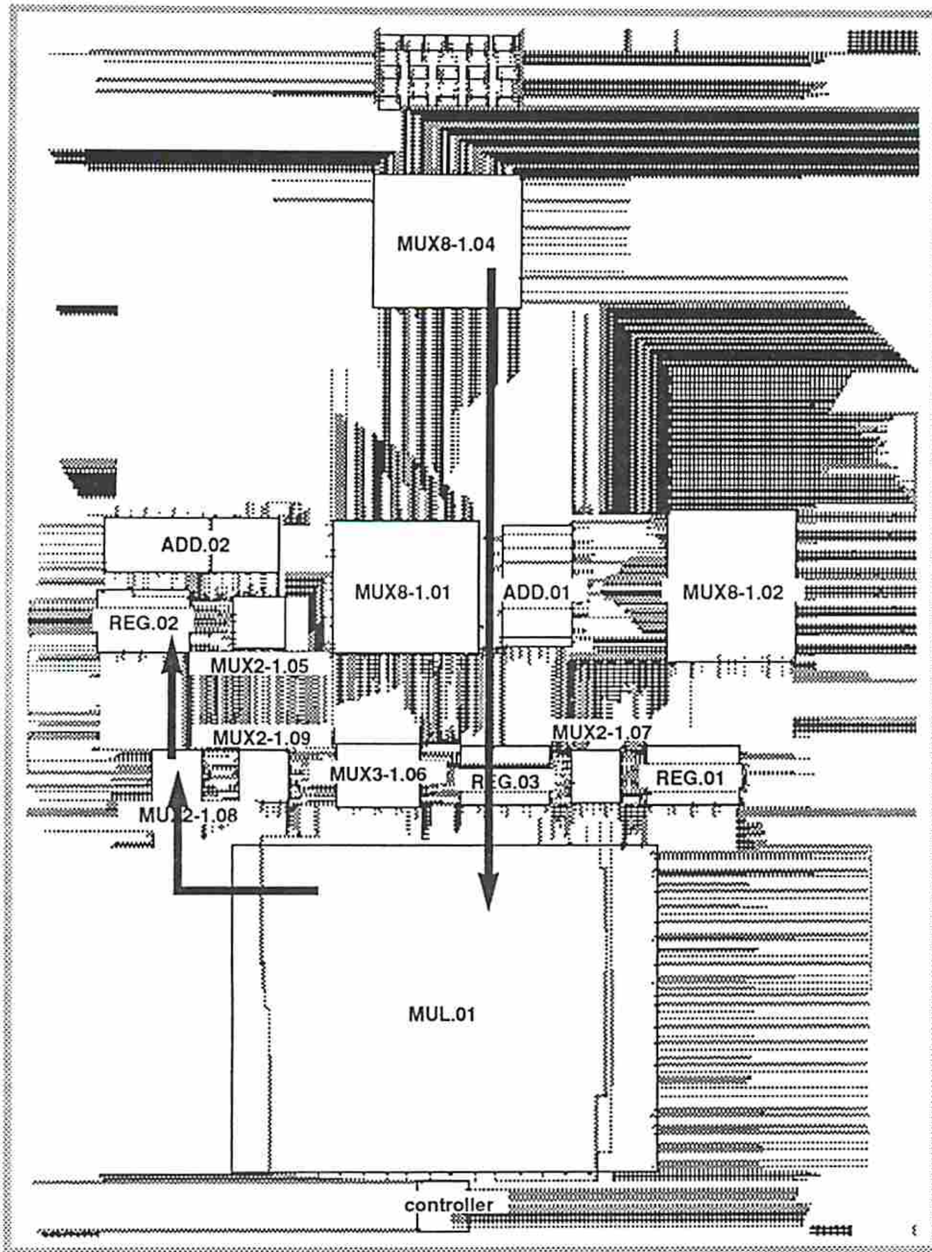


Figure A.1: The Layout of *ChipCrafter Placement I* using LADS Schedule

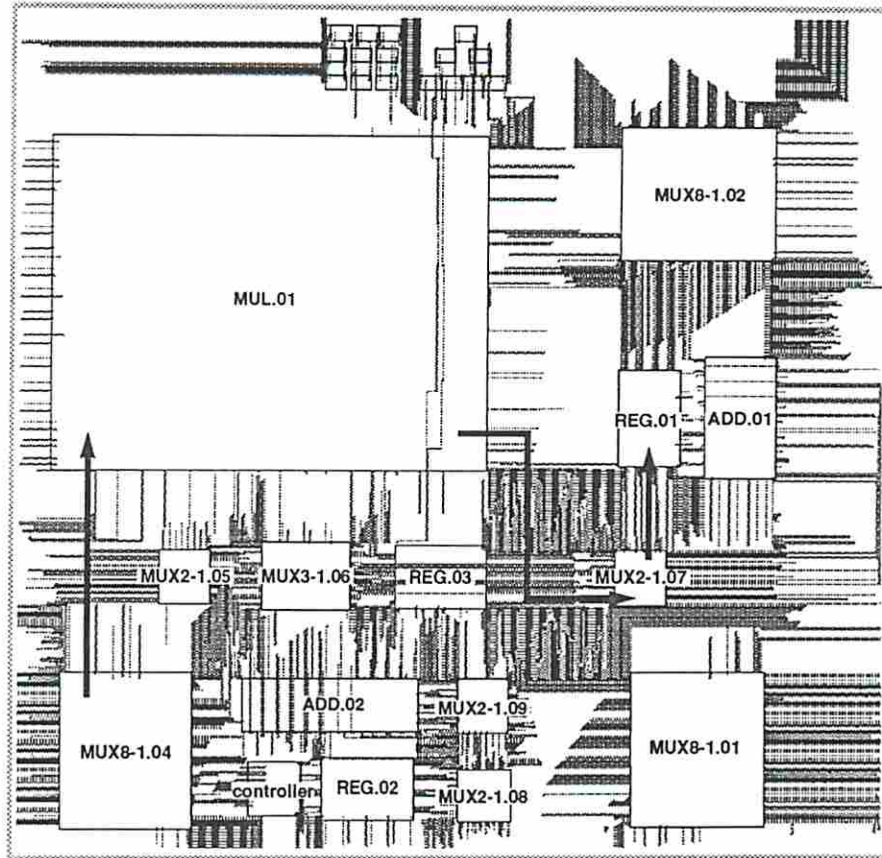


Figure A.2: The Layout of *ChipCrafter Placement II* using LADS Schedule

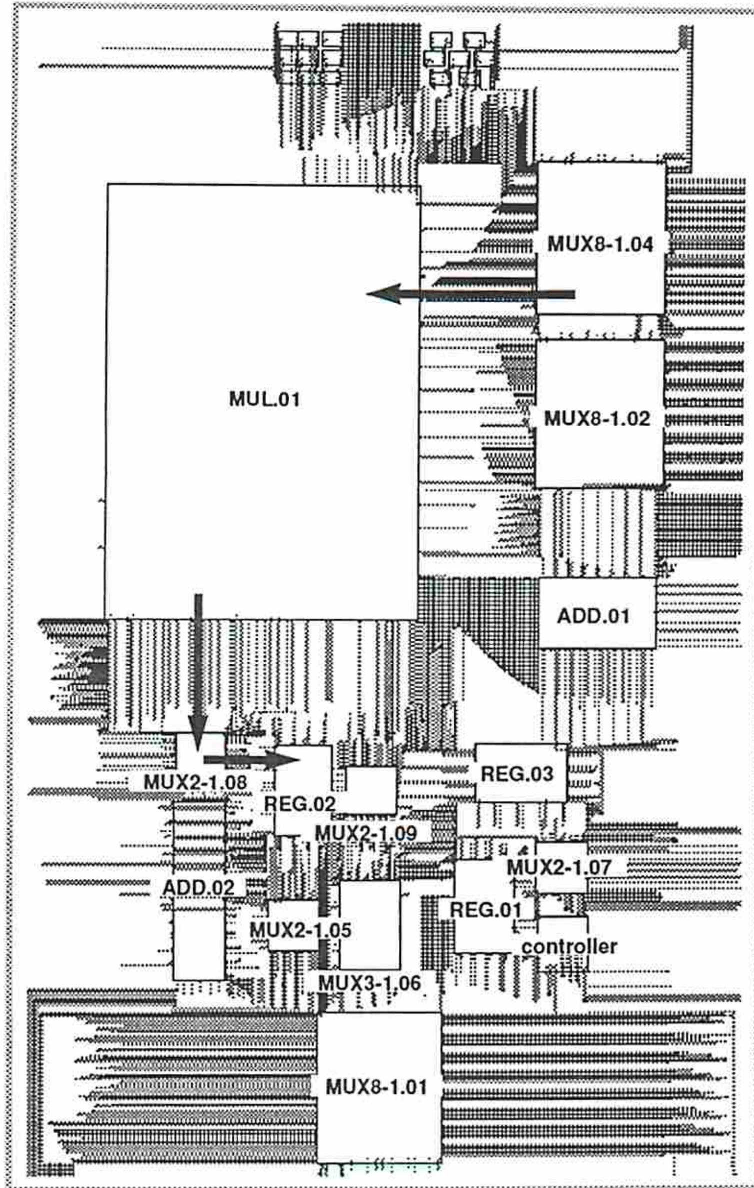


Figure A.3: The Layout of *ChipCrafter Placement III* using LADS Schedule

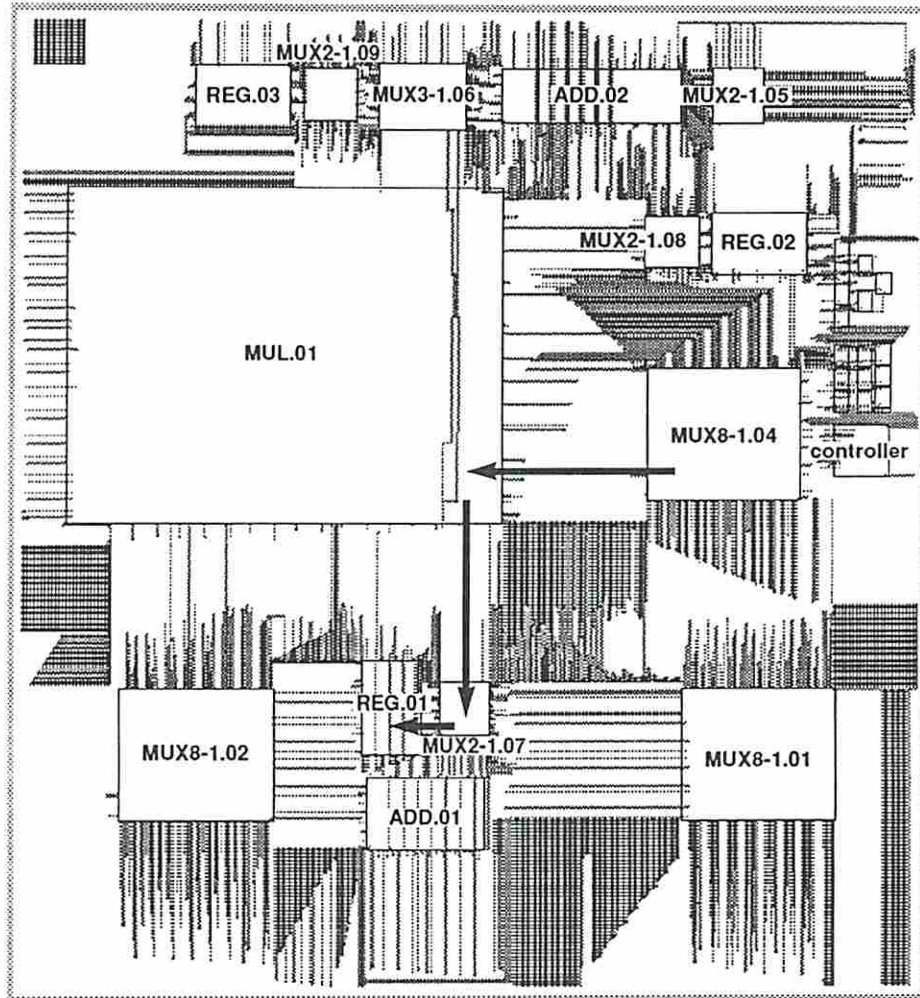


Figure A.4: The Layout using LADS Schedule and Floorplan without Pin Assignment

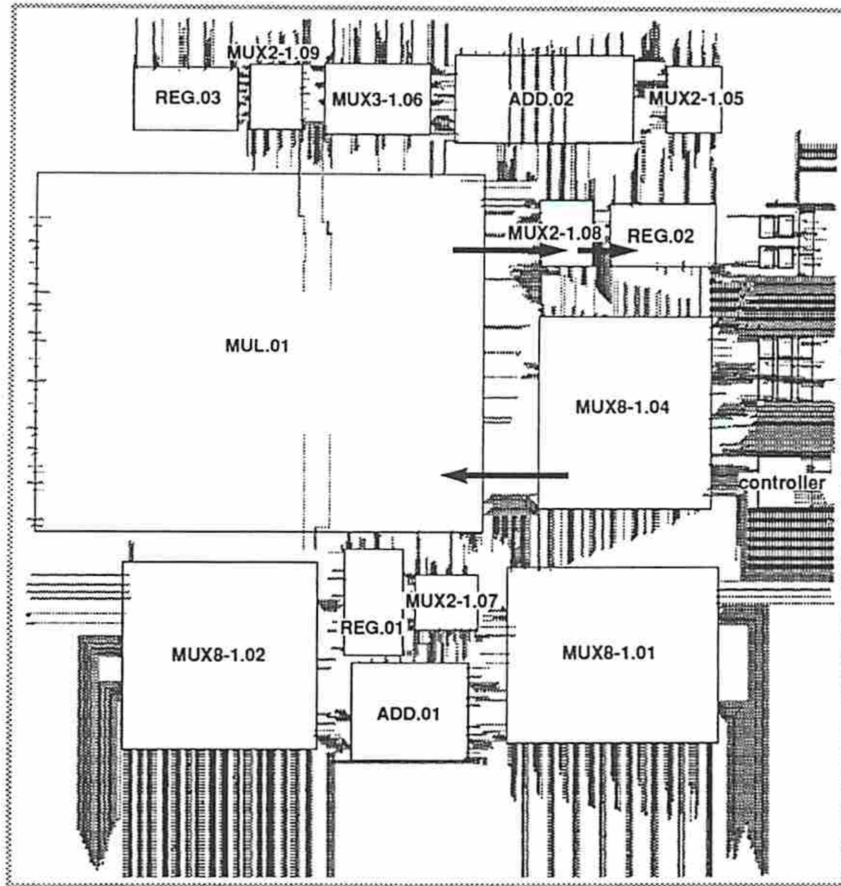


Figure A.5: The Layout using LADS Schedule and Floorplan with Pin Assignment

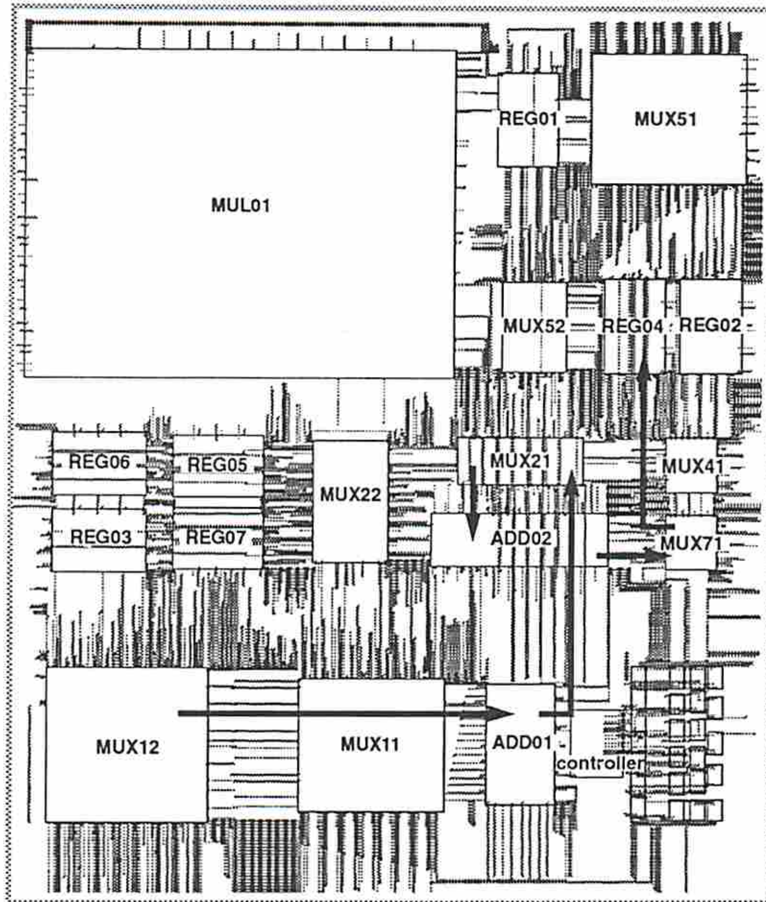


Figure A.6: The Layout of *ChipCrafter Placement I* using FDS Schedule

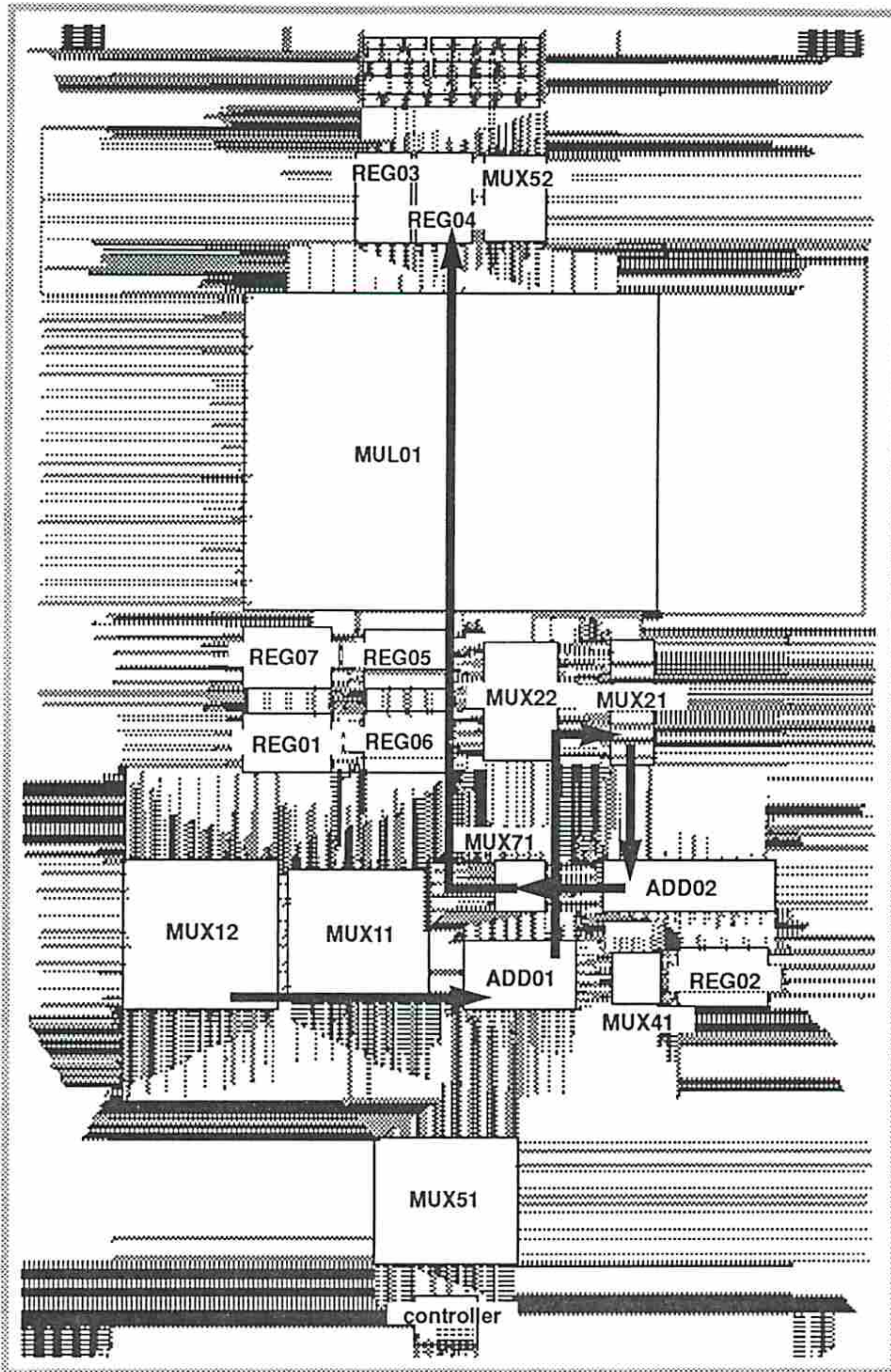


Figure A.7: The Layout of *ChipCrafter Placement II* using FDS Schedule

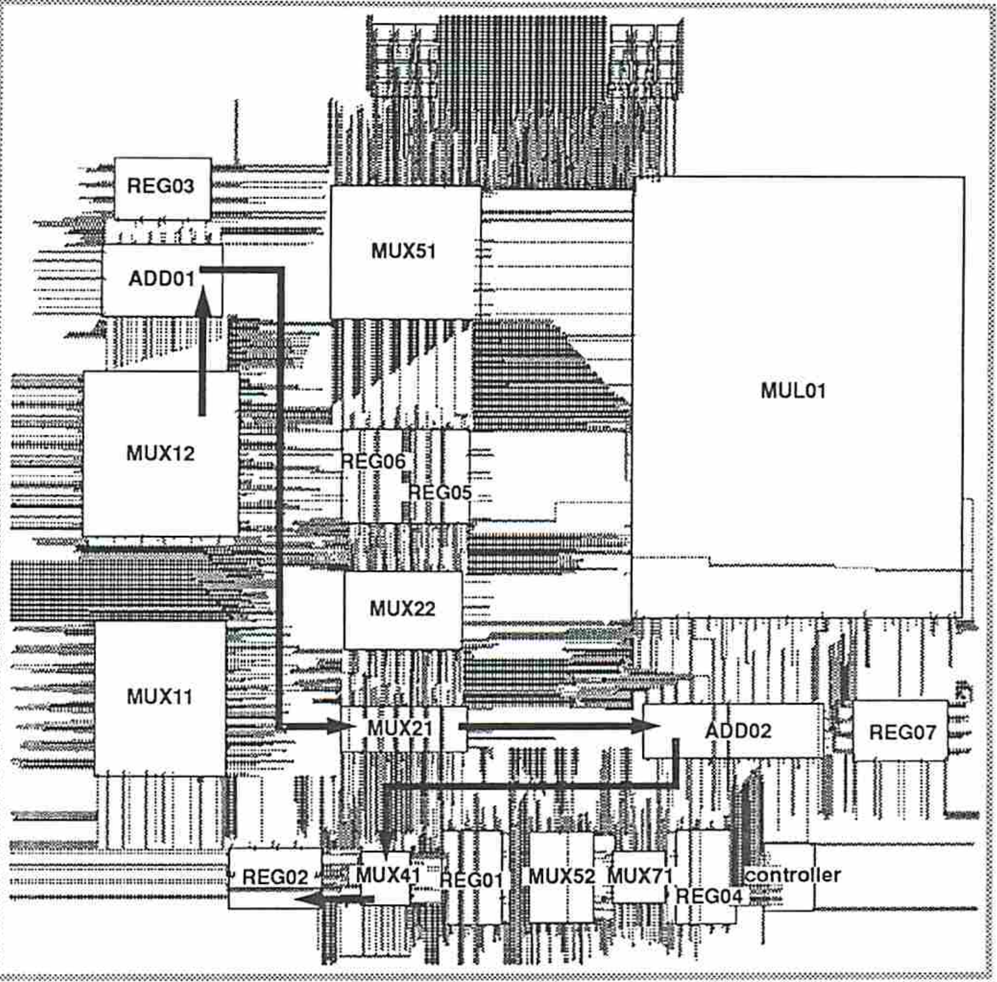


Figure A.8: The Layout of *ChipCrafter Placement III* using FDS Schedule

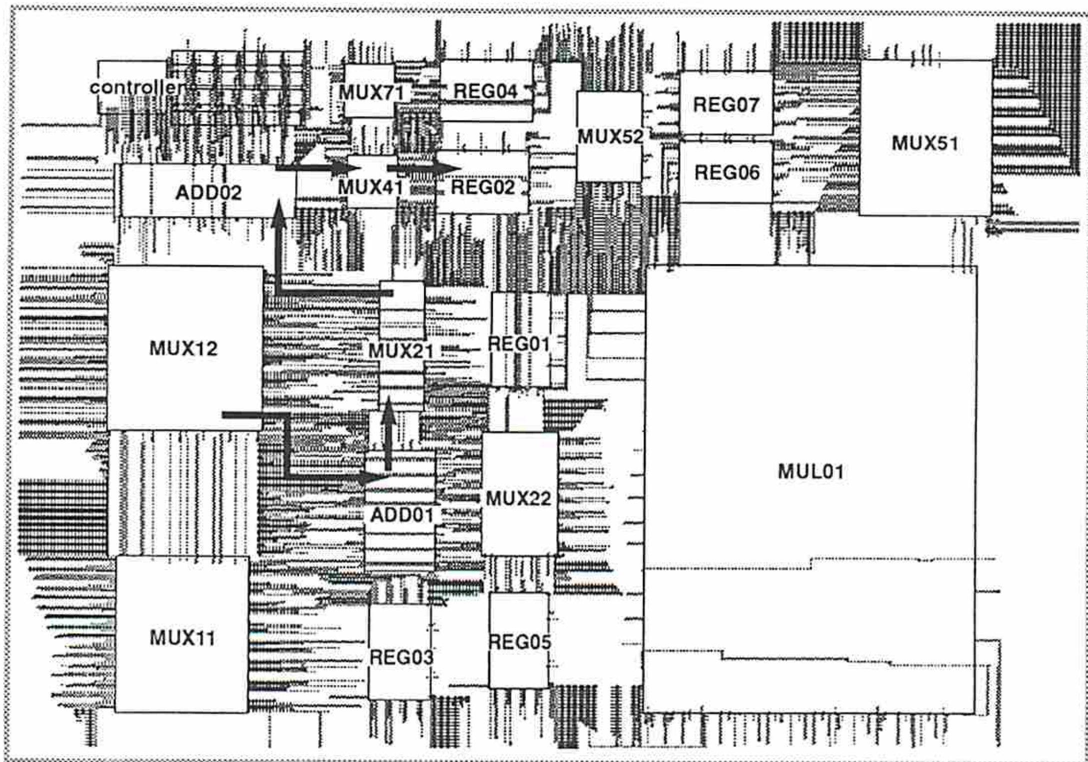


Figure A.9: The Performance-Driven Layout using FDS Schedule without Pin Assignment

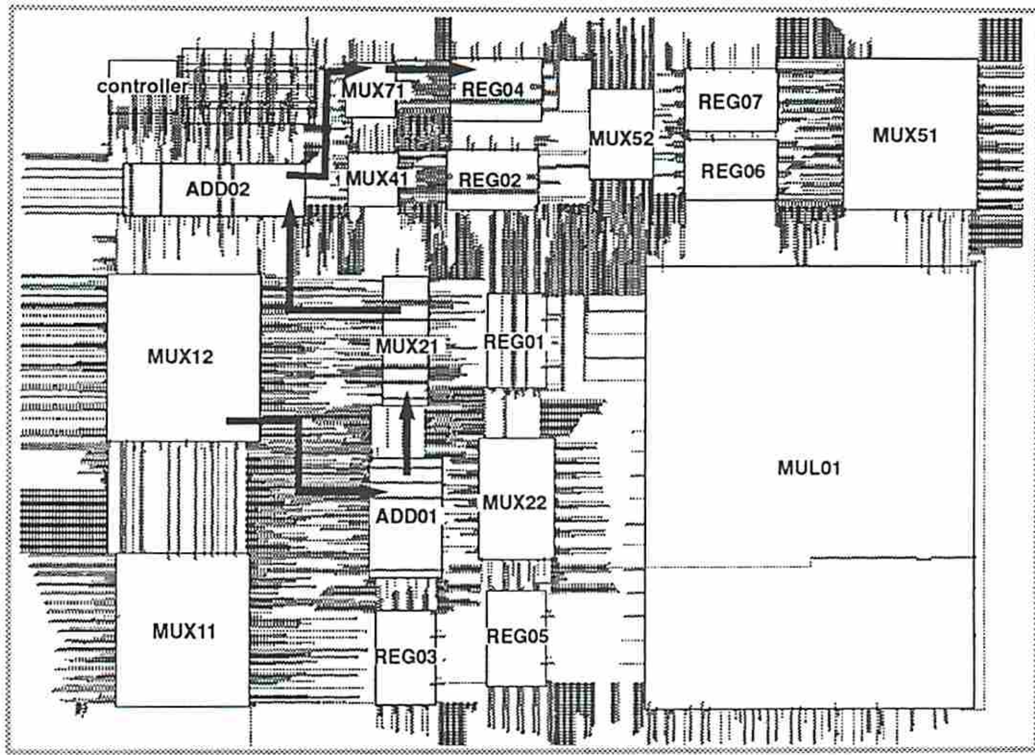


Figure A.10: The Performance-Driven Layout using FDS Schedule with Pin Assignment

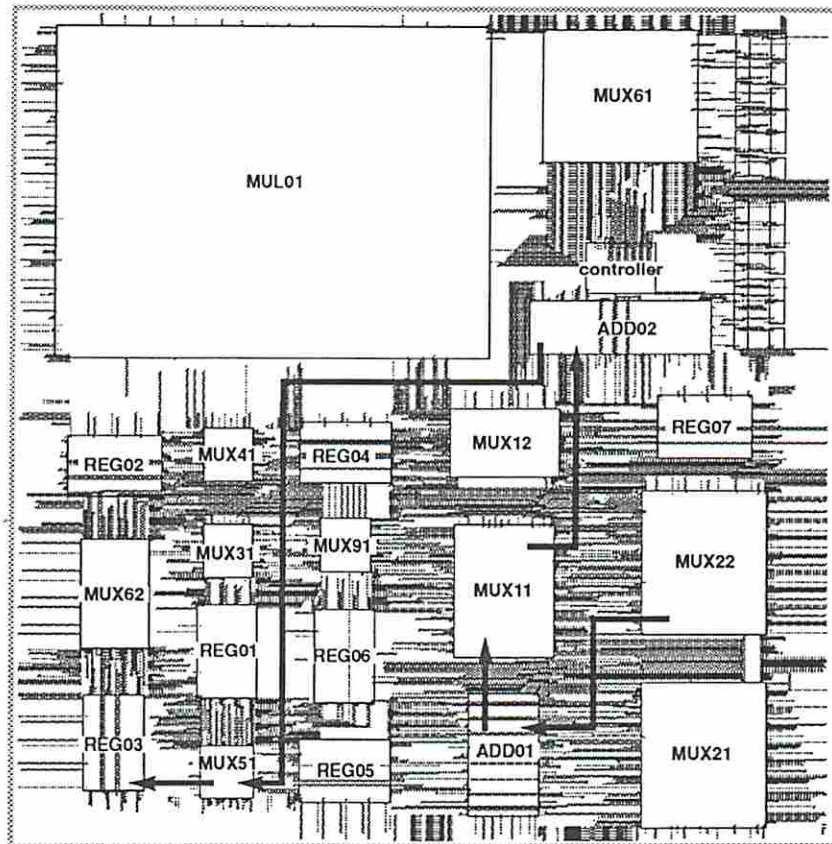


Figure A.11: The Layout of *ChipCrafter Placement I* using MAHA Schedule

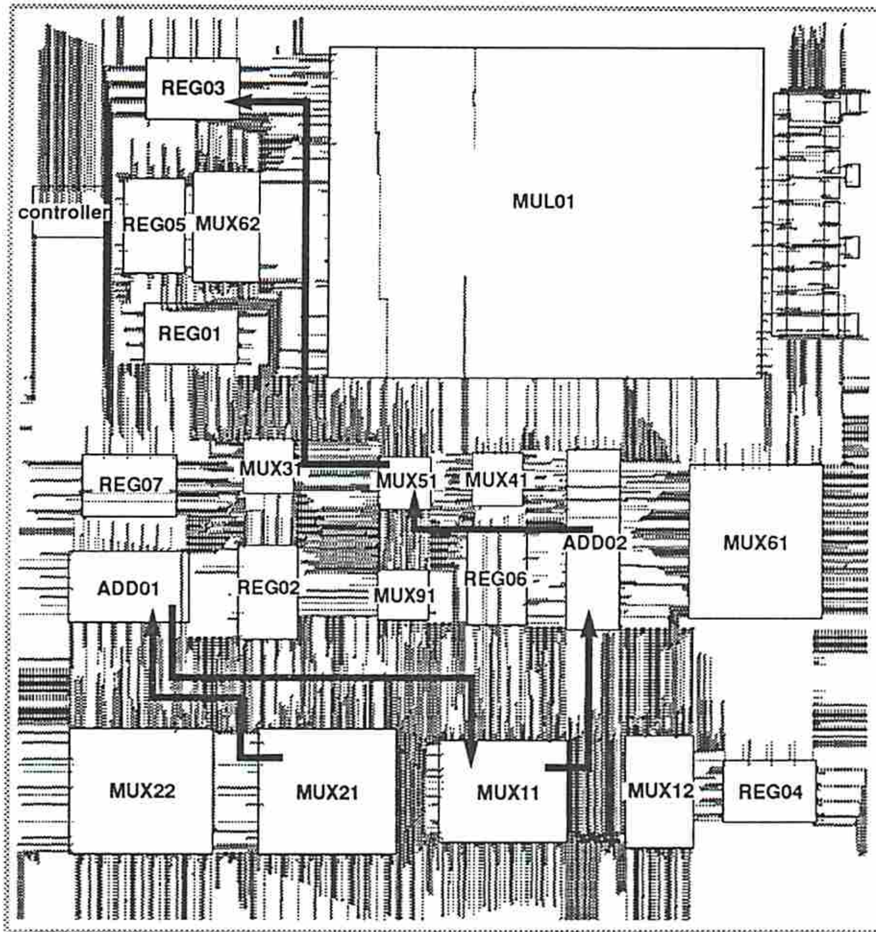


Figure A.12: The Layout of *ChipCrafter Placement II* using MAHA Schedule

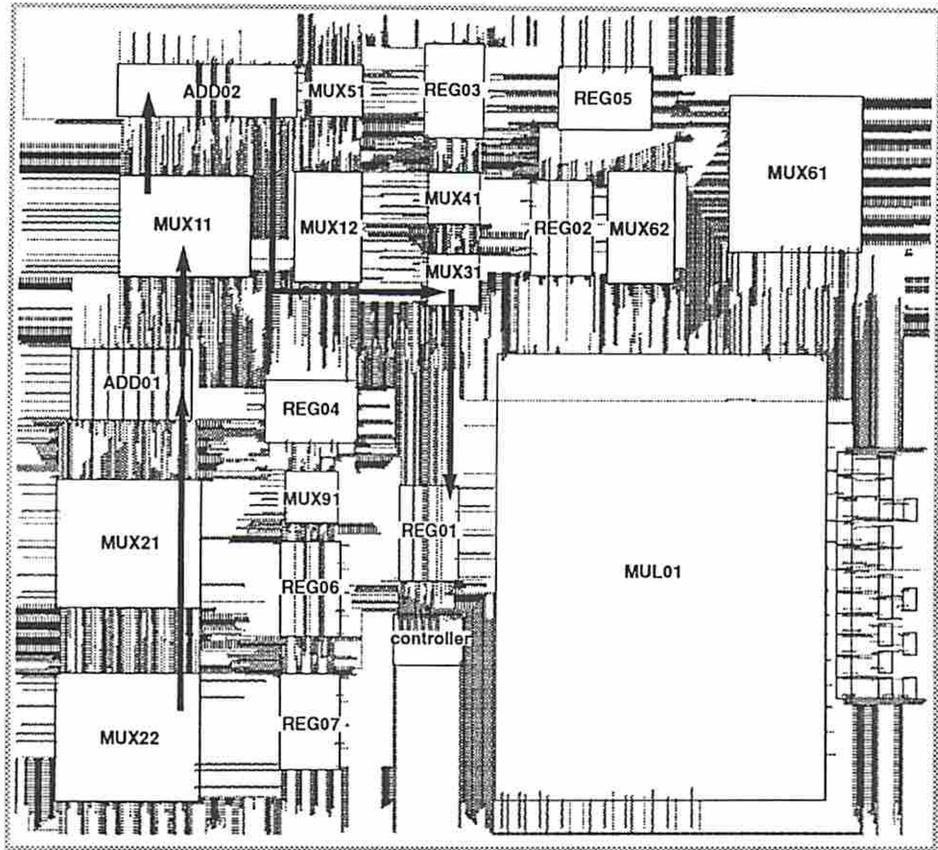


Figure A.13: The Layout of *ChipCrafter Placement III* using MAHA Schedule

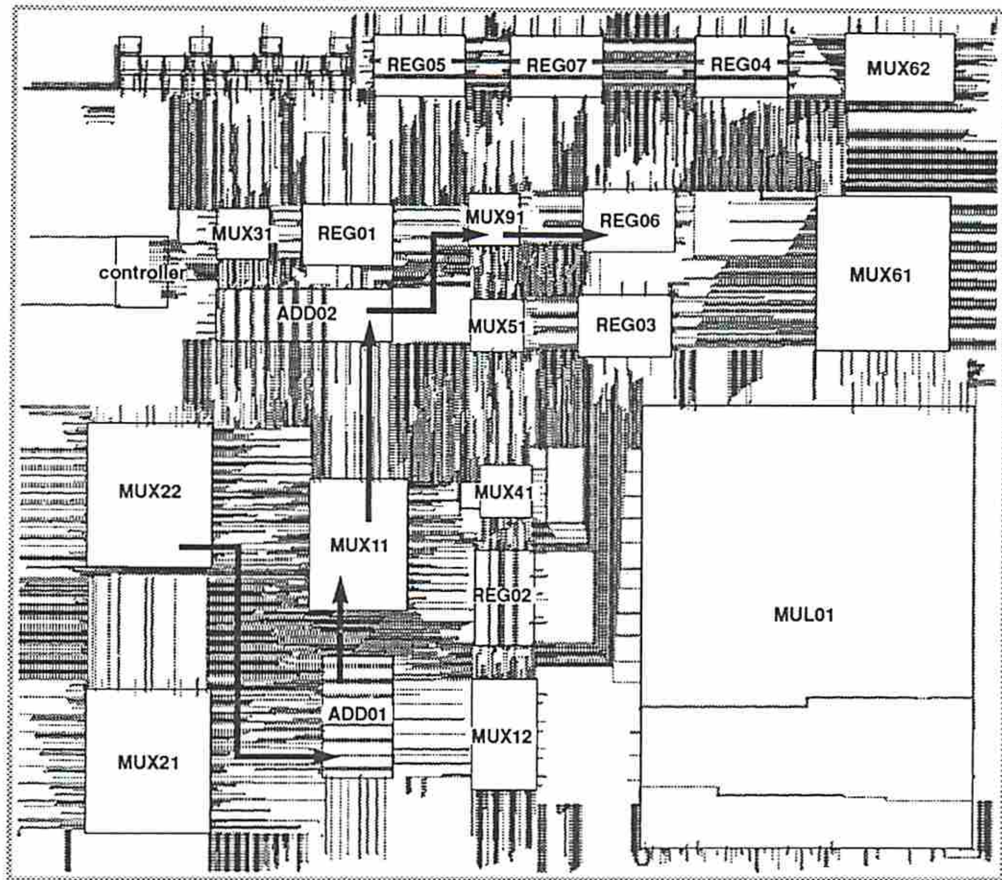


Figure A.14: The Performance-Driven Layout using MAHA Schedule without Pin Assignment

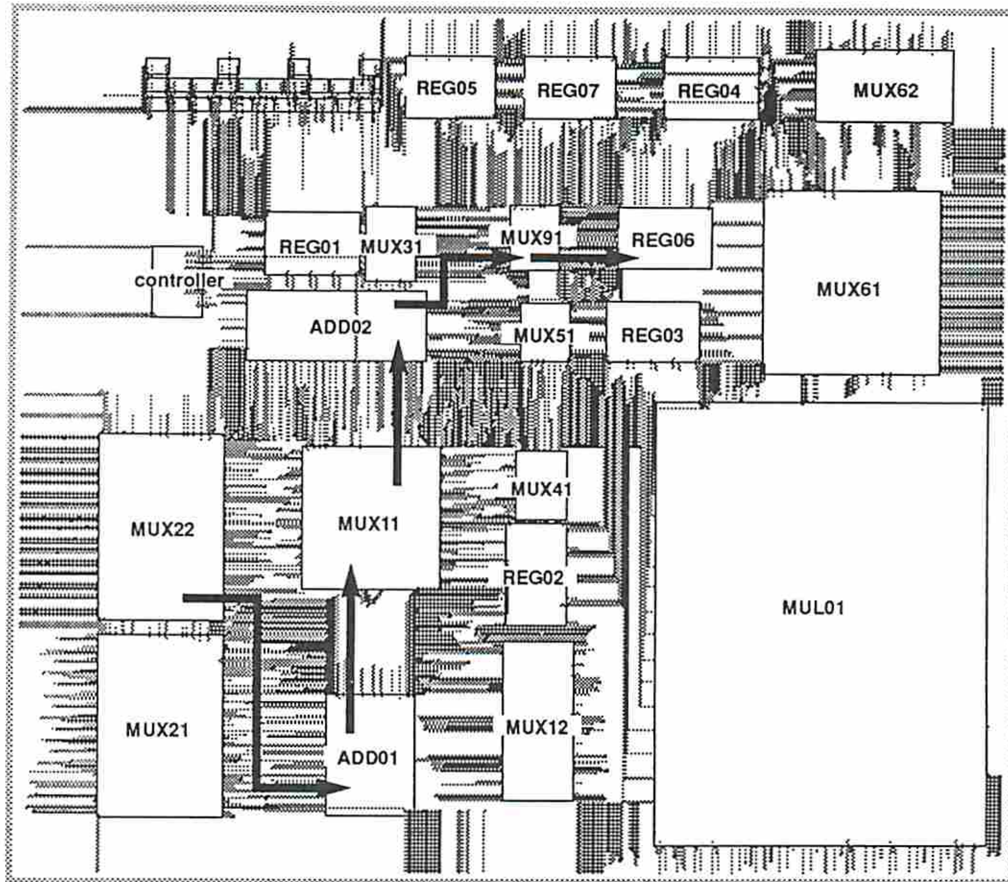


Figure A.15: The Performance-Driven Layout using MAHA Schedule with Pin Assignment

Appendix B

Layouts of the 4-Time-Step Non-Pipelined Differential Equation Solver Example

This appendix shows the partial bindings produced by LADS and the layouts of the 4-time-step non-pipelined differential equation solver example presented in Chapter 6. The layouts were produced using ChipCrafter. Solid lines in the layouts show the critical paths obtained by the timing simulator in ChipCrafter. The figures are scaled in order to reflect the relative dimensions among the layouts. Three layouts were created for this example. *ChipCrafter I* and *ChipCrafter II* use the placements produced by ChipCrafter with different cooling schedules.

Operation	*Operator*	*Register*	*Period*
mul1	MUL01	REG01	2 - 2
mul2	MUL02	REG02	2 - 2
mul3	MUL01	REG01	3 - 3
mul4	MUL02	REG02	3 - 3
mul5	MUL02	REG02	4 - 4
mul6	MUL01	REG01	4 - 4
add1	ADD01	None	N.A.
add2	ADD01	REG03	2 - 2
sub1	SUB01	REG03	4 - 4
sub2	SUB01	None	N.A.
cmp1	CMP01	None	N.A.

Table B.1: The Bindings for the 4-Time-Step Non-Pipelined Differential Equation Solver Example Produced by LADS

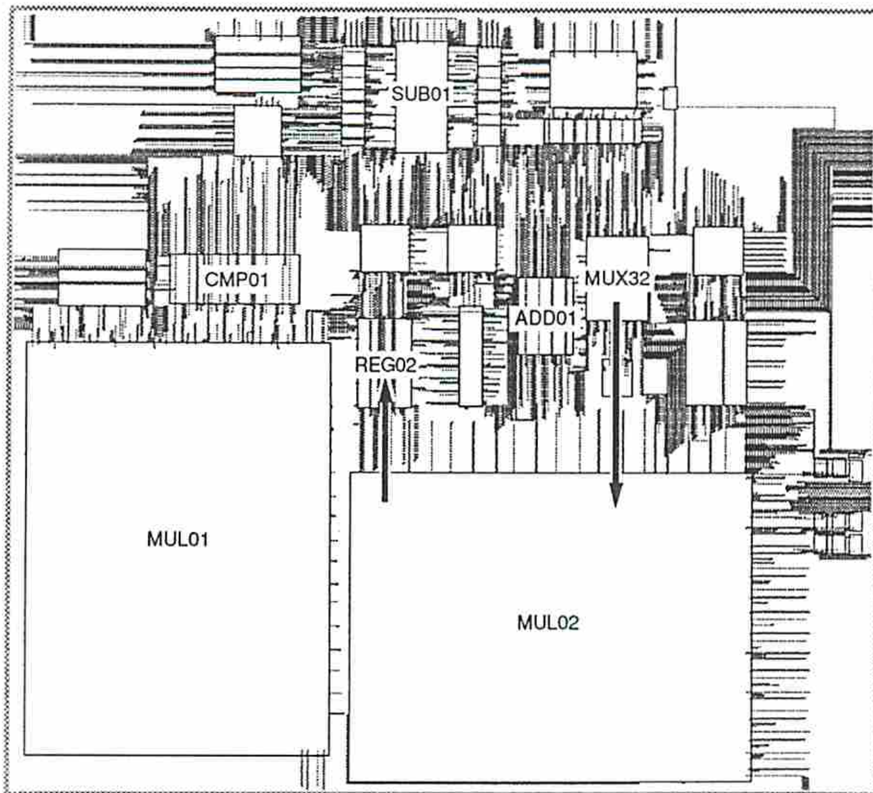


Figure B.1: The Layout of *ChipCrafter I* using LADS Schedule

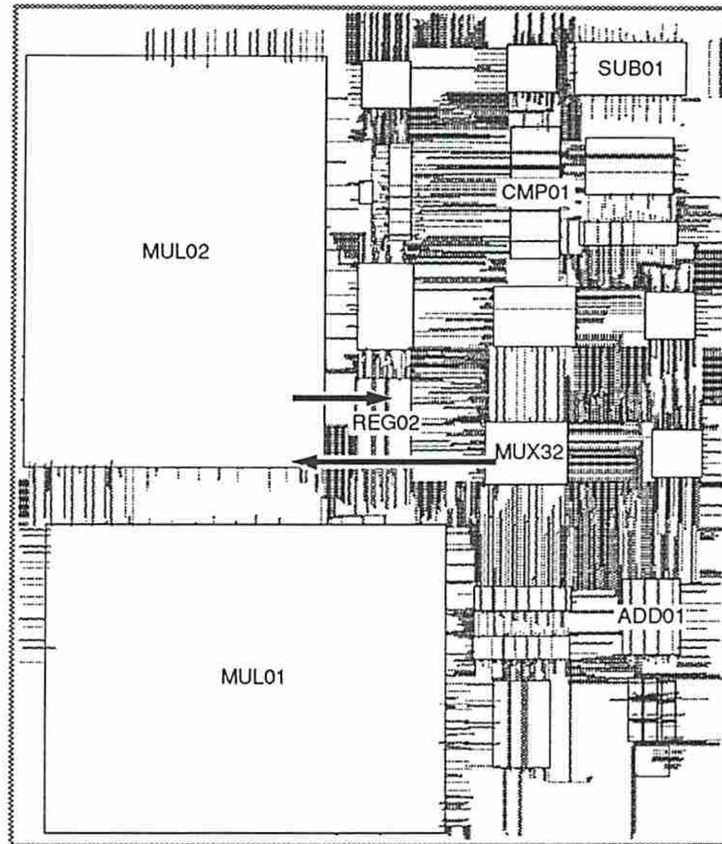


Figure B.2: The Layout of *ChipCrafter II* using LADS Schedule

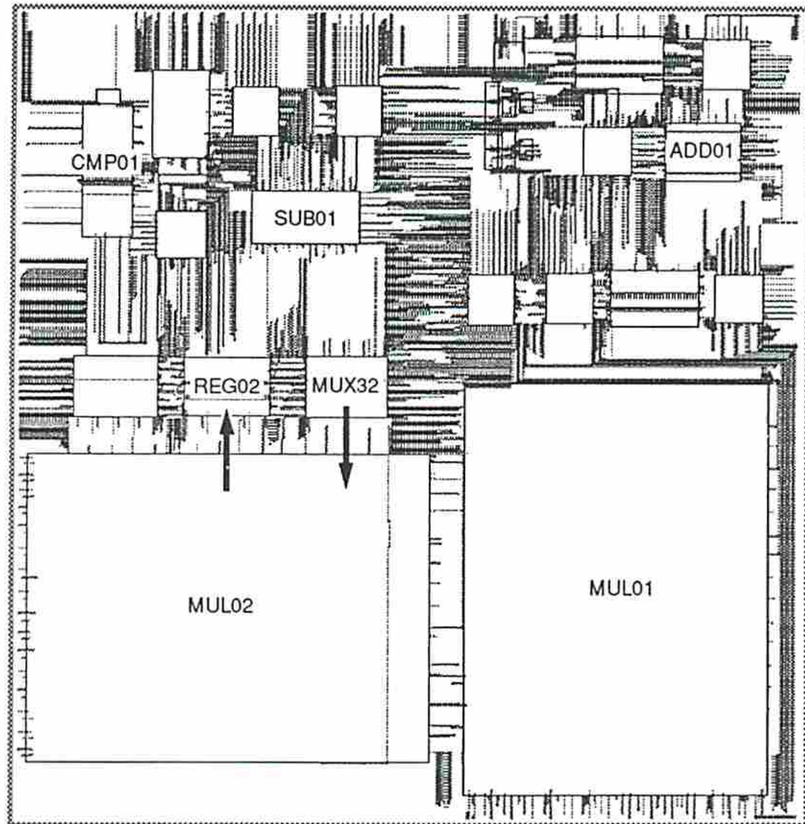


Figure B.3: The Layout using LADS Schedule and Floorplan

Appendix C

Layouts of the 12-Time-Step Non-Pipelined Elliptic Filter Design

This appendix shows the partial bindings produced by LADS and the layouts of the 12-time-step non-pipelined elliptic filter design presented in Chapter 6. The layouts were produced using ChipCrafter. Solid lines in the layouts show the critical paths obtained by the timing simulator in ChipCrafter. The figures are scaled in order to reflect the relative dimensions among the layouts. *ChipCrafter I-V* use the placements produced by ChipCrafter with different cooling schedules. *LADS Placement I* and *LADS Placement II* use the floorplan produced by LADS but with different manual placements for the modules allocated by MABAL. All the layouts were also produced using the buffer resizing program in ChipCrafter to improve design performance.

Operation	*Operator*	*Register*	*Period*
add1	ADD01	REG01	2 - 2
add2	ADD01	REG03	3 - 3
add3	ADD02	REG02	2 - 2
add4	ADD03	REG03	2 - 2
add5	ADD03	REG01	3 - 3
mul1	SQR01	REG03	5 - 5
mul2	SQR01	REG01	4 - 4
add6	ADD01	REG03	6 - 6
add7	ADD01	REG02	5 - 5
add8	ADD02	REG02	6 - 6
mul3	SQR01	REG02	7 - 7
add9	ADD01	REG01	12 - 12
add10	ADD02	REG01	5 - 5
mul8	SQR01	REG01	6 - 6
add11	ADD03	None	N.A.
add13	ADD01	REG01	8 - 8
add14	ADD01	REG03	7 - 7
add15	ADD02	REG03	8 - 8
mul4	SQR01	REG02	9 - 9
add16	ADD01	REG01	10 - 10
add17	ADD02	None	N.A.
add18	ADD02	REG03	9 - 9
mul5	SQR01	REG02	10 - 10
add19	ADD01	REG01	11 - 11
mul6	SQR01	REG02	12 - 12
add20	ADD03	REG01	7 - 7
mul7	SQR01	REG02	8 - 8
add21	ADD02	None	N.A.
add22	ADD02	None	N.A.
add23	ADD03	None	N.A.
add24	ADD01	None	N.A.
add25	ADD01	REG01	9 - 9
add26	ADD02	None	N.A.
add27	ADD02	None	N.A.

Table C.1: The Bindings for the 12-Time-Step Non-Pipelined Elliptic Filter Design Produced by LADS

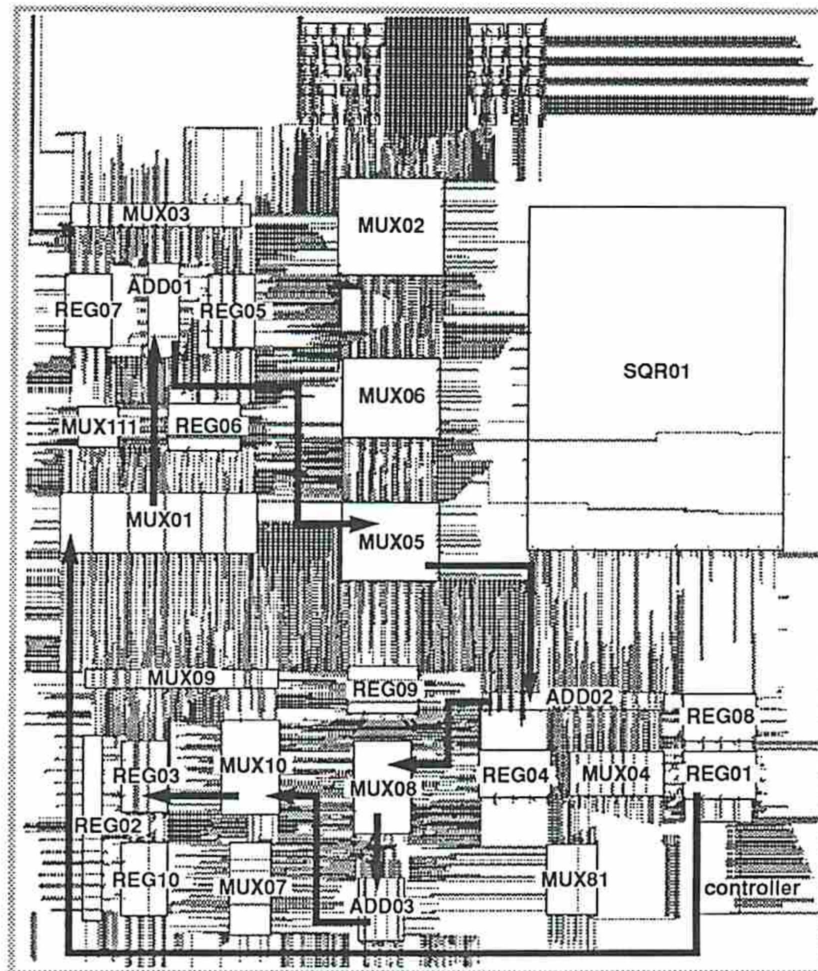


Figure C.1: The Layout of *ChipCrafter Placement I* before Buffer Resizing

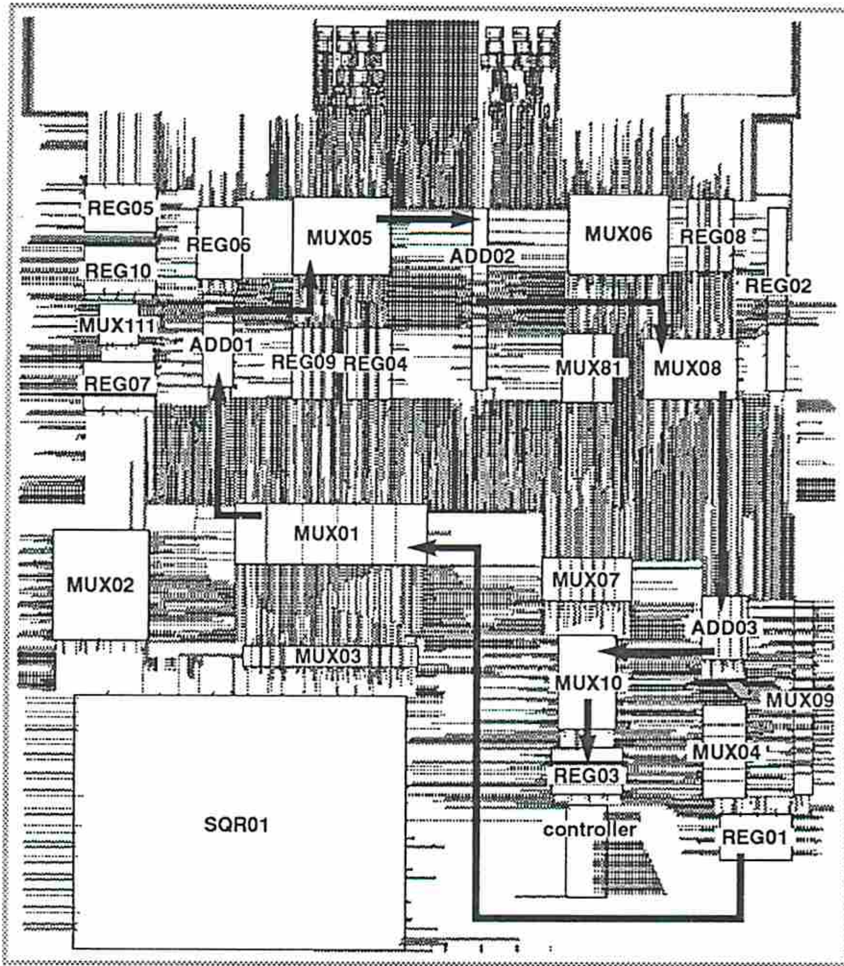


Figure C.2: The Layout of *ChipCrafter Placement II* before Buffer Resizing

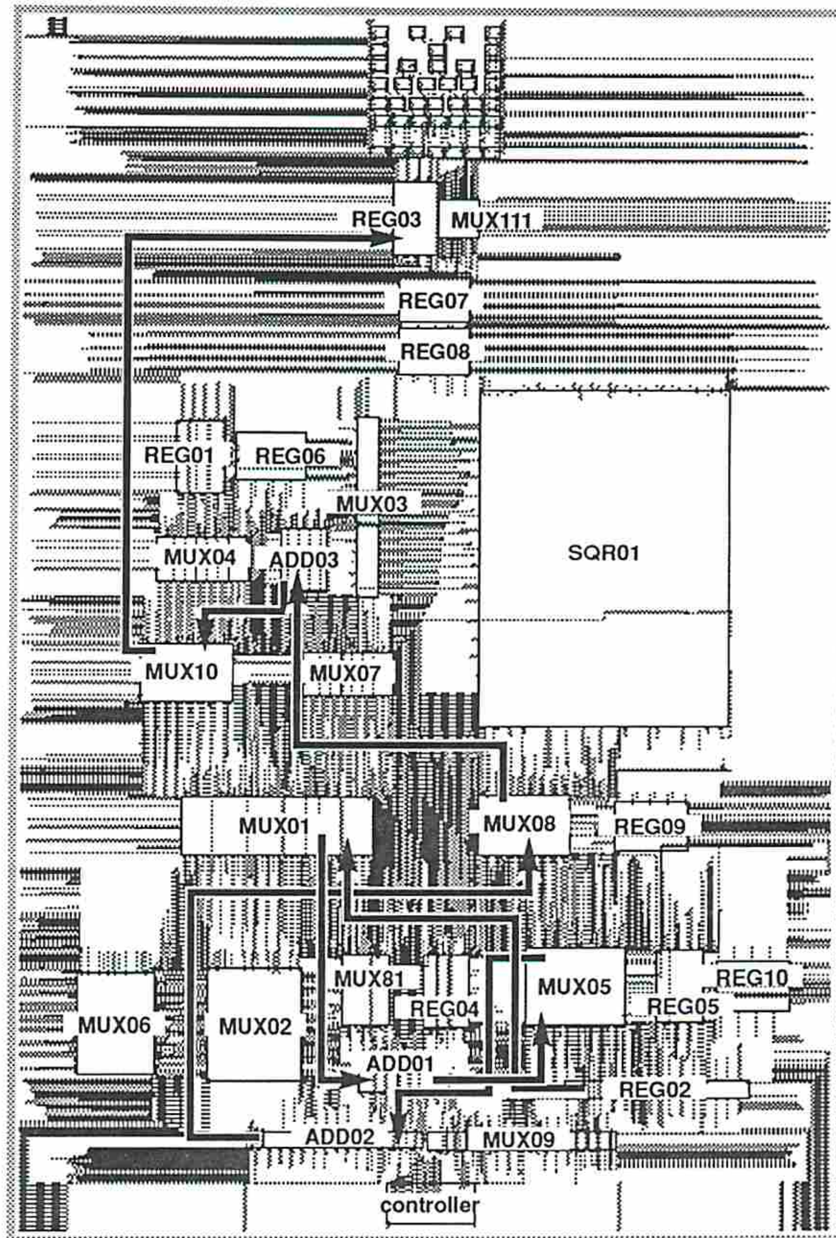
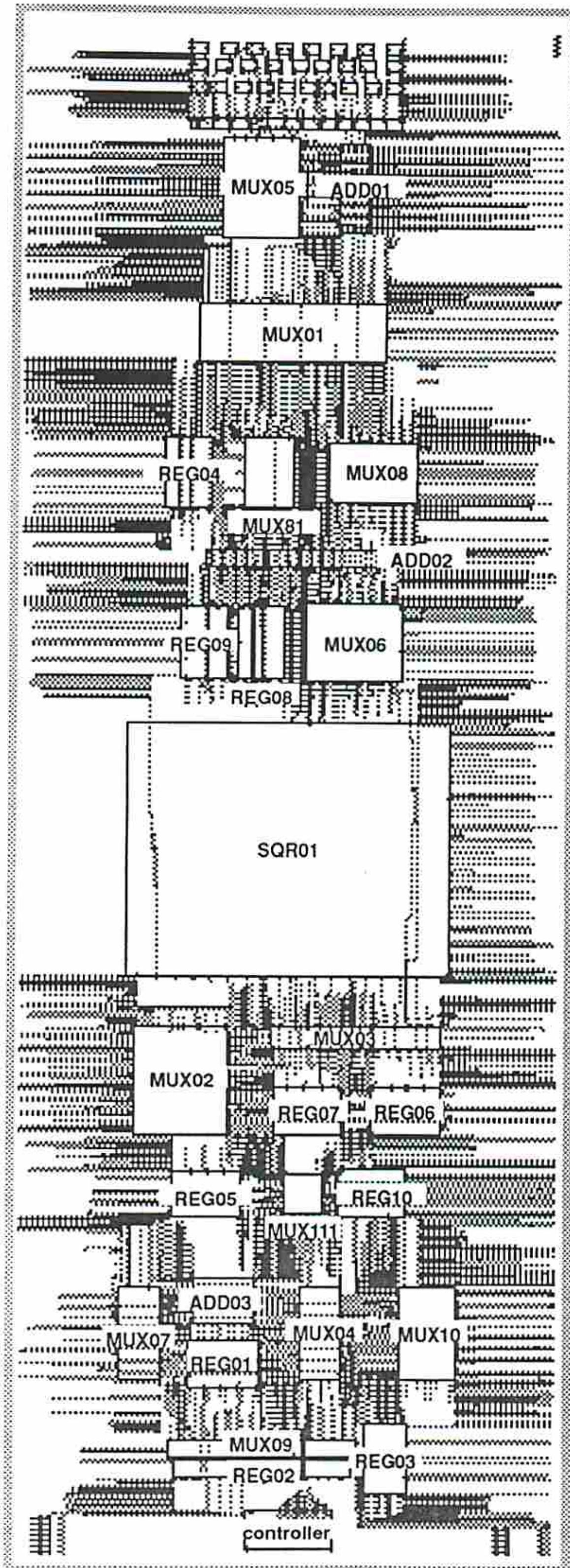


Figure C.3: The Layout of *ChipCrafter Placement III* before Buffer Resizing



Simulation Critical Path:

REG03 -> MUX02 ->

ADD01 -> MUX05 ->

ADD02 -> MUX08 ->

ADD03 -> MUX10 ->

REG03

Figure C.4: The Layout of *ChipCrafter Placement IV* before Buffer Resizing

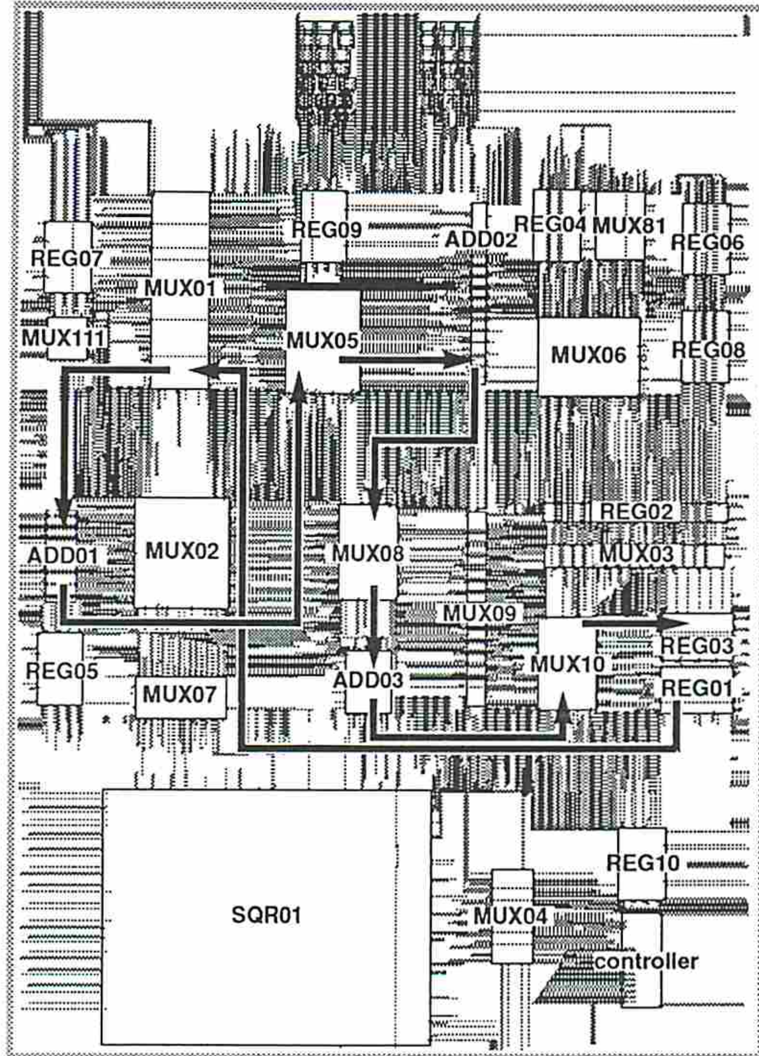


Figure C.5: The Layout of *ChipCrafter Placement V* before Buffer Resizing

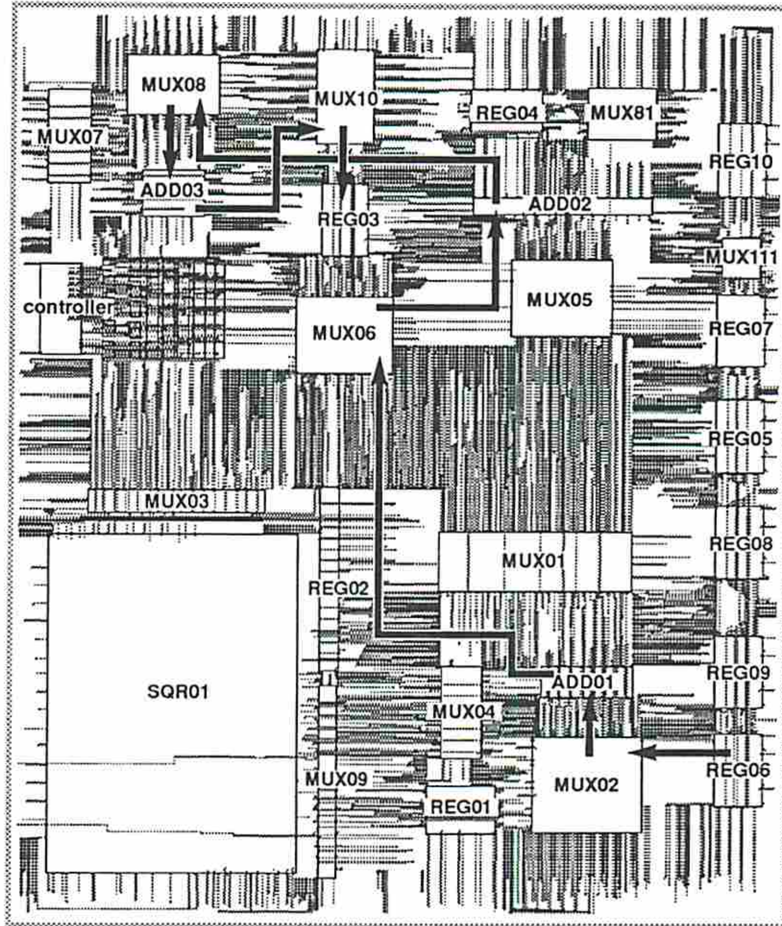


Figure C.6: The Layout using *LADS Placement I* before Buffer Resizing

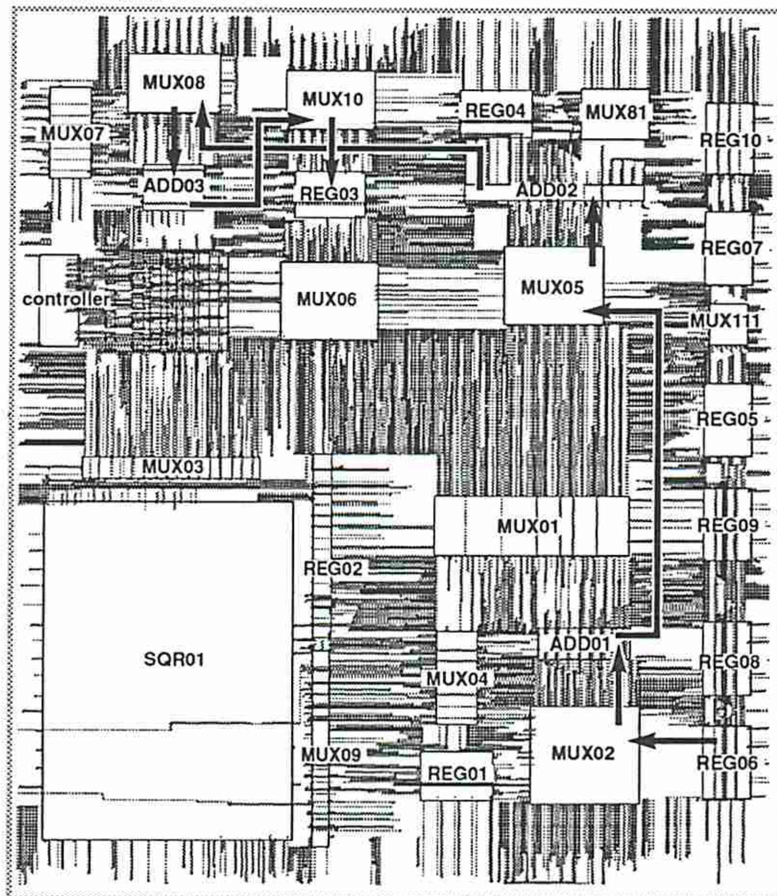


Figure C.7: The Layout using *LADS Placement II* before Buffer Resizing

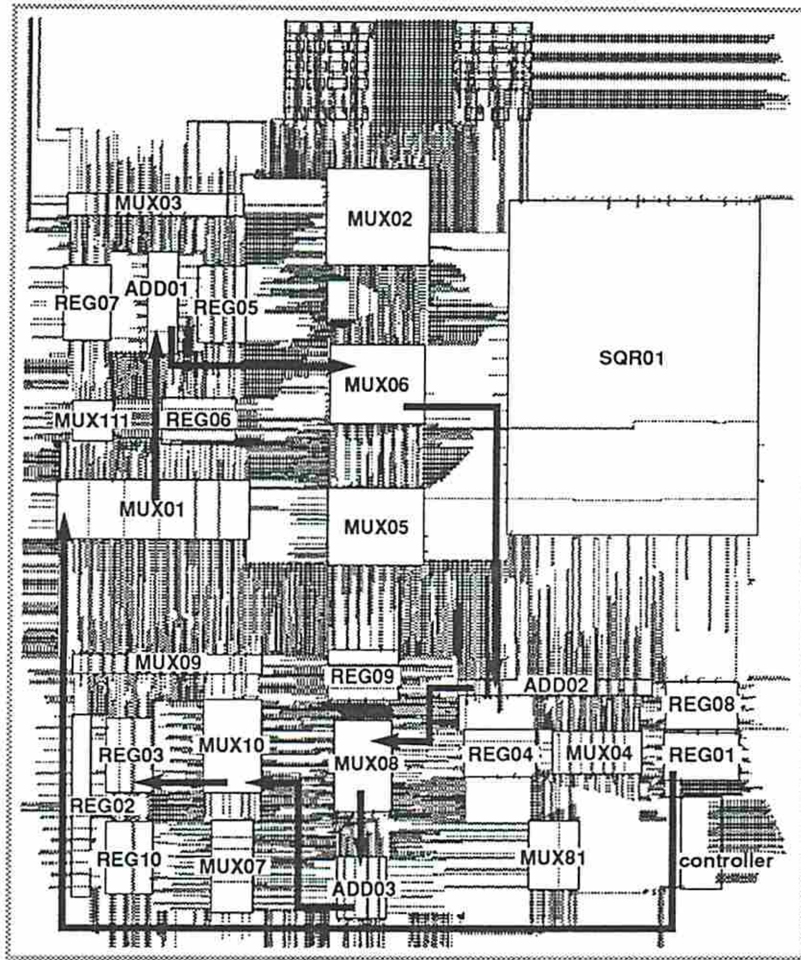


Figure C.8: The Layout of *ChipCrafter Placement I* after Buffer Resizing

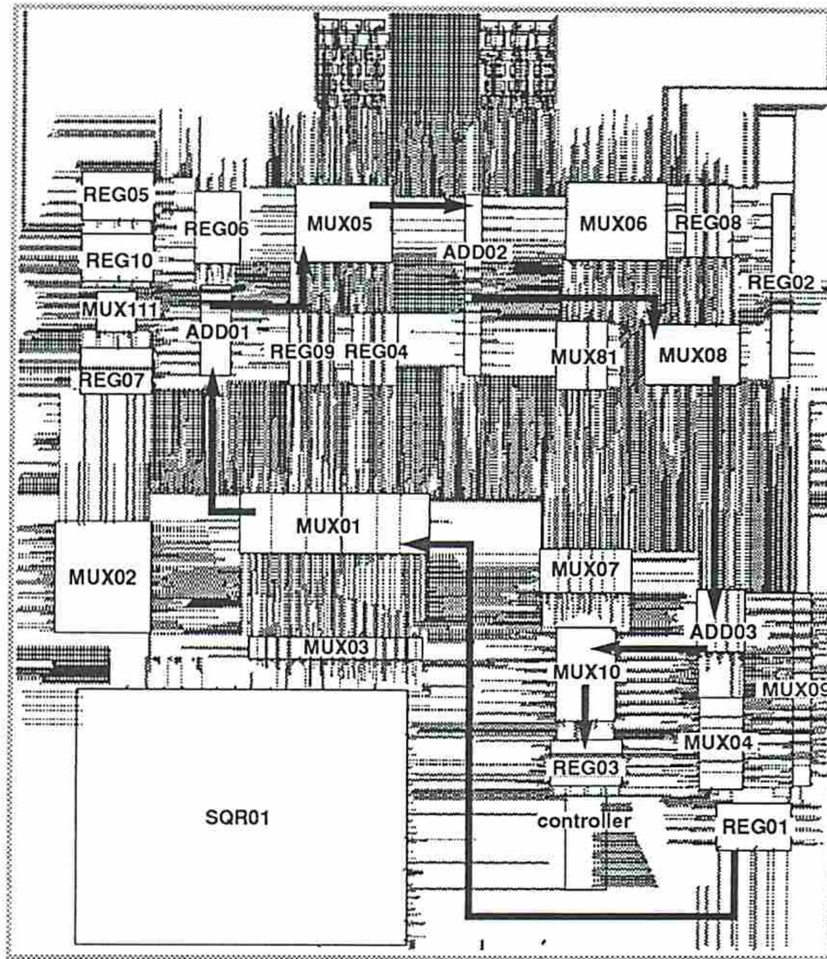


Figure C.9: The Layout of *ChipCrafter Placement II* after Buffer Resizing

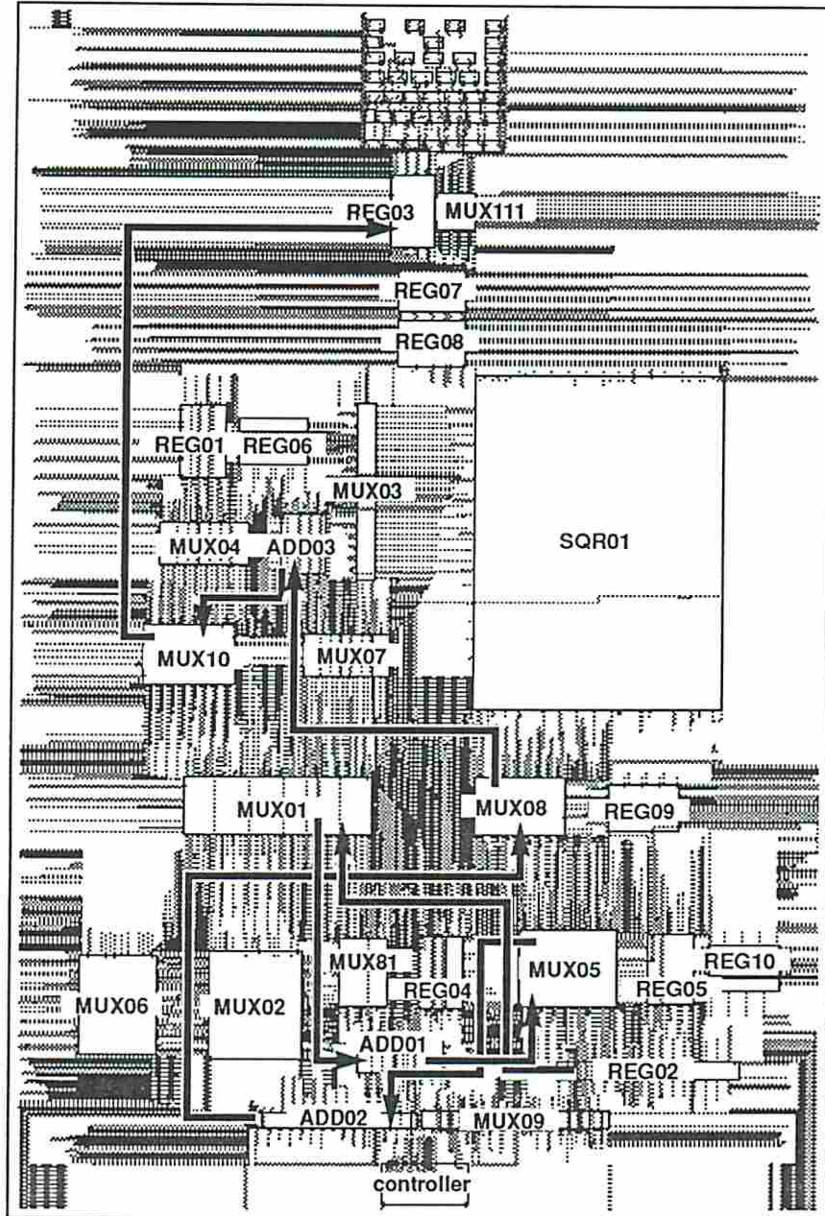
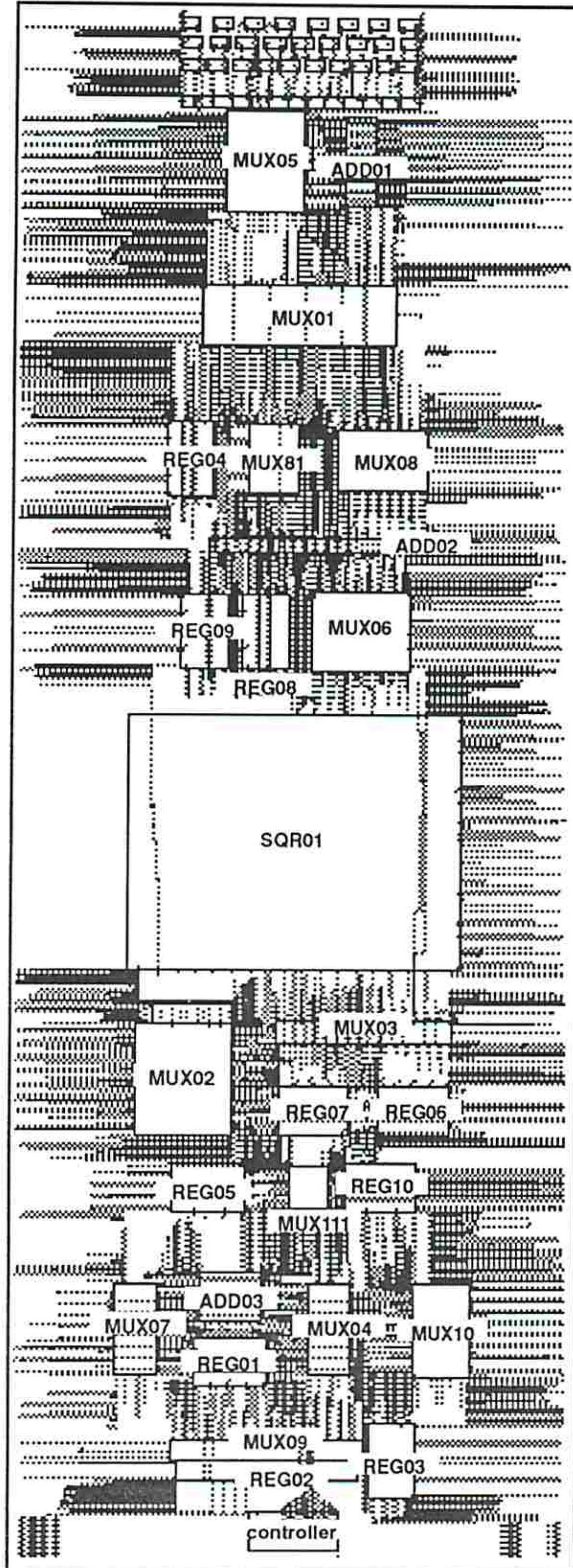


Figure C.10: The Layout of *ChipCrafter Placement III* after Buffer Resizing



Simulation Critical Path:

REG03 -> MUX02 ->

ADD01 -> MUX05 ->

ADD02 -> MUX08 ->

ADD03 -> MUX10 ->

REG03

Figure C.11: The Layout of *ChipCrafter Placement IV* after Buffer Resizing

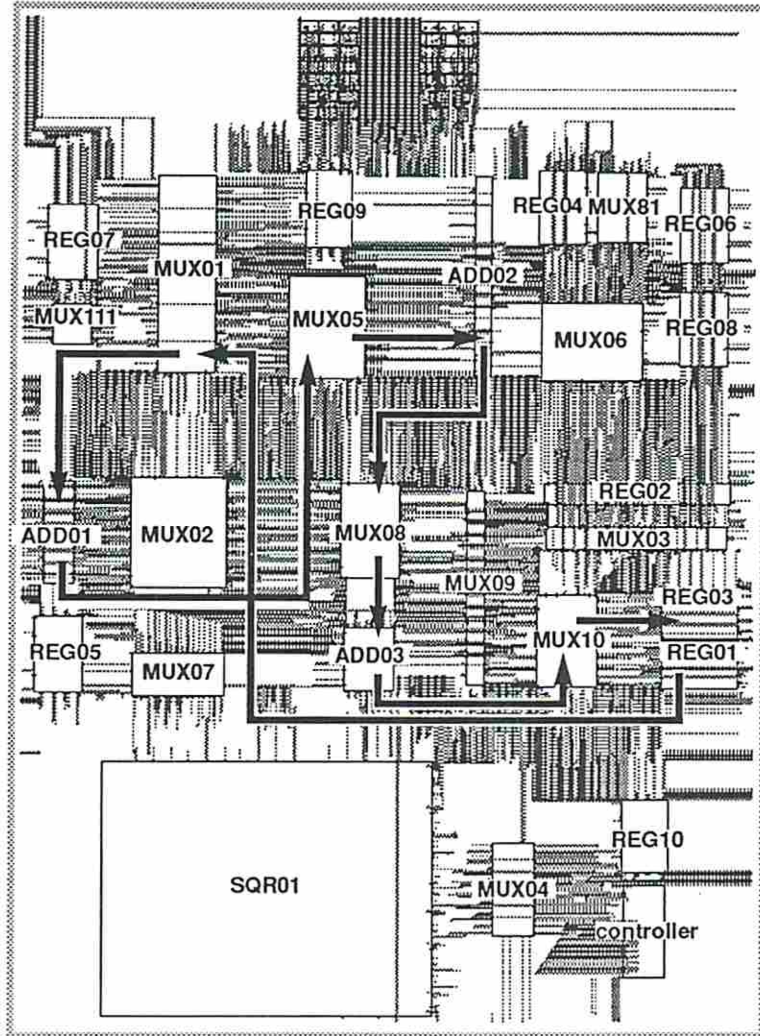


Figure C.12: The Layout of *ChipCrafter Placement V* after Buffer Resizing

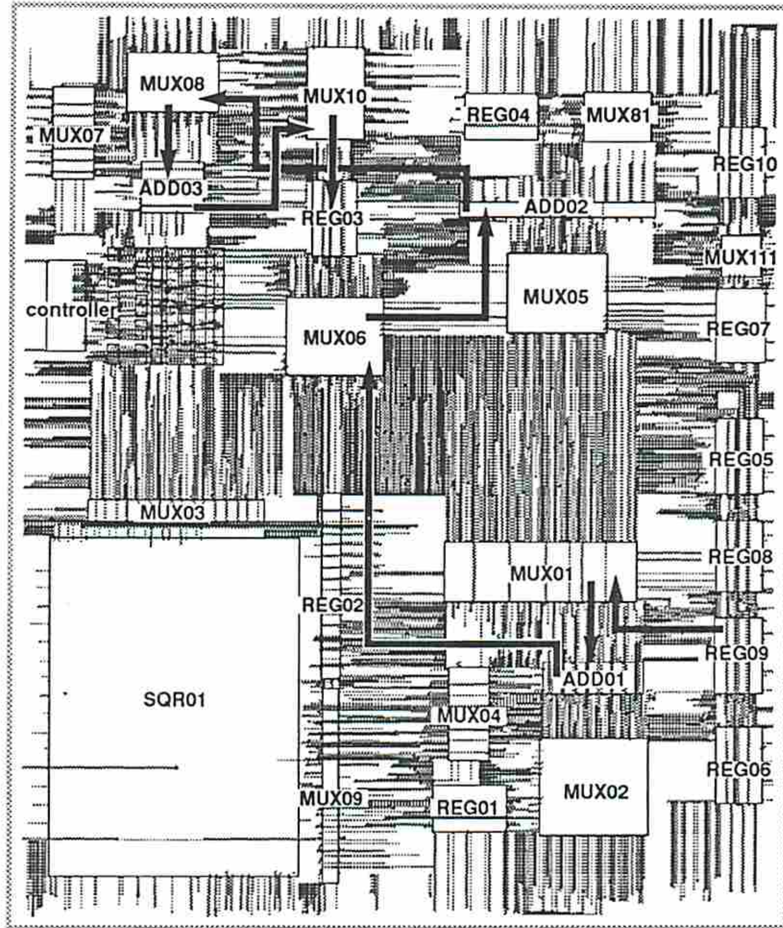


Figure C.13: The Layout using *LADS Placement I* after Buffer Resizing

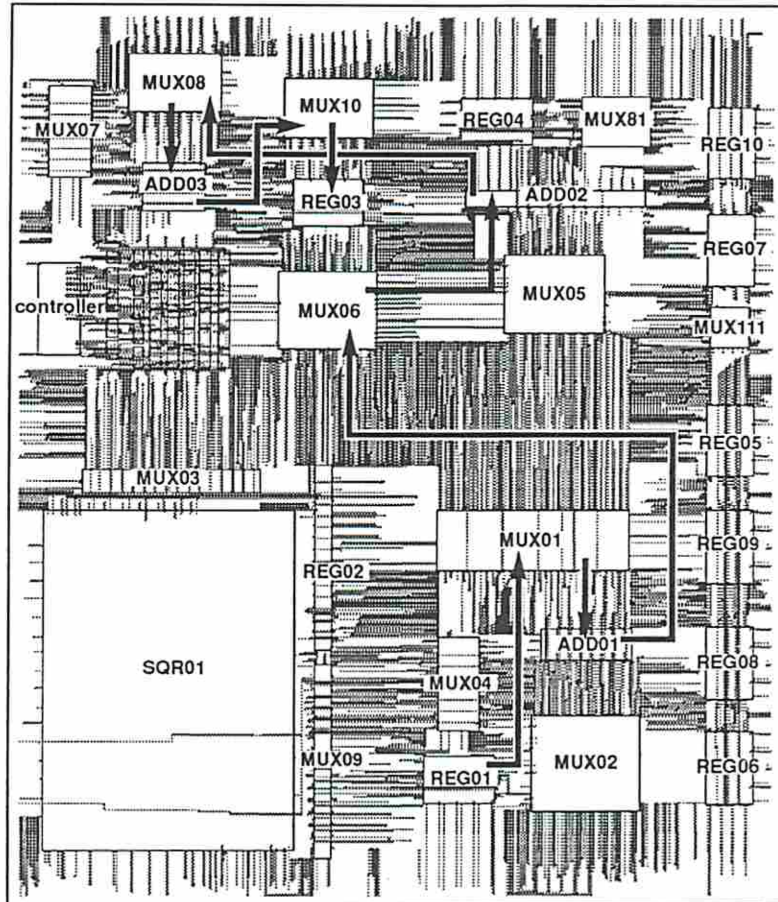


Figure C.14: The Layout using *LADS Placement II* after Buffer Resizing