Control Path/Data Path Tradeoffs in VLSI Design

Mitchell J. Mlinar

CEng Technical Report 91-16

A Dissertation Presented to the
FACULTY OF THE GRADUATE SCHOOL
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
DOCTOR OF PHILOSOPHY
(Computer Engineering)

(Copyright May 1991)

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, CA. 90089-2562

June, 1991

Dedication

To my wife, Sharon
and my children, Alison, Eric, and Rebecca.

You who endured the sweat and the tears,
can now share in the glory.

Acknowledgments

I would like to thank my advisor, Prof. Alice Parker, under whose direction this thesis formed. Only she can appreciate the amount of time spent towards my guidance and technical growth. I would also like to thank Professors Melvin Breuer and Dennis Estes for serving on my thesis committee. Their suggestions and comments have been most helpful.

Two other faculty members have also contributed in part to this thesis. Technical discussions with Professor Sarma Sastry have proved most illuminating. Professor James Yee opened new doors which exposed me to alternative paths and provided new insight on many problems.

I would also like to thank my friends in the "pits": Jorge Seidel, Meera Balakrishna, Kayhan Küçükçakar, and Shiv Prakash. They made the long years more bearable.

Special thanks go to my collegue, Rajiv Jain. His loyalty and friendship helped to keep my "eyes on the prize" even in the worst of times. Long talks with Sally Hayati also kept my sanity intact. My PhD was far more interesting with you two around.

I also acknowledge the support of TRW; their financial aid during the early years is appreciated. Research assistance provided by the the Defense Advanced Research Projects Agency (Contract N00014-87-K-0861) was also helpful. Last, but not least, is my current employer EEsof who has quietly tolerated my school schedule.

To my three children - Alison, Eric, and Rebecca - your welcome interruptions kept my goal in perspective. I would also like to thank my mother for getting me started on this long path.

Finally, and most important, my wife deserves a large part of the credit. Her unlimited patience and support have turned my dream into a reality.

Contents

Dedication ii

Acknowledgments iii

List Of Figures xii

List Of Tables xvi

1 Introduction 1

1.1 Problem Statement 2
1.2 Motivation 4
1.3 Problem Approach 6
1.3.1 Data Path Analysis 6
1.3.2 Routing and Storage Analysis 7
1.3.3 Control Path Analysis 8
1.3.4 System-Level Partitioning 10
1.4 Evaluation Tools within ADAM 13
1.4.1 SLIMOS: Module Set Selection 13
1.4.2 MAHA: Non-Pipelined Data Path Synthesis 13
1.4.3 Sehwa: Pipelined Data Path Synthesis 14
1.4.4 HAPPE: Data Path Estimation 14
1.4.5 MABAL: Multiplexer and Bus Allocation 15
1.4.6 Berkeley PLA Synthesis Package 15
1.4.7 PASTA: PLA Control Area Estimator 16
1.4.8 PLEST Data Path Area Estimator 17
1.5 Related Work 17

1.5.1 Data Path Synthesis 17
1.5.2 Control Path Synthesis 18
1.5.3 Synthesis considering both control and data path analysis 20
1.5.4 High-level hardware/firmware/software partitioning 23
1.6 Thesis Outline 24

2 Extensions to Data Path Synthesis **26**
2.1 Introduction 26
2.2 Overview of the MAHA Algorithm 28
2.3 Evolution of MAHA 31
2.3.1 Synthesis of Serialized Designs 32
2.3.2 Register Extensions 33
2.3.3 Multiplexer Extensions 35
2.3.4 Extension for Localized Constraints 36
2.4 The MAHA Program Structure 36
2.4.1 Algorithm Input 37
2.4.1.1 Dataflow Graph Input 37
2.4.1.2 Library Generation 38
2.4.1.3 Local Constraint Specification 38
2.4.1.4 Global Constraints 39
2.4.2 Clock Cycle Generation 39
2.4.3 Critical Path Partitioning and Allocation 40
2.4.4 Off-Critical Path Analysis and Allocation 46
2.4.5 Dataflow Graph Stretching 48
2.4.6 MAHA Synthesis Procedure 50
2.4.7 Runtime Analysis 55
2.5 Loop Transformation in Data Path Synthesis 56
2.5.1 Simple Loop Transformation 57
2.5.2 Transformation Algorithm 61
2.5.3 Runtime Analysis for Loop Transformation 67
2.6 Examples and Synthesis Results 67
2.7 Limitations of MAHA 90
2.8 Summary 102

3 **Area Estimation of Data Path Controllers** **105**
3.1 Introduction 105
3.2 A PLA Control Area Model 110
3.2.1 General Model 110
3.2.2 Computing R_c and p 112
3.2.2.1 Stage Control Method 112
3.2.2.2 Register Control Method 115
3.2.2.3 Multiplexer/Bus Control Lines 118
3.2.3 A Specific Model for Berkeley PLA Synthesis Tools 123
3.3 Loops and Conditionals 126
3.3.1 Effect of Loops on PLA Area 126
3.3.1.1 Loop Unrolling 128
3.3.1.2 Implementing Loops using a Counter 129
3.3.2 Effect of Conditional Branches on PLA Area 133
3.4 Extending the PLA Model 138
3.4.1 Estimation of Folded PLA Area 138
3.4.2 Extensions for Pipelined Controllers 140
3.5 Validation of the PLA Model 142
3.5.1 Validation Process 142
3.5.1.1 Basic Model 143
3.5.2 Loops and Conditionals 145
3.5.3 Pipelined Designs 153
3.5.4 PLA Column Folding 153
3.5.5 Use of PASTA 153
3.5.6 Shortcomings of Using the Berkeley Toolset 155
3.6 Summary 157

4 **Predicting Register and Multiplexer Requirements** **159**
4.1 Introduction 159
4.1.1 System to be Modeled 160
4.2 Register Area in Non-pipelined Designs 161
4.2.1 Register Bounds in Non-Pipelined Designs 164
4.2.2 Estimating Register Use in Non-Pipelined Designs 175

4.3 Register Area in Pipelined Designs 178
4.3.1 Register Bounds in Pipelined Designs 178
4.3.2 Estimating Register Use in Pipelined Designs 181
4.4 Predicting the Number of Microcycles 184
4.5 Multiplexer Area 189
4.5.1 Theoretical Bounds on Multiplexer Area 189
4.5.2 Estimating Multiplexers in Non-Pipelined Designs 190
4.5.2.1 Multiplexer Area with Operators in Non-Pipelined Design 192
4.5.2.2 Multiplexer Area with Registers in Non-Pipelined Design 195
4.5.2.3 Example for Non-Pipelined Design 195
4.5.3 Estimating Multiplexers in Pipelined Designs 197
4.5.3.1 Multiplexer Area with Operators in Pipelined Design 199
4.5.3.2 Multiplexer Area with Registers in Pipelined Design 201
4.6 Experiments and Validation 201
4.7 Summary 208

5 Analysis of Control Path/Data Path Tradeoffs 212
5.1 Introduction 212
5.2 Types of Control Path/Data Path Tradeoffs 212
5.2.1 Composition of Operators 213
5.2.2 Operator Decomposition 214
5.2.3 Sequential/Combinational Tradeoffs 214
5.3 Control Path/Data Path Bitwidth Tradeoff Model 219
5.3.1 A Simple Model 220
5.3.2 Bit-dependent Operators 227
5.3.3 Special Operator Bitwidth Considerations 230
5.3.4 Example using the small model 231
5.3.5 Limitations of the Control/Data Bitwidth Model 233
5.4 Summary 236

6 **Control Path/Data Path Tradeoff Evaluation** **237**
6.1 Introduction . . . 237
6.2 Control Path/Data Path Tradeoffs Evaluation . . . 237
6.2.1 A Methodology for Evaluating Tradeoffs . . . 238
6.2.2 Synthesis versus Prediction: An Example . . . 240
6.2.2.1 Design Synthesis . . . 240
6.2.2.2 Design Prediction . . . 242
6.2.2.3 Synthesis versus Prediction . . . 243
6.3 Tradeoff Examples . . . 250
6.3.1 Bitwidth Tradeoffs . . . 252
6.3.1.1 Non-Pipelined Bit Serialization: Floating Point Processor . . . 252
6.3.1.2 Pipelined Bit Serialization: Elliptical Filter . . . 258
6.3.2 Composition Tradeoffs . . . 264
6.3.2.1 Merging Different Bitwidths: Temperature Controller . . . 264
6.3.2.2 ALU Substitution: Floating Point Coprocessor . 268
6.3.3 Multi-processor Tradeoffs . . . 271
6.3.4 Tradeoff Trends . . . 275
6.4 Summary . . . 279

7 **Conclusions and Future Research** **280**
7.1 Contributions . . . 280
7.2 Automating System-Level Tradeoffs . . . 282
7.2.1 Detection of Tradeoff Locations . . . 282
7.2.2 Ordering System-Level Tradeoffs . . . 284
7.2.3 Automated Application of Tradeoffs . . . 285
7.3 Future Research . . . 285

Appendix A
Notation . . . 295

Appendix B
MAHA Usage: Inputs and Outputs . . . 298

B.1 MAHA Inputs . . . 298
B.1.1 Node Description . . . 300
B.1.2 Edge Description . . . 300
B.1.3 Module Description . . . 301
B.2 Executing MAHA . . . 303
B.2.1 Interactive Execution of MAHA . . . 303
B.2.1.1 MAHA Automatic Operation . . . 306
B.2.2 Command Line Execution of MAHA . . . 309
B.3 MAHA Interactive Output . . . 309
B.4 An Interactive Example . . . 310
B.5 File Formats . . . 320
B.5.1 Dataflow Description File . . . 320
B.5.1.1 Node Description . . . 320
B.5.1.2 Edge Description . . . 321
B.5.2 Module Description File . . . 322
B.5.3 Constraint Description File . . . 323
B.5.4 MAHA Output File . . . 325

Appendix C
PLA Loop Counter Area Evaluation . . . 329
C.1 Product Terms in the PLA for the Register Control Method . . . 332

Appendix D
PASTA Usage: Inputs and Outputs . . . 336
D.1 PASTA Input . . . 336
D.2 PASTA Output . . . 339
D.3 Notes . . . 340
D.4 PASTA Example . . . 341

Appendix E
REG Usage: Inputs and Outputs . . . 343
E.1 REG Input . . . 343
E.2 REG Output . . . 344
E.2.1 Non-pipelined Register Estimation . . . 344

E.2.2 Pipelined Register Estimation 345

Appendix F

MUX Usage: Inputs and Outputs 347
F.1 MUX Input 347
F.2 MUX Output 348

Appendix G

Floating Point Coprocessor Description 351
G.1 Floating Point Coprocessor Value Trace 351
G.2 Floating Point Coprocessor Park Normal Form Description 373

List Of Figures

1.1 Area/Time for Serialization 6
1.2 Tradeoff Analysis Flow Diagram 11

2.1 Example dataflow graph 29
2.2 General Overview of MAHA Algorithm 30
2.3 Example delay operation insertion 34
2.4 MAHA Critical Path Partitions (p_{cp}) 41
2.5 MAHA Critical Path Scheduling and Allocation 42
2.6 Example with Critical Path Partitioned 44
2.7 Demonstration dataflow graph # 2 45
2.8 MAHA Off-Critical Path ASAP Scheduling and Allocation: Part 1 of 2 47
2.9 MAHA Off-Critical Path ASAP Scheduling and Allocation: Part 2 of 2 48
2.10 MAHA Critical Path Stretching 49
2.11 Initialization of MAHA 50
2.12 Demonstration dataflow graph # 1 52
2.13 Overall operation of MAHA: Part 1 of 2 53
2.14 Overall operation of MAHA: Part 2 of 2 54
2.15 Simple loop 58
2.16 Loop with mid-graph exit condition 60
2.17 Loop with mutual exclusion and mid-graph exit 61
2.18 Loop after simple transformation 62
2.19 Disconnection of mutually exclusive branches 63
2.20 Completed loop transformation 63

2.21 Procedure for Transforming Cyclic into Acyclic Dataflow 64
2.22 Loop Transformations 65
2.22 Loop Transformations (cont.) 66
2.23 Demonstration data flow Graph 68
2.24 MAHA Results for 2-stage Design 70
2.25 MAHA Results for 3-stage Design 71
2.26 MAHA Results for Cheapest Design 72
2.27 Sehwa Results for Cheapest Design 75
2.28 Parallel Example 77
2.29 Temperature Controller 78
2.30 Multiplier Example 79
2.31 FIR Filter 79
2.32 AR Lattice Filter 80
2.33 Random Graph 81
2.34 Simple Conditional 82
2.35 Large Conditional 83
2.36 Synthesis Results for Small example 85
2.37 Synthesis Results for Multiplier 86
2.38 Synthesis Results for FIR Filter 87
2.39 Synthesis Results for AR Lattice Filter 88
2.40 Synthesis Results for Random Graph 89
2.41 MAHA design of AR filter 92
2.42 Human design of AR filter 93
2.43 Elliptical Wave Filter 99
2.44 Elliptical Filter Designs of Several Synthesis Systems 100
2.45 Sample Schedule of FIR Filter 103
2.46 Identical Delay/Cost of FIR Filter with Fewer Registers/Muxes . 103

3.1 PLA Finite State Machine 106
3.2 Computer Runtime 108
3.3 Two Example Data Paths 113
3.4 8-state PLA with a register control line shown 114
3.5 AR Lattice Filter 117

3.6 Multiplexer control methods 119
3.7 Different PLA multiplexer control styles 120
3.8 Internal construction of PLA 124
3.9 Loop on Status Flag 126
3.10 Fixed Count Loop 127
3.11 Variable Count Loop 127
3.12 PLA with External Counter and Register Decoding for Loop . . . 131
3.13 Conditional Path Example 135
3.14 Complex Conditional Path Example 137
3.15 Data Path of Averaging Operation 145
3.16 Serialized ADD operation 146
3.17 Complex Conditional Graph ©1986 N. Park 150

4.1 AR lattice filter showing cutset 162
4.2 Example layered network 165
4.3 Example figure highlighting eligible edges 167
4.4 Algorithm for DFG transformation/flow lower bound assignment . 168
4.5 Transformation of dataflow showing lower bound on flow 170
4.6 Transforming conditional graphs for register computation 171
4.7 Criss-cross 173
4.8 Modified criss-cross marked with lower bound on flow 174
4.9 Register values for AR lattice filter 177
4.10 Graph depicting pipelined scheduling 180
4.11 Relation between partitions and latency 183
4.12 Register values for pipelined AR lattice filter 185
4.13 Example dataflow graph showing scheduling 196
4.14 Example dataflow graph showing RTL architecture 198
4.15 Computation of y_i and M_i^u 200
4.16 Pipelined conditional graph 207

5.1 AR Filter showing Operation Groupings 215
5.2 8-bit multiplier implemented using 4-bit multipliers 217
5.3 Example Dataflow Graph 218
5.4 Serial versus Parallel Implementation 221

5.5 Incremental Area vs Serialization Quotient for $A_{bi-op} = 100000$. 226
5.6 Bit-dependent Operator Area versus Maximum Serialization . . . 228
5.7 Best Operator Area versus Serialization (multiplexers and registers) . . . 229
5.8 8-bit Multiply using 4-bit Multiplier . . . 231
5.9 Small Dataflow Graph . . . 232
5.10 Random Graph . . . 234

6.1 Control Path/Data Path Tradeoff Evaluation . . . 239
6.2 Evaluation of Design Space: Prediction and Synthesis . . . 240
6.3 AR Filter Area Comparison . . . 244
6.4 AR Filter Totals . . . 245
6.5 AR filter design (clock cycles = 10) . . . 247
6.6 AR Filter Design Region . . . 249
6.7 Floating Point Coprocessor: Serialize multiply/divide . . . 255
6.8 Floating Point Coprocessor: Serialize add/sub/cmp/neg . . . 257
6.9 Floating Point Coprocessor: Serialize all modules . . . 259
6.10 Predicted Elliptical Filter: Operators Only . . . 262
6.11 Predicted Elliptical Filter: Total Design w/o wiring . . . 263
6.12 Temperature Controller . . . 265
6.13 Module Bitwidth Tradeoffs: Data Path . . . 266
6.14 Module Bitwidth Tradeoffs: Total . . . 267
6.15 Floating Point Coprocessor: Operators . . . 269
6.16 Floating Point Coprocessor: Total . . . 270
6.17 Floating Point Coprocessor: Modified (Datapath) . . . 272
6.18 i8251: Parallel designs . . . 274
6.19 i8251: Serial designs . . . 277

C.1 Minimal Covering of product terms for Counter . . . 330

List Of Tables

1.1 Minimum Sensitivity of PLA Complexity versus Area 9

2.1 MAHA Results for Simple Example 69
2.2 Modules Used for Synthesis 69
2.3 Sehwa Results for Simple Example 73
2.4 Comparison of Synthesis Results: Sehwa 76
2.5 Summary of Register and Multiplexer Area for Cheapest Design . 90
2.6 Module Library of Adders and Multipliers 95
2.7 Module Sets Evaluated by MAHA 96
2.8 Comparison of MAHA results to Human/Random: **Area**[1] 97
2.9 Comparison of MAHA to Human/Random: **Time**[1] 98
2.10 Summary of MAHA Validation 101

3.1 Multiplexer Control Analysis 121
3.2 Validation of PLA model 125
3.3 Unrolling versus External Counter for Loop Implementation . . . 133
3.4 PLA Area Estimation: Basic Model 144
3.5 PLA Area Estimation: Conditionals and Loops 146
3.6 Summary of Results for Pipelined PLA 153
3.7 Comparison of Folded PLA Area 154
3.8 PLA Area Estimation using Data Path Prediction Tools 155

4.1 Comparison of register requirements: non-pipelined designs 172
4.2 Comparison of AR Filter estimated and actual register use 177
4.3 Comparison of pipelined AR Filter estimated and actual register use 184

4.4 Pipelined Predicted Data Values 187
4.5 Comparison of Predicted and Actual Microcycles 188
4.6 Register prediction: Non-pipelined designs 202
4.7 Register prediction: Non-pipelined designs 203
4.8 Register prediction: Pipelined designs 205
4.9 Register prediction: Pipelined designs 206
4.10 Multiplexer estimation: Non-pipelined designs 209
4.11 Multiplexer estimation: Non-pipelined designs 210
4.12 Multiplexer estimation: Pipelined designs 211

5.1 Sensitivity of δ to n and ζ 223
5.2 Minimum and Maximal Change in PLA Parameters versus n . . . 223
5.3 Analysis of Serialization Factor #1 for the Example shown in Figure 5.9 232
5.4 Analysis of Serialization Factor #2 for the Example shown in Figure 5.10 233

6.1 Module Library for AR Example 241
6.2 AR Filter Designs: Predicted 246
6.3 AR Filter Designs: Actual 246
6.4 AR Filter Designs: Error Summary 248
6.5 Non-Pipelined Design Curve Computation Time (sec.) 250
6.6 Module Library 253
6.7 Multiplier/Divider Floating Point Bit-Serialization 254
6.8 Add/Sub/Cmp/Neg Floating Point Bit-Serialization 256
6.9 Combined Floating Point Bit-Serialization 258
6.10 Elliptical Filter Module Sets 260
6.11 Original Elliptical Filter: Predicted 260
6.12 Original Elliptical Filter: Actual 261
6.13 Floating Point Coprocessor with ALU 269
6.14 i8251: Individual designs 272
6.15 i8251: Parallel Design Combinations 273
6.16 i8251: Parallel Design Comparisons 275
6.17 i8251: Serial Design Combinations 276

6.18 i8251: Serial Design Comparisons 276

C.1 Comparison of Internal versus External Decoding of Loop Counter 331
C.2 Summary of Counter Impact on PLA Parameters 332

D.1 Fixed Values in **PASTA** 340

Chapter 1

Introduction

With the lowered cost and broadened functionality of digital systems, an increasing number of primarily non-electronic companies are inserting complex digital chips into their products. In conjunction with this rapid growth, the life cycle of any given digital product is decreasing. Thus, an intense competitive stance is needed to ensure survival in the electronics industry. A company must get to the market first with a new chip; those getting there second may find the market saturated by a competitor or exhausted - the market window was missed.

A consequence of this requirement is that digital systems designers use CAD tools to produce chips which comprise hundreds or thousands of individual predesigned modules. Unfortunately, this is not just a simple matter of manufacturing the first design that is feasible. To produce chips of superior price and performance, a number of candidate chip designs must be analyzed. Exploration of this design space in digital systems can be very expensive in computer run-time. Further, there are portions of the design space which are not even considered by current CAD programs; thus, it is the intuitive judgment of the human designer – a good designer – that navigates the design through these areas.

One important aspect of design, partitioning a behavior at the system level into control and data path structures, is currently performed solely by human designers. Expert designers understand the tradeoffs between these two structures, but have no tools for locating good partitions. Even experienced designers may choose inferior designs due to the enormity of the design space and the interaction between the data path and control path structures.

The research presented here is directed towards resolving the problem of applying control path/data path tradeoffs and evaluating design decisions against a set of global constraints. Both experienced and inexperienced system designers will be capable of realizing superior designs; in addition, expert designers have a technique for evaluating design changes at the system level. A methodology for exploring the design space to locate superior designs quickly using these tradeoffs and evaluation tools will be described. The remainder of this chapter details the problem and the approach taken.

1.1 Problem Statement

In the design of large electronic systems, particularly those which utilize a central processor, partitioning to separate the hardware, firmware, and software subsystems is usually based upon the experience of human designers. Given a human-dominated design environment, high-level partitioning into these subspaces is economical and necessary to completing a design in a timely manner; this separation allows each to be developed fairly independent of the others. The quality of the initial partitioning drives the final result since design options are reduced at an early stage. Tradeoffs between the control path and data path functionality performed early in the design process to produce the structure, henceforth referred to as *control path/data path tradeoffs*, are crucial to producing good designs.

When performed at an early stage, control path/data path tradeoffs allow major decisions to be evaluated at far less computing cost than at a later stage. For example, the decision whether to develop a pipelined design can be performed at the system level, but may be impossible to change once actual synthesis has started at a lower (RT (*Register-Transfer*) or circuit) level. Even if such major backtracking is feasible, a substantial part of the design would have to be discarded with a comparable increase in design time and potentially wasted resources.

The research suggested here is intended to fill a gap in the design process and subsequently improving the RT-level design produced. To be viable, the method used should locate the *globally* superior design region quickly without

resorting to exhaustive search. Current design tools are inadequate as they do not explore all regions of the design space even if considerable computing resources are expended. For example, in the AR Lattice Filter shown later, there are 16 multiply and 12 addition operations. Using a small module library of only three elements (multiplier, adder, and multiply-adder pair), there are 256 possible structures at the system level (not achievable using current synthesis tools) resulting in *over* 3000 feasible RT-level designs. The enormous detail present at the RT-level spawns tradeoffs of a local nature since global evaluation is too costly. Clearly, a set of tools for eliminating inferior structures at the start would enhance the design process by dramatically reducing the number of RT-designs to fully construct and evaluate later.

One approach for exploring designs at the system level is transforming the behavior to produce different structural representations. An early effort which discusses global transformations is found in Snow's dissertation [Sno78]. Snow showed how analysis of behavior provides an implementation-independent environment for performing some control path/data path tradeoffs. A collection of translation templates which operated upon the data path were derived from the experience of optimizing compilers. As a practical demonstration, some of the transformations were applied to two subsets of the PDP-11 processor to show the resultant improvement.

There are also many other types of transformations which operate upon dataflow graphs and alter their internal behavior. Tree height reduction [Tri85] and loop winding [Gir87] are two examples. Other compiler optimizations such as strength reduction and loop unwinding are also candidates [AUS79].

A typical synthesis engine produces alternative implementations from the same behavior. For example, some engines provide a range of designs from the most parallel to the most serial, which changes the area of the controller and data path hardware. A decision whether to use a pipelined or non-pipelined approach is another high-level tradeoff.

To more closely describe the effort, the nature of control path/data path tradeoffs as they are treated in this research will be defined. At the system level, a design can be viewed structurally as a control path operating data path

hardware including routing and storage. Design information present at an early stage which affect system-level tradeoffs includes

- the behavioral description,
- user goals and constraints, and
- available (hardware) modules.

A *behavioral description* is an abstract view of the system in terms of operations and the values produced and consumed; this pure form lacks any structural or physical attributes. In addition, there are a number of different gauges by which to measure the quality of a design, among which are

- circuit operation time,
- circuit area,
- pin count,
- test coverage, and
- power consumption.

Describing the interaction *within* the control path and the data path as well as *between* the control and data paths is fundamental to this research, with the goal to locate superior implementations. Emphasis is placed upon characterization of this separating line between the two design subspaces. Since this "line" is a direct consequence of individual data path and control path architectures which are governed by system-level constraints, an understanding of each subspace is also important. Tools which separately address data path and control path design will be necessary to evaluate system-level partitioning; such tools become an integral part of the research.

1.2 Motivation

An objective of the research is to produce techniques which accept a behavioral description, perform user tradeoffs at a high level, evaluate those tradeoffs with

respect to global constraints, and produce a system-level partitioning between the control path and data path which results in better designs. The boundary between the control path and data path is varied through the use of tradeoff *templates* to locate superior designs. (A superior design is one whose structure achieves or surpasses the goals while meeting constraints imposed by the design specification.)

The foremost design parameters considered in this thesis are *time* and *area*. Note that other parameters may be chosen such as power consumption or testability when performing tradeoffs; this thesis does not address them. A designer either strives to minimize time with respect to an area constraint or minimize area with respect to a time constraint. Based upon these two primary factors, items related to circuit design are

- operator area and time,
- routing logic and storage area and delay,
- controller area and time, and
- wiring area and delay.

As an example, a given behavior can usually be implemented by a large amount of hardware which operates quickly or a small amount of hardware that must be shared serially to complete the task. Effects of this "serialization" upon the control path and data path area are shown in Figure 1.1. The area of the data path drops with increasing serialization. However, since the controller has more states as well as increased routing and storage hardware, its area grows. The total design area, which is the sum of area for the data and control path, reaches a "boundary point" in the area/time curve where the area is a minimum. Designs greater in time than the "boundary line" at the point x_0 are also larger and thus *inferior* to solutions lower or equal in time to the boundary point.

Figure 1.1 clearly shows that control area can not be ignored, although its effect is governed by the size and complexity of the data path. The figure also shows that there is a relation between the amount of serialization and both the control path and data path area. Applying control path/data path tradeoffs

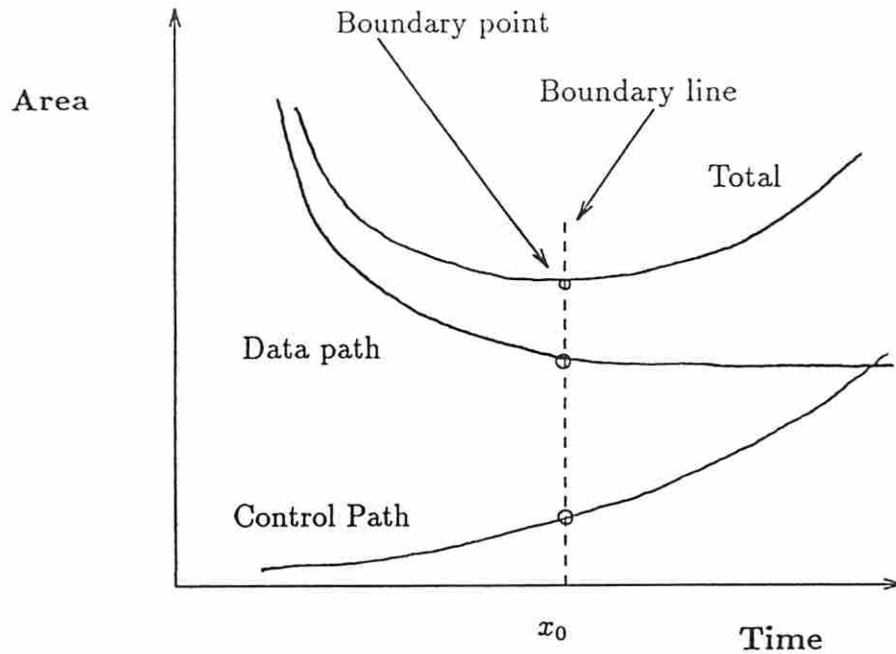


Figure 1.1: Area/Time for Serialization

relies on the development and verification of accurate control path and data path estimation models.

1.3 Problem Approach

The focus of this research is upon describing control path/data path tradeoffs for partitioning a circuit description and producing a methodology for evaluating these tradeoffs. Starting with a behavioral description in the form of a dataflow graph and a set of constraints, a circuit structure has been found which is capable of producing the best designs. Steps in this structure's construction include the data path, routing and storage, and the control path.

1.3.1 Data Path Analysis

There are many aspects of data path synthesis which affect circuit area and time. Those related to system-level tradeoffs are readily defined. One task of data path synthesis is partitioning of a behavioral description (usually in the form of a dataflow graph) into timesteps. This offers a tradeoff between operator

sharing which reduces the cost of the data path and the larger execution time which results.

Second, the type of data path synthesis, or design style, has two major categories which are specified at the system level: pipelined and non-pipelined design. Both design styles will be considered, although a larger focus will be upon non-pipelined designs.

Module selection forms a third category. There are numerous issues among which are ranking of acceptable modules (area and time), generation of modules, bitwidth considerations, and influence of shared operators on the selection process.

Finally, module allocation is distinct from module selection. After module selection, the type of module bound to a given operation has been chosen, but not the quantity of modules or sharing of the specific hardware with other operations. Wiring and routing/storage delays are two often ignored factors in module allocation. In this research, system partitioning will not consider second-order aspects of module sharing such as fan-in/fan-out (other than multiplexer allocation) or placement proximity; module speed, area, and sharing will be considered.

Module selection, bit-serial operators, and timestep partitioning will be addressed in this thesis. Although an important factor, design style selection is beyond the scope of this thesis.

1.3.2 Routing and Storage Analysis

Signal routing and storage requirements must be considered to evaluate the effects of applying control path/data path tradeoffs. Signal routing is accomplished through buses or multiplexers; storage is performed by registers. The use of multiplexers, buses, and registers can play a significant role in determining the quality of a design. Since area is the dominant contribution of routing and storage hardware to a design, analysis will concentrate on this parameter. Time delay is of lower priority since its effect is essentially uniform over a wide range of designs (i.e. a fixed contribution to each time step).

The effect of a tradeoff upon wiring area depends upon the transformation and the circuit size. Module selection may have some effect upon wiring between modules, and impacts internal module wiring. However, for typical standard cell and random logic designs, local changes have small impact on the global wiring area. Since a general model for standard cell wiring area exists [KP86], one can assess the overall impact of system partitioning upon wiring area.

1.3.3 Control Path Analysis

Contrary to data path synthesis, little has been written about control path synthesis in recent years. This is primarily due to the regular structures in control synthesis (both PLAs and microcode) which have been heavily researched in the past; the problem is considered solved. Current control path literature concentrates upon either minimizing the product terms in PLAs or construction of microcoded and nanocoded controllers. At present, PLAs are the predominant method for constructing controllers.

Automated synthesis tools for PLAs have been in widespread use for several years. However, a theoretical analysis of the area consumption of PLA state machines has yet to be published. Basic analysis will consider the following:

- describing the PLA area with respect to its parameters (inputs, outputs, and product-terms),
- theoretically bounding and/or estimating the number of product-terms,
- state machine size effects,
- register and multiplexer operation effects,
- conditional path effects, and
- fixed and variable loop effects.

The PLA area is a direct consequence of the data path requirements. Hence, predicting the PLA area with the data path parameters is one important goal of the research. State machine size as well as register and multiplexer control influence PLA area; these effects will also be modeled. Conditional paths and

Table 1.1: Minimum Sensitivity of PLA Complexity versus Area

States	Area (mil^2)	Impact of Increasing Term by One: Output Term Δ %	Output Term Δ Area	P-term Δ %	P-term Δ Area
2	11.8	10.0	1.2	6.1	0.7
5	27.8	5.5	1.5	5.9	1.6
10	51.0	4.1	2.1	5.0	2.6
20	121.8	2.7	3.2	3.4	4.1
50	459.4	1.5	6.7	1.7	7.8
75	932.3	1.0	9.5	1.1	10.3
100	1518.4	0.8	12.4	0.9	13.1
150	3158.7	0.6	18.1	0.6	19.0
200	5345.9	0.4	23.8	0.5	24.7
300	11576.5	0.3	35.3	0.3	36.4

loops introduce complex but necessary extensions. Finally, the model will be extended for folding and data path design style considerations.

One difficulty with a centralized PLA controller is that its size grows until it has the dominant area in a design. Furthermore, a point is reached where a small increase in control operation results in an enormous impact on the PLA area.

Expanding control using an existing PLA would entail increasing the number of output lines and/or the product terms and possibly the input lines. In the *best* case, just one of these parameters would be increased. Table 1.1 reveals the effect of increasing a term by **one** as the 2 μm CMOS PLA size increases.

Although the increased output and product-term area effects as a percentage of the total shrink with increasing PLA size, raw area rises at a much higher rate. For example, increasing a state machine from 200 to 201 states consumes nearly as much area as *doubling* the states of a 5-state PLA or building a *completely separate* 4-state PLA controller. Furthermore, the layout may ultimately become too large to use efficiently since either the power bus must be enlarged or a longer clock cycle time must be accepted or both. Large PLAs also provide less flexibility for the floorplan when combined with the data path on a single chip. These results suggest that partitioned or distributed controllers might be useful

in large and/or complicated designs. Indeed, this approach has been used in the Motorola 680x0 and Intel 80x86 line of microprocessors.

In data path synthesis, large blocks can be partitioned into smaller functional blocks to produce a more efficient layout. This concept will also be applied to the PLA controller for specific operations such as register and multiplexer control as well as loop constructs; some aspects of the control will be handled by external hardware to reduce the overall area and keep the control blocks manageable. Large PLAs are often much slower than the hardware they operate. Thus, breaking the controller into smaller functional blocks may also prove useful in improving the speed of the controller.

In summary, there are two observations about the PLA area. First, any additional functionality would, at minimum, impact the number of outputs or product-terms. Additional terms cause the PLA to grow in area at a non-linear rate. Second, and more important for system-level partitioning, this relation reveals that neglecting control costs when partitioning a given dataflow graph into a smaller timesteps (resulting in potentially less data path hardware but more states) may not produce the best designs.

1.3.4 System-Level Partitioning

System-level partitioning, as defined in this thesis, is strictly data path/control path partitioning. Tradeoffs are performed to explore a larger portion of the design space and realize implementations that are unobtainable through a direct interpretation of the behavior. Thus, a design which better meets the user objectives may be achieved.

In this thesis two global methods are explored for performing system-level partitioning: analytical and empirical. Despite the lure of a purely analytical approach for obtaining an optimal solution, a complete and accurate analysis is exceedingly difficult, although some progress is reported here on some data path/control path tradeoffs. By contrast, the empirical approach proves not only to be adaptable to any transform type, but also can be controlled intelligently to yield results rapidly. A flow diagram of this second method is shown in Figure 1.2.

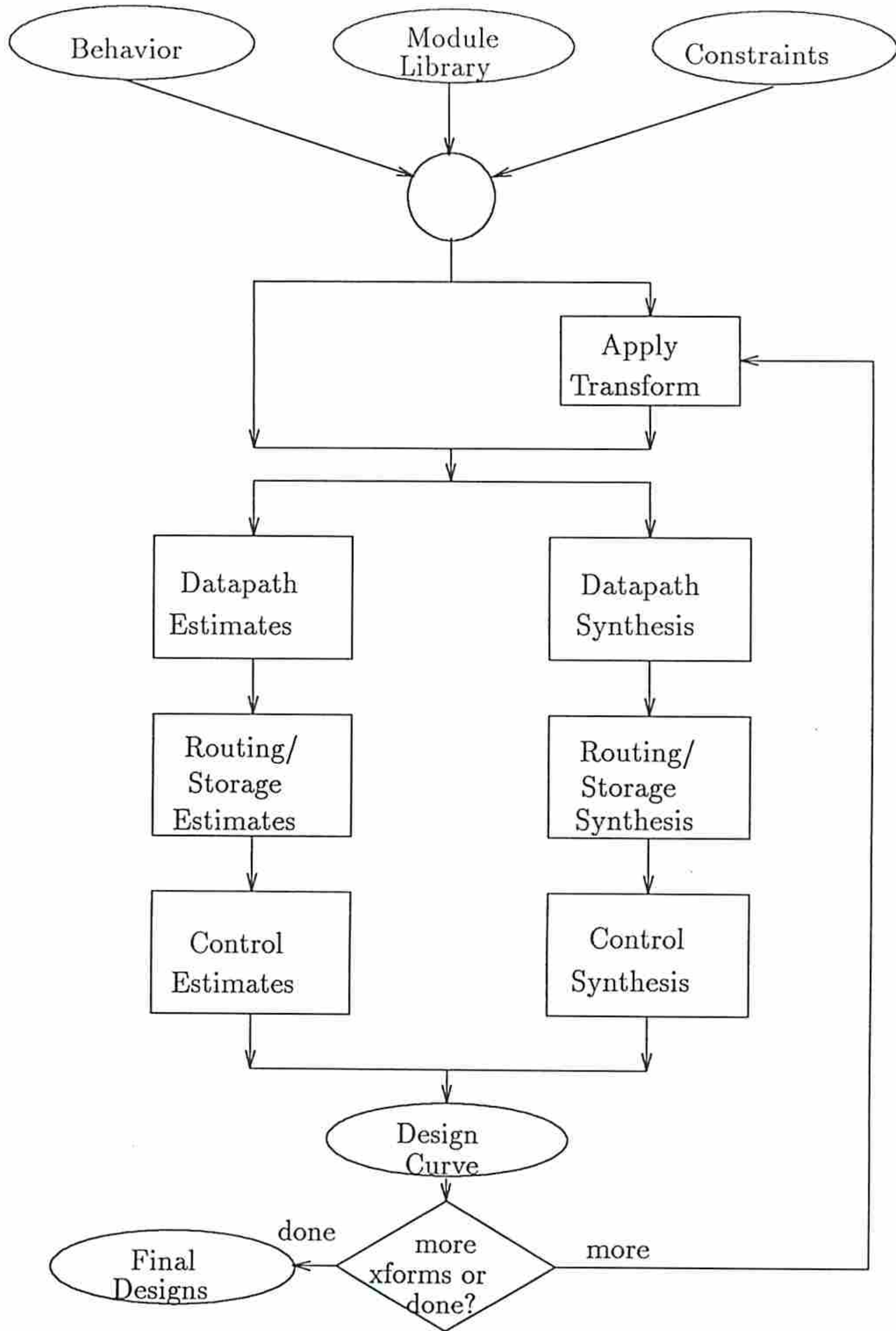


Figure 1.2: Tradeoff Analysis Flow Diagram

The tradeoff procedure relies on both the presence of quick estimators and also a complete set of synthesis tools. Estimators are used to bound an area/time region in the design space which encompasses the design objectives. Should the estimates appear promising, synthesis tools are targeted to produce designs in this smaller region. Therefore, it should be possible to perform a number of tradeoffs in a timely manner.

There are many types of tradeoffs which affect the ultimate performance of a circuit. Classes of tradeoffs having the most impact on the overall design are comprised of

- bitwidth tradeoffs,
- decomposition tradeoffs,
- composition tradeoffs, and
- distributed control.

For bitwidth and decomposition tradeoffs, such as bit serial operators or use of shift-add circuitry to perform a multiply, increased serialization or simplification of the data path may adversely affect the number of control states and, hence, the control area. Composition tradeoffs such as ALU substitution increase the potential for sharing data path hardware at the expense of additional ALU control lines. Finally, distributed control divides a centralized controller into smaller slave controllers which may reduce the total control and wiring area as well as offering independent synchronous operation. This is offset by a data path area increase since sharing to the same degree might be no longer possible.

System-level tradeoffs allow the designer to explore design alternatives which enable a structural decision to be made quickly and correctly. However, such tradeoffs require an accurate model of both the data path and control path area. Currently, there is a model available for predicting data path area, including wiring area, which will be utilized [KP84]. However, an area estimation model for PLA controllers did not exist when this research began. Thus, one was developed. In addition, the predictors lack register and multiplexer estimation; these models were also formulated.

Another parameter to consider in evaluating tradeoffs is data path and control path delay. If it is assumed that a controller is faster than the hardware it is operating, which is usually the case for small designs, then this problem simply reduces to bitwidth effects on data path delay. For larger controllers, control delay may be a factor. However, such work is beyond the scope of this thesis, and has since been addressed by others within the ADAM group.

1.4 Evaluation Tools within ADAM

To evaluate tradeoffs, utilities for both synthesizing and estimating a complete design from a behavioral description must be available. The different parts of a design such as the control path, routing/storage and data path are constructed using different and, unfortunately, unrelated tools; these must be completely integrated into a single coherent package to accomplish system-level tradeoffs.

1.4.1 SLIMOS: Module Set Selection

Prior to estimation or synthesis, module styles which will implement the various functions in the dataflow graph are selected. The module styles chosen from a library of candidates are driven by the behavior (dataflow graph) and the constraints. **SLIMOS** selects modules for pipelined design using a theoretical area-time lower-bound model [JPP88].

Although **SLIMOS** is targeted for pipelined designs, it has also been able so far to construct the best module sets for non-pipelined designs. A theoretical model for module set selection of non-pipelined designs has yet to be published.

1.4.2 MAHA: Non-Pipelined Data Path Synthesis

Data path synthesis is concerned with timestep partitioning and allocation of pre-defined hardware modules to a dataflow graph. Typically, these functions are constrained by cost at one boundary and speed at the other. Early in the design process, it is desirable to explore the possible *global* design space, using the dataflow graph and fixed module styles, to find the superior solution space.

MAHA was written to provide the designer (automated or human) with scheduling and allocation solutions given a dataflow graph, a module library, and area and/or time constraints [PPM86]. An initial linear module assignment and scheduling of critical path operations is followed by a cost-based assignment using the concept of *freedom*. *Freedom* is a measure of the time range during which operations (off the critical-path) can be scheduled consistent with the timing constraint. Operations with the least *freedom* (tightest fit) are scheduled first. The program either minimizes speed subject to a area constraint, minimizes area subject to a time constraint, or provides a set of feasible solutions.

Extensions to **MAHA** include resource-sensitive scheduling, register allocation (worst case), and multiplexer allocation to produce a more accurate area estimate. In addition, delay operation insertion was added to widen the design search space, and local timing constraints can be entered.

1.4.3 Sehwa: Pipelined Data Path Synthesis

Sehwa is a synthesis program for producing pipelined designs which uses the same inputs as **MAHA**. **Sehwa** performs functional pipelining, which means that there is no physical stage which corresponds to a logical grouping of operations. The program handles conditional branches and considers the effects of register assignment; it also alters its schedule dependent upon the degree of resynchronization expected in the design. The underlying theory and implementation of **Sehwa** is given in [PP88].

1.4.4 HAPPE: Data Path Estimation

HAPPE (High level Area Performance Prediction and Estimation) consists of two tools which predict the area and delay of the data path for completed designs given the module set and dataflow graph information. **PSADNP** is used to obtain non-pipelined estimates; **PSADP** is used for pipelined estimates. The underlying theories of the pipelined area/time prediction [JPP87] and non-pipelined area/time prediction [JMP88] are given elsewhere.

1.4.5 MABAL: Multiplexer and Bus Allocation

MABAL allocates the routing and storage hardware for either a pipelined or non-pipelined design [KP90]. This program uses a cost-driven heuristic to allocate multiplexers and/or buses for interconnecting operators and registers. **MABAL** recognizes operator commutivity which allows it to reduce the size of multiplexer trees or produce smaller buses.

Starting with schedule produced by a synthesis program, **MABAL** performs register, bus, and multiplexer assignment. (An algorithm which can produce optimal register assignment in polynomial time was incorporated into **MABAL** [KP87].) The cost (area) of sharing registers and operators is compared against the actual routing hardware area and a projection of the wiring area; the cheapest approach is always taken. The output of **MABAL** is an RT-level design which could be used as an input for a logic/layout synthesis tool.

1.4.6 Berkeley PLA Synthesis Package

There are many methods for building a controller, of which random logic, timing generators, microcode, and PLAs are the most popular. For larger designs, microcode and PLAs are favored due to their regular structure and availability of design tools to implement them.

Currently, PLAs are increasing in popularity among designers. Since more complex VLSI chips are currently being produced, there is often more than one controller to allow multiple tasks (such as memory management and prefetch) on the same chip. Smaller distributed controllers favor PLA implementation over microcontrollers when cost is to be minimized. Consequently, the thesis will focus upon controllers implemented by PLAs, although the area and time tradeoffs for microcode are expected to be similar.

The Berkeley CAD Toolbox has a number of programs to assist in the design of a PLA. At the top level is **PEG** (PLA Equation Generator) which accepts a high-level state machine description as input and produces a set of state equations useful to other tools to implement a controller [Ham83]. The high-level state machine grammar accepted by **PEG** is sufficient to describe any synchronous controller; programs created are isomorphic to Moore state diagrams.

EQNTOTT is a program which accepts state equations and derives the sum-of-product terms used in PLA design. By piping the output of **PEG** into **EQNTOTT**, the PLA "personality matrix" is generated. To produce a heuristically optimized PLA (minimum number of product-terms), the output of **EQNTOTT** is passed through **ESPRESSO**. Ultimately, **MKPLA** accepts this personality matrix and produces a layout of the PLA. A tool was written for this thesis research which extracts the area from the PLA layout for validation of the PLA models.

1.4.7 PASTA: PLA Control Area Estimator

Rather than synthesizing a PLA, it is more useful to obtain a quick area estimate when a number of designs need to be explored rapidly. The Berkeley tools have proved to be adequate for small examples, but slow down at an exponential rate with the number of controller steps. A survey of the literature did not reveal any such estimation tool, so it was constructed as part of this thesis.

Control area estimation specific to PLA state machines has been extensively analyzed here. Several parameters which determine the size of an unfolded PLA have been identified including

- timestep partitions (states),
- register control,
- (conditional) test inputs,
- conditional branches,
- multiplexer control signals, and
- control loops (fixed and variable).

An algorithmic model called PASTA (Pla Area eSTimator Algorithm) which determines the area of a general PLA state-machine controller is discussed in Chapter 3. This model encompasses both pipelined and non-pipelined design and has been extended for prediction of folded PLAs.

1.4.8 PLEST Data Path Area Estimator

PLEST estimates the area of a standard cell layout from the number of cells (modules), number of nets, and average wire length [KP86] [Kur87]. Although use of **PLEST** allows the designer to consider another aspect of design which contributes area, there is an issue of design time versus utility. During the initial system-level evaluations where a wide variety of designs are attempted (which is likely to result in wide area changes), a precise wiring estimate may not be necessary. Behavioral descriptions on the order of 100 operators take 30% longer for data path analysis when **PLEST** is used versus the sole use of **MAHA**. Hence, execution of **PLEST** may only be warranted for discrimination between designs having similar areas.

1.5 Related Work

In this section, a survey of research related to control path/data path tradeoffs is presented. Extensive search has uncovered no references in the specific area of interest; however, there are several published works in the area of hardware/firmware/software tradeoffs that are closely related and will be reviewed. In addition, since the foundation of the proposed research relies on robust tools for data path and control path analysis, an overview of these areas is included first.

1.5.1 Data Path Synthesis

Over the past decade, there has been broad emphasis on automated design of data path hardware, more aptly described as data path synthesis. (This is also referred to as high-level synthesis.) Data path synthesis starts with a behavioral description in the form of a dataflow graph to produce the data path hardware in two steps: event scheduling and resource allocation. Event scheduling partitions the dataflow graph into one or more clock cycles of some fixed length (for non-pipelined synthesis). Resource allocation assigns hardware modules to operators with an emphasis upon resource sharing; the amount of sharing depends upon the particular design and the event scheduling.

Since optimally synthesizing data path hardware based upon a set of constraints is thought to be an NP-complete problem [KT83, Tho86], a variety of approaches have been published which achieve good designs in a reasonable amount of time. These range from a purely heuristic approach for allocation such as **EMUCS** [McF81] to a formal approach for scheduling and allocation which uses mixed integer-linear programming [HP83]. Purely heuristic approaches suffer from being overly rigid. Although they typically perform the synthesis task quickly, the resulting designs can be far from optimal due to some aspect of the design that was missed. For even the small examples cited in literature, a human designer could do better.

At the other end of data path synthesis is a rigorous formal definition which can be used to produce optimal designs. Early work by the author was directed at describing and synthesizing register-transfer (RT) designs [PKM84]. Peripheral research related to Hafer's formal approach revealed that even for small problems, producing optimal designs can be expensive [HP83, HP81]. Optimal data path synthesis involved solving a mixed integer-linear programming problem where the matrix size grows roughly at $O(n^3)$ where n is the number of operations in the data path. (According to Hafer [HP83], the number of relations grows at $O(hn^2)$ where h is the number of hardware elements. In the case where every operation gets a hardware element, $h = n$.) Clearly, there is a limit on the size of designs which can be performed using this method. An overview of other data path synthesis programs is published elsewhere [PH87].

1.5.2 Control Path Synthesis

In comparison to data path synthesis, research in control path synthesis is limited. Control path design is a less difficult problem since its characteristics depend closely upon the data path configuration. Furthermore, controllers are generally either regular structures such as PLAs or standard architectures such as a microengine, which further reduces the design complexity.

During early research efforts, primary emphasis was placed upon automatic generation of microsequence controllers for data path hardware [Nag80, NP81, NCP82]. In Nagle's research, a control graph is formed by analyzing an ordered

acyclic dataflow graph which has been preallocated (hardware assigned to each operation node), but without any specific timeslot specified. A collection of routines reduce the control graph to a minimal (not necessarily optimal) microprogram through two methods: autonomous reduction and attraction weights.

Autonomous reduction evaluates the current control graph and selects the node with the highest weight (maximal state overlap) to be assigned a separate field in the microcode; the iterative process terminates when the microcode word width increases beyond some specified limit. This task is accomplished in linear time with respect to the control graph. Attraction weights are used to determine sharing in $O(n^2)$ time; control nodes which can be executed in the same partition are grouped during successive iterations until no further sharing is possible.

A more general approach to controller design appeared in a thesis by Baldwin [Bal84]. An expansion of the concepts discussed in Leive's thesis [Lei81] pertaining to automated logic synthesis and module selection is the basis of Baldwin's dissertation. Baldwin formulated a controller design language to describe the operation of a circuit which is subsequently transformed into a control structure using a rule-based system. Ultimately, the controller is synthesized. Problems specific to controllers such as loops and conditional branches were discussed and implemented in his system.

Although Baldwin's transforms applied to the control structure are generally useful, experimental results using TTL components were marginal. A variety of synthesized designs were compared against a small number of human-produced controllers using the same components. Even the most inexperienced of the human designers generated a circuit far better than that synthesized by Baldwin's heuristics. Synthesis of a PLA or microcode controller is difficult using this approach since it relies upon a library of control components which can be allocated to the control graph in much the same manner as data path synthesis. This method is not viable for the research presented here.

More recent control path synthesis concentrates upon PLA design. To date, all of the known available (non-commercial) tools in this area are derivatives of the Berkeley CAD system for PLA design [Ham83]. Since a large part of the preliminary research pertains to PLA area estimation, which was validated using

the Berkeley synthesis tools, further evaluation of the Berkeley tools is presented in Chapter 3.

1.5.3 Synthesis considering both control and data path analysis

At a higher level in the design process, interactions between the control path and data path are considered. An early paper that addressed the problem of control path/data path tradeoffs is the Architecture and Design Assessment System (ADAS) for digital signal processors [FSC84]. Using a directed graph to describe the behavior, ADAS allows the designer to evaluate performance and cost of a design using a top-down approach. A high-level design is first partitioned into major functional blocks by the designer who can verify (using Petri Net simulators) whether the graph appears to be performing the desired function properly. Nodes in the graph are successively decomposed by the designer and performance/cost analyzed through design tools. These tools provide the user with information about the current design at each step; however, it is the designer who decides what action should be undertaken to improve the result.

The impact of design changes upon the control path and data path is assessed in the ADAS module binding package. Initially, a set of hardware resources are explicitly mapped onto data path operations by the designer; these can be rearranged to improve sharing while determining the impact on the controller. Conversely, the control path architecture could be modified to analyze alternative data path designs to reflect the varied hardware allocation. Both steps are performed so as to minimize cost without exceeding the original constraints.

Although ADAS utilizes algorithms and heuristics to assist in the design process, a human operator must make all of the decisions. Design decisions are still required in the system described here. However, such interaction can be confined to the top level of the design with lower level predictors and synthesis tools performing the lower level tasks automatically. Furthermore, the set of transformations described here could be automated in some future system to produce superior combined control path/data path designs.

In Marshall's PhD dissertation [Mar86], a more automated system is presented. Marshall constructed a system which produces a board level design from a high-level description. Starting with a system description in OCCAM [Ltd85], the OCCAM intermediate code generated is used for simulation and synthesis. Marshall focused upon a "microcontroller" implementation where a predefined set of I/O modules is available to be attached to a Z-80 microprocessor with some RAM and ROM. These I/O modules are allocated as needed and software is generated for the Z-80 to handle the additional hardware.

Marshall's work is unique in synthesizing both hardware and software at the board level. Some software is part of a pre-defined executive for communicating with I/O devices; however, the overall system (as in his voltmeter and cash register examples) is custom generated by the synthesis engine. Although Marshall requires a minimal transputer as the structural base, in order to achieve better designs synthesis, he will eventually have to consider both hardware and firmware (CPU with software) analysis with no predefined and restrictive macro-structures.

The large scope of combining data path and control path synthesis becomes apparent in Casavant's dissertation on design automation [Cas85]. Casavant presents a working system which synthesizes both control path and data path elements from a behavioral description. Initially, functional units are assigned to the behavior which are then scheduled into microinstructions. After scheduling is complete, registers are assigned heuristically to minimize area. Microcode generation and multiplexer allocation form the last steps of the process.

Although Casavant has produced an impressive amount of synthesis tools, the scope of each is limited to a small set of user-controlled algorithms. In fact, all actions performed by the system are done under user control; no attempt was made to find a good solution automatically. In addition, the initial scheduling of the system is fixed and cannot be modified; backtracking is not directly supported. Resource changes, which are dictated solely by the user, will invoke a new schedule, hardware/register assignment, and microcode design. Minimizing time subject to a maximum cost (area) constraint is the goal of all of his synthesis tools.

Casavant's work is well-suited to an interactive workstation as major decisions such as hardware availability, register usage, bus resources, and microcode word sizes must be provided by the user. Clearly, an experienced designer is necessary to achieve the full potential of the tools, or to achieve good designs at all. An automated design manager would be a valuable and useful extension of this work.

Another recent work which focuses upon high-level tradeoffs between the control and data path is the CAMAD system [Pen86]. A behavioral description is transformed into a structural representation through synthesis of both the control and data parts. The standard data flow digraph is used for data representation whereas an extended timed Petri Net model is used for control. In the initial construction, each node in the data path has a corresponding node in the control graph to which it is linked.

For the actual control path/data path tradeoff, Peng treats the two parts as a single graph which include both C-edges (control) and D-edges (data). Edge "costs" are assigned based upon frequency of execution (from simulation or analysis) with arbitrary weighting (W_d and W_c). Graph partitioning (Kernighan and Lin) is performed until sufficiently small modules are realized. Circuit area and performance are directly influenced by varying the weights, W_d and W_c.

Peng's initial work appears very promising in that it combines both control and data path parts into a single model. However, this major salient feature is also its weakness as even small designs are very complex to represent. Optimizations on the data path are performed simultaneously with the control path, making design changes more expensive computationally and potentially introducing "design oscillations". Interestingly, Peng's partitioning of both the data path and control path results in numerous localized controllers. It was not determined in his research whether widely distributed control is more cost effective than fewer and more centralized PLAs or microcode controllers.

1.5.4 High-level hardware/firmware/software partitioning

A thorough search of the literature reveals sparse research in the area of system-level partitioning. One aspect that is closely related is hardware/firmware tradeoffs. Early analysis of hardware/firmware/software tradeoffs focused upon computer design [BD71, Man72]. Both Barsamian and Mandell discussed hardware/firmware/software tradeoffs as related to processor and system design. In particular, analysis centered on (at that time) a fairly new method called microprogramming. The authors saw the microprogramable processor as a vehicle to increase hardware complexity. (The limitation of hardware complexity was due to the only type of controller then available for small processors: random logic.) Increased hardware capability reduces external software requirements. Although the specific topic is now dated, some of the concepts discussed are not.

Mandell argues that the "outward trade" of functions (distributed control) is expensive but may be worth the increase in cost. Also, the amount of hardware required should be adjusted to achieve a specific performance goal which initially might appear infeasible. In essence, a tradeoff between hardware and firmware/software is performed so that constraints of cost and speed are met. Barsamian states that machines need not be designed by "the faster, the better" rule. Rather, tradeoffs can allow the use of slower hardware if the central processor can execute more complex operations locally. A computer model (MAMO) which considers usage of major components (CPU, memory, I/O, etc.) is presented; simulation of a sorter revealed where the best cost/performance improvements could be performed. All of these concepts have an analogy in VLSI chip design and, in particular, the research suggested here.

Another idea which has surfaced more recently is evaluating the frequency of execution. Hennessy claims hardware/software tradeoffs in processor design should consider the typical execution of a program [H$^+$82]. Among his many points, he argues that general CPU operations should *not* set (a group of) condition codes but rather a specific compare instruction should set a single flag. He cites a vast amount of actual program execution which shows the infrequent use of condition codes without first using a compare instruction. Although his

arguments are directed towards RISC architecture, expected execution is an important tool for chip design. For example, if a conditional which involves a divide operation is used infrequently *even* if it is located on the critical path (the path with the longest delay), the "average" performance is affected little by the actual implementation (whether in hardware or software). In the search for good designs, one can target parts of the behavioral description which offer the greatest potential improvement globally and result in more efficient use of computer resources.

1.6 Thesis Outline

Since many of the tools needed to perform control path/data path tradeoffs did not exist prior to this research, an extensive part of the research is tool development. A description of **MAHA**, the non-pipelined synthesis program, is presented in Chapter 2. Scheduling, allocation, and stretching the dataflow graph (serializing the design) are detailed. In addition, transformation of a behavioral description containing loops into an acyclic form is also described.

A model for estimating PLA control area (**PASTA**) is included in Chapter 3. Two methods for producing data path controllers using PLAs are formulated and extended for loops and conditionals. Control of pipelined architecture is also addressed. Finally, the model is broadened to encompass estimation of folded PLAs.

Estimation of routing and storage requirements is given in Chapter 4. Prediction of register usage relies upon a modified flow analysis of the behavioral graph. Results from the register estimates and output from the data path prediction tools are used by the multiplexer estimator. A statistical model for multiplexer usage is derived and verified.

In Chapter 5, control path/data path tradeoff types are described. Using the control model described in Chapter 3, an analytical model is derived for one type of control path/data path tradeoff and validated.

A more general tradeoff analysis method, which uses the models and heuristics of earlier chapters, is presented in Chapter 6. This method is applied to

numerous designs to illustrate a variety of tradeoffs against a set of user constraints. Large examples include a floating point coprocessor and the i8251 UART.

Finally, Chapter 7 concludes with a summary and discussion of future research topics.

Chapter 2

Extensions to Data Path Synthesis

2.1 Introduction

Over the past decade, there has been broad emphasis on automated design of data path hardware, also known as data path synthesis or high-level synthesis. Data path synthesis starts with a behavioral description in the form of a dataflow graph to produce data path hardware with three tasks: event scheduling, resource allocation, and module binding. Event scheduling partitions the dataflow graph into one or more clock cycles of some fixed length (for non-pipelined synthesis). Resource allocation assigns hardware modules to operators and also allocates data steering (multiplexers and/or buses) and storage modules (registers) while maximizing resource sharing; the amount of sharing depends upon the particular design, event scheduling, and module bindings to the graph operations.

Since optimally synthesizing data path hardware to meet a set of user-defined constraints is recognized as an NP-complete problem [KT83, Tho86], a variety of approaches have been published which achieve good designs in a reasonable amount of time. These range from a purely heuristic approach for allocation such as **EMUCS** [McF81] to a formal approach for scheduling and allocation which uses mixed integer-linear programming [HP83]. Purely heuristic approaches suffer from being overly rigid. Although they typically perform the synthesis task quickly, the resulting designs can be far from optimal due to some aspect of the design that was missed. For even the small examples cited in literature, a human designer could do better.

At the other end of data path synthesis is a rigorous formal definition which can be used to produce optimal designs. Early work by the author was directed at describing and synthesizing Register-Transfer (RT) designs [PKM84]. Peripheral research related to Hafer's formal approach revealed that even for small problems, producing optimal designs can be expensive [HP83, HP81]. Data path synthesis included solving a linear programming problem where the matrix size grew nearly at $O(n^3)$ where n is the number of operations in the data path. Clearly, there is a limit to the size of designs which can be performed using this method.

The work described in this paper is similar in intent to the **EMUCS** algorithm designed by McFarland, but actually has its roots in a control synthesis algorithm developed by Nagle [NCP82]. Nagle's notion of **freedoms** has been directly applied to this research.

For a synthesis tool to meet the requirements on flexibility, the program has to make decisions in some order such that earlier decisions do not overly constrain later decisions. It must also have some method for computing the effect of a single design decision on the overall area and speed of the resulting hardware.

Thus, the synthesis problem is described as follows: The program should input a dataflow description of the hardware behavior and optional constraints on area and performance, and must output a data path structure consisting of registers, operators, and required interconnections, along with a time schedule giving the ordering of operations. The program should make the most constrained decisions first, so that the ordering of decisions does not greatly affect the optimality of the resulting design. It should adjust either to area or speed constraints and it should be able to measure the impact of each design decision to avoid extensive search of the design space. Finally, the program should be able to restart or backtrack when it is clear constraints will not be met with the current strategy.

The next section of this chapter describes the operation of the Modified Automatic Hardware Allocator (**MAHA**), which synthesizes non-pipelined designs in the manner described above. Restrictions of the original version of **MAHA**

[PPM86], as well as refinements and additions which have since been incorporated, are described. The **MAHA** program structure and loop handling are detailed. Synthesis examples and limitations conclude the chapter.

2.2 Overview of the MAHA Algorithm

MAHA carries out the synthesis tasks described above in the following manner. First, a list of operations is read by **MAHA**, followed by a list of values transferred between these operations. (An example dataflow graph containing operations as nodes and values as edges is shown in Figure 2.1.) The longest path (in time) from top to bottom or *critical path* is located; **MAHA** divides this path into p time steps (or partitions) of equal duration. Each time step now represents one minor cycle or register transfer. **MAHA** allocates operators for the critical path in a first-come first-serve fashion, with multiple operations sharing resources as long as the operations do not occur in the same time-step. Then, **MAHA** decides which of the remaining or *off-critical* path operations have the least scheduling freedom. Thus, the first operations scheduled, which may have scheduling difficulties, get the first chance to share resources. Operations scheduled later may find most resources fully utilized; however they have the greatest scheduling flexibility, and thus are more likely to find a free resource in some time slot. **MAHA** adds resources as necessary when it schedules these off-critical path operations.

At some point, **MAHA** may run out of resources due to an area constraint. At this point, it repartitions the critical path into more time steps, and begins allocation over. Adding more partitions allows scheduling the resources to be used by more operations than before; thus, fewer resources may be required. If area is to be minimized, **MAHA** will incrementally add as many time steps as possible without violating the timing constraint; if speed is to be maximized, **MAHA** will add resources as long as it can without violating the area constraint. (Note that both area and time are optionally adjusted by the associated register and/or multiplexer scheduling and allocation.) A general overview of the Modified Automatic Hardware Allocator and scheduler is included in Figure 2.2.

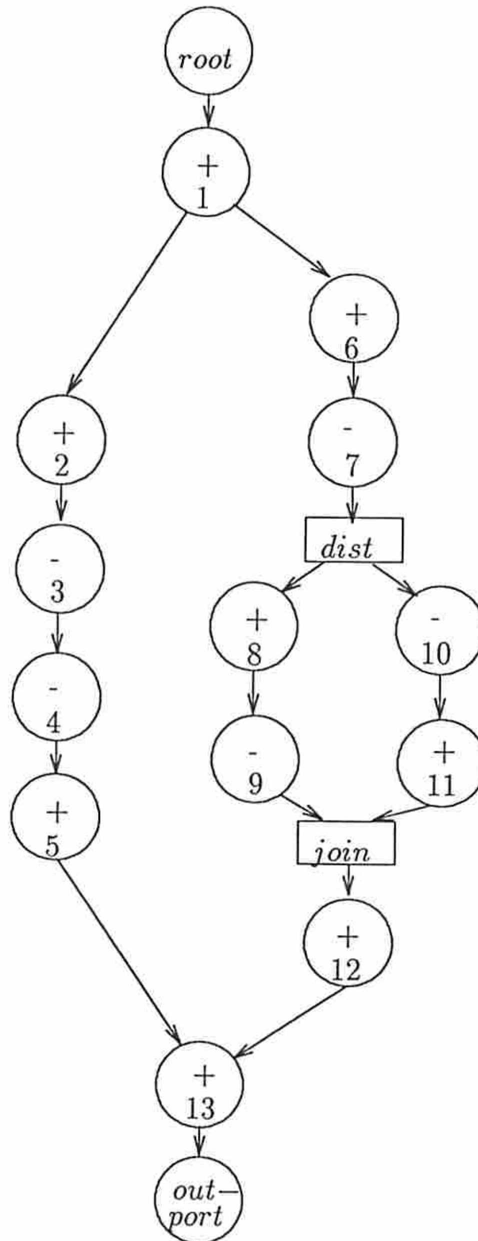


Figure 2.1: Example dataflow graph

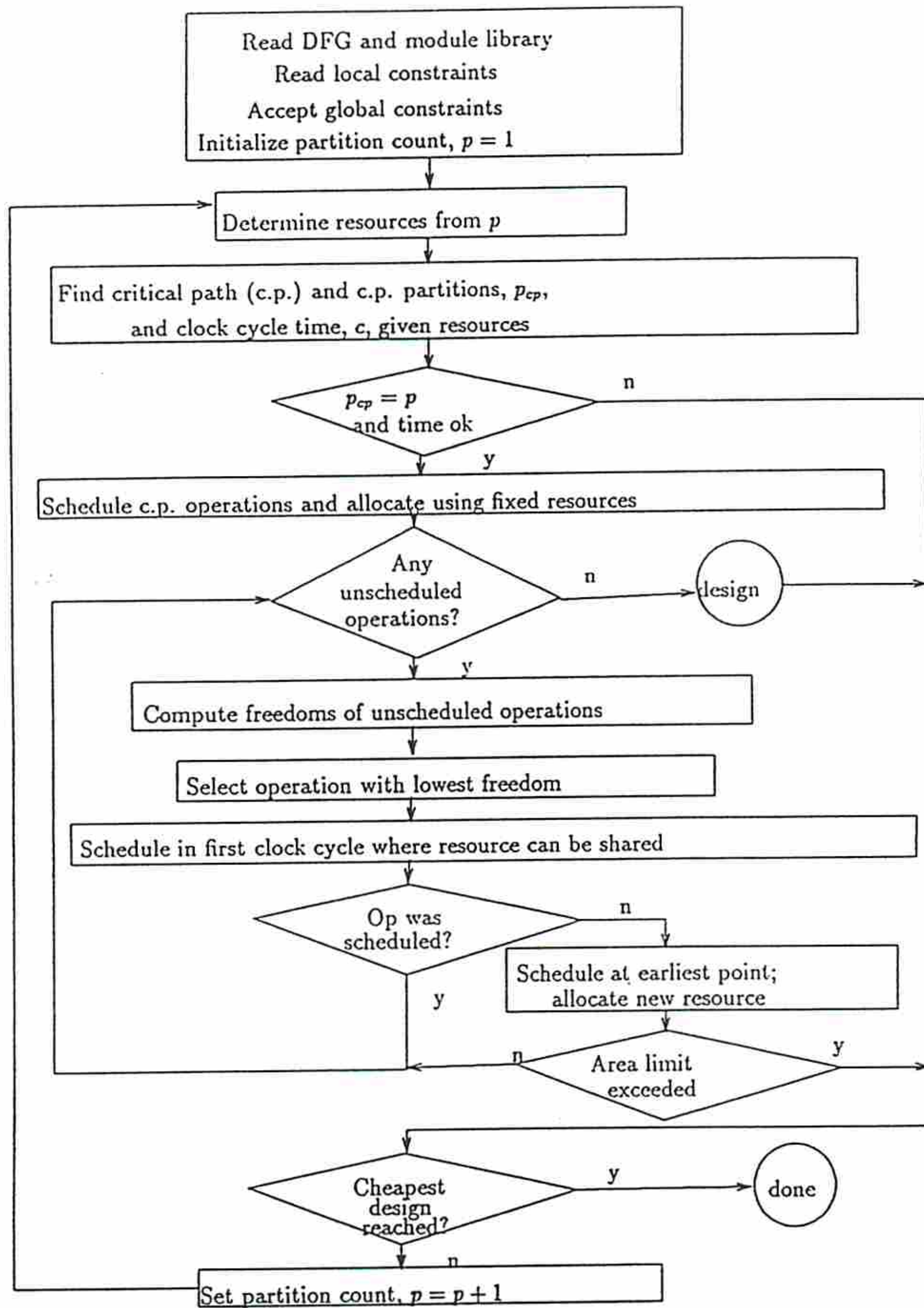


Figure 2.2: General Overview of MAHA Algorithm

Each node in the dataflow graph represents an operation to be performed by a piece of hardware. The critical path operations, which include operations on multiple critical paths, are removed from the set of all operations and are assigned hardware first. Since the critical path operations are sequentially ordered, there are no timing conflicts. Also, since the critical path determines the overall speed, doing the critical path assignment first ensures the fastest possible resulting design. Further, once the critical path has been assigned, one has a lower bound on the total area. Thus, the approach used in this heuristic partitions the problem.

The method for computing the critical path in the original version of **MAHA** is carried out by a program by Park called the Clocking Scheme Synthesis Package (CSSP) [PP85a]. Although CSSP was originally intended to clock data paths already designed, it was used within **MAHA** to avoid duplication of effort. This program takes a set of operations and values and computes optimal clocking schemes, the critical path, and clock cycle times for single and multiphase clocks. Given this information, hardware can be assigned to clock periods; specifically, critical path operations are assigned to a time step and bound to hardware modules for that clock cycle. Operations which are not on the critical path could share this hardware during other clock cycles.

Before discussing the algorithm used in **MAHA** further, the notion of **freedom** introduced in Chapter 1 is again described. The freedom of an operation is defined as the difference between the time when the input values are available for a given operation and the time when the result of that operation is required, less the propagation delay of the hardware. Once the critical path has been assigned, the remaining operations are assigned by iteratively computing the freedoms of each operation and scheduling the operation with the least freedom.

2.3 Evolution of MAHA

In its initial form, **MAHA** performed a subset of the overall data path design as originally envisioned by Parker. One limitation involved the critical path partitioning scheme whereby the dataflow could be divided or *serialized* into a maximum of n time-steps where n is approximately the number of operations

on the critical path. This restricted the degree by which **MAHA** could serialize a design for lower area, a limitation which was resolved with delay operation insertion discussed in this section.

Further, register and multiplexer effects were ignored in the original utility. Solely using operator area may give misleading results. For example, assume a design with two clock cycles has twice the operator area consumption of a design with four cycles and both exhibit identical overall delay. One might be tempted to discard the larger design. However, additional multiplexers and registers are needed for the smaller design which increases both its delay time and area. Furthermore, control area - which is not addressed here - is also increased. Consideration of routing and storage elements were added to **MAHA** and are discussed in this section.

A third consideration in synthesis is localized constraints. For example, two operations in a dataflow graph may be constrained to occur within/outside a certain interval. Extensions to the synthesis program addressed the need for localized constraints.

Finally, wiring area is another aspect of the data path area; however, this aspect of data path analysis is handled externally to **MAHA** using the **PLEST** utility [KP86]. In its current form, **MAHA** computes and outputs values needed to estimate the wiring area, but does not perform any internal wiring area analysis.

2.3.1 Synthesis of Serialized Designs

The first release of **MAHA** serialized designs up to the limit of as-soon-as-possible (and as-late-as-possible) scheduling of the critical path. There are two problems with this approach. First, scheduling was not resource sensitive. If two identical operations in series along the critical path had a collective delay smaller than the minimum clock time, they might always be scheduled into the same time step. Hence, they could never share the same operator which may result in a lower circuit area. Only a resource sensitive scheduling technique would eventually force these operations into different time steps.

Second, resource-sensitive scheduling of off-critical path operations was also lacking. In particular, when identical operations lie either on and off (or both off) the critical path, there is the potential that they would be scheduled into the same time-step and not be able to share resources. One approach would be to arbitrarily force the off-critical path operation into a different time step. However, this may result in a scheduling conflict with another operation and, more importantly, may extend and thereby *change* the critical path. Given this impact, it is also quite probable that operations scheduled up to this point could more efficiently utilize resources given the extra "breathing" space.

A solution is to stretch the critical path by the introduction of *delay* operations. These operations, which have no area associated with them, are inserted along the critical path where resource conflicts occur with (or between) off-critical path operations. One such example is shown in Figure 2.3 where a delay operation has been inserted prior to *-7*. By allowing time for the off-critical path operation *-3*, it could utilize the same hardware resource as *-7* and lower the overall circuit area.

Once the modification to the graph is completed, it is synthesized normally. Insertion of delays may continue until only one resource of each operation type is needed for a design; at this point, the dataflow algorithm is fully serialized and no further area reduction is possible (given the module set).

2.3.2 Register Extensions

Although registers may be a small percentage of the total area in large designs, they can represent an appreciable area in smaller or serialized designs. In addition, register delay affects the clock cycle time. A simple linear strategy to compute register area and delay was incorporated into **MAHA**.

After operator allocation is complete, each value in the dataflow graph is visited; if the source and destination are in different time steps, the value is a candidate for register assignment. If a register has been attached to this value from the operation in a prior check, additional registers are superfluous. Also, values emanating from *root* do not have registers attached.

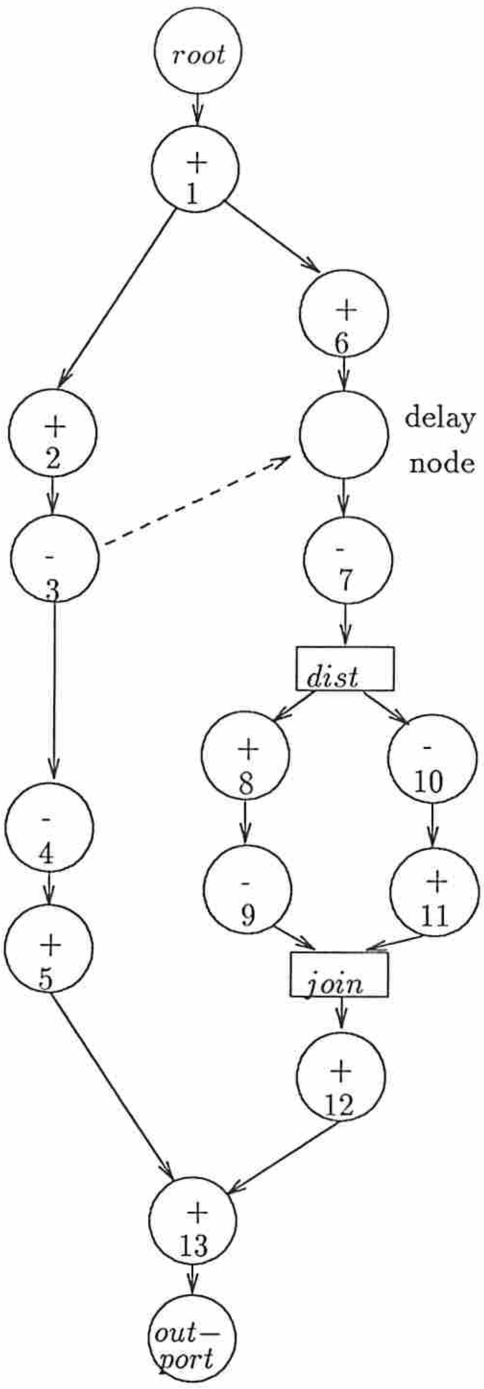


Figure 2.3: Example delay operation insertion

This approach accomplishes register allocation in $O(values)$ time and provides a hard upper bound on the number of registers required; the results may be improved by taking advantage of better external algorithms.

Register delay is also a consideration for data path synthesis. Since a register is attached between each time step, the clock cycle time is increased to accommodate the register delay.

2.3.3 Multiplexer Extensions

Multiplexers can also impact area in a design with extensive sharing. For more accurate results, and in particular for assessing design changes, the area of multiplexers must be considered. **MAHA** adds multiplexers as needed for operator sharing. Multiplexers are not shared.

Within any time-step where multiplexers are utilized, multiplexer delay may exceed the "clock slack" time. "Clock slack" is the difference between the clock cycle time and the maximum delay of all paths through operations assigned to that specific time-step which contain the operation being allocated multiplexers. If there is sufficient slack, a given *multiplexer tree* may not impact the clock cycle time. In the worst case, the clock time would be increased by the particular conflicting multiplexer tree delay. **MAHA** increases the clock time to accommodate multiplexer tree delay.

Area and delay calculations for each *multiplexer tree* are incorporated into **MAHA**. Basically, for a given m-bit n:1 multiplex, a *tree* consisting of 2:1 multiplexers has an area of

$$A_{muxtree} = m(n-1)A_{mux} \tag{2.3.1}$$

where A_{mux} is the area of a 2:1 multiplexer [Kur87]. The time delay is determined by the maximum depth of the tree and is written as

$$T_{muxtree} = \lceil \log_2 n \rceil T_{mux} \tag{2.3.2}$$

where T_{mux} is the delay time of a 2:1 multiplexer.

2.3.4 Extension for Localized Constraints

In many designs, constraints may be imposed upon a local area of the algorithm. As an example, due to some constraint, one operation cannot begin until another has been started. Synthesis bounded by local constraints is not a new concept, although only one synthesis package to date is known to accept them [NT86].

Local constraints should allow specifying minimum and/or maximum delays between any two operations. In a dataflow graph, there is an implicit constraint given by the ordering of the operations. For example, an value between operation x and operation y can be viewed as "operation y starts after operation x finishes", in addition to the data transfer aspect. Extending this concept, one could model a minimum delay between any two operations by inserting an edge along with a delay operation (if needed), provided no loops are introduced by adding the new edge.

Contrary to specification of minimum delay, maximum delay cannot be represented solely by modifying the dataflow graph. As **MAHA** serializes the graph, the delay between operations naturally lengthens. There is no inherent mechanism by which the delay between any two operations can be limited. Hence, the simplest method is used for **MAHA**: modify the scheduling algorithm to detect and avoid constraint violation (both minimum and maximum). When an operation is scheduled, its constraints are checked and the schedule discarded if a violation occurs.

2.4 The MAHA Program Structure

The original version of **MAHA** was written in Franz LISP. Since then, **MAHA** has been ported to the C programming language with several enhancements added. However, the basic structure of **MAHA** has remained unchanged throughout.

Discussion of **MAHA** is divided into five separate areas: algorithm input, clock cycle generation, critical path partitioning and allocation, off-critical path analysis and allocation, and graph extension. An overview of the complete procedure is then given.

2.4.1 Algorithm Input

There are four inputs to the algorithm: behavior in the form of a dataflow graph, library of hardware modules which can implement the operations in the dataflow, local constraints, and global constraints.

2.4.1.1 Dataflow Graph Input

The behavior of a system is described using a dataflow graph where operations are represented as nodes and value transfers described by directed edges. **MAHA** expects the graph to have a single entry point from a node specifically named *root* which has no incoming edges. Similarly, the graph terminates at a single exit point named *outport* which has no outgoing edges. All other nodes must lie along some acyclic path from *root* to *outport*. An example graph was shown in Figure 2.1.

Note that *root* and *outport* are "place holders"; the function performed is essentially a no-op with no area or delay associated with them. Thus, they have no impact on the synthesis results.

In the actual program, node tags which specify the operation as being the **top** or **bottom** of a loop are also accepted. Although **MAHA** does not accept cyclic graphs, these tags allow it to generate useful designs for algorithms which had loops prior to transformation. An external algorithm which transforms cyclic graphs into acyclic ones prior to synthesis is presented later in this chapter.

Finally, the graph is colored to delineate mutually exclusive operations. Two special operations designated *dist* and *join* are used to initiate and terminate a multi-way conditional branch. The square boxes labelled **d** and **j** in Figure 2.1 are distribute and join, respectively. Each *dist* must have a matching *join* and all operations within the specific conditional must have a path to the matching *join*. Operations **+8** and **-9** are mutually exclusive to **-10** and **+11**. Thus, the same operator could be used for **+8** and **+11** since only one is active at any time. Coloring of a graph for conditional branches is described in Park [Par85].

2.4.1.2 Library Generation

After the hardware library is selected by the user, **MAHA** analyzes each type of operation performed in the dataflow graph and generates a list of all components in this library which can perform it. If more than one hardware module can perform a given operation, **MAHA** averages the area and speed of all such modules to form a single "average module" type associated with that particular operation. Operations which have a local delay constraint require an additional check to eliminate non-compliant hardware modules which do not meet this constraint.

Having an average library simplifies hardware assignment as each operation has only a single component in the library which can implement its function. Thus, the propagation delay of every operation is fixed. It also allows centering the design in a chosen area in the design space without fine tuning for a particular module selection.

MAHA is not intended to perform module selection. If module selection is performed prior to synthesis, then a reduced library can be passed to **MAHA**. Using this method along with appropriate function labelling on the operations in the dataflow graph, each operation will have only one hardware type which can implement the function. This fixes the propagation delay; the area associated with the implementation depends upon hardware resource sharing. This is usually the mode in which **MAHA** is operated.

In our example of Figure 2.1, one **adder** and one **subtractor** module type would be selected; the **adder** has a delay of **100** and area of **200**, the subtractor **110** and **220**. The number of instantiations (uses) of these operators depends upon the particular design.

2.4.1.3 Local Constraint Specification

MAHA can optionally accept local user constraints. These constraints specify any of the following:

- minimum propagation delay of a given operation
- maximum propagation delay of a given operation

- minimum delay between two operations
- maximum delay between two operations

For delay between two operations, either the starting or ending time of the operation can be specified. These constraints are independent of the actual graph dependencies. An example local constraint would be

$$T_b(-7) \leq T_e(-4) - 50 \tag{2.4.3}$$

which reads "operation -7 should start at least 50 time units before operation -4 completes".

2.4.1.4 Global Constraints

When designing hardware, one is usually confronted by constraints on the area and maximum delay associated with the circuit. Global constraints are

- A_u, the maximum circuit area allowed, and
- T_u, the maximum circuit time.

Usually one global constraint is minimized while obeying an upper limit on the other.

2.4.2 Clock Cycle Generation

Feasible clock cycle times are constructed by **MAHA** prior to synthesizing any designs. Each operation is assigned a propagation delay associated with its hardware module. A union of all maximum times between any pair of operations which are connected via some path form the clock cycle array; theoretical foundations and the algorithm are detailed in [PP85a]. The clock times are bounded on the low side by the maximum delay of any operator (c_{min}) and naturally bounded by the maximum delay from *root* to *outport* on the high side. The first condition is imposed by a precondition that a given operation must be completed in the same cycle that it is initiated; however, multiple operations per clock cycle are allowed.

Using the example module library described earlier, then $c_{min} = 110$ and the maximum delay is 720. The list of all possible clock times, in order, is 110, 200, 210, 220, 300, 310, 320, 410, 420, 520, 620, and 720, which forms the clock array.

2.4.3 Critical Path Partitioning and Allocation

With the dataflow graph, module library, *and* the number of partitions, p, of equal clock time, c, into which the graph should be divided, the critical path can be determined. (The clock cycle time is one chosen from the clock array.) It will be shown later how to determine the best clock time c given p.

First, the minimum number of resources required is computed using lower-bound area-time theory for non-pipelined synthesis [JMP88]. Given the number of partitions, the number of hardware (operator) resources (o_i) necessary for implementing n_i operations of type i in the dataflow graph is

$$o_i = \left\lceil \frac{n_i}{p} \right\rceil \tag{2.4.4}$$

giving a minimal area of

$$A_m = \sum_i o_i A(i) \tag{2.4.5}$$

where $A(i)$ is the area of hardware resource type i. (n_i is the maximum number of modules of type i needed for the most parallel design; n_i may be less than the actual number of operations of type i in the presence of conditional branches.) If there is an area constraint (A_u) and $A_m > A_u$, there is no need to synthesize this particular design. A higher partition count which results in an area less than A_u must be chosen.

The critical path marking algorithm is a variation of that given in Park [PP85b]. Each operation is assigned a propagation delay equal to its component delay. All possible paths from *root* to the *output* are followed and the maximum time at each operation is retained. Each operation is marked as it is visited to prevent traversing any portion of the graph more than once. Hence, the critical path is found in $O(n^2)$ time, where n is the number of operations in the graph.

Given clock cycle time (c), Parks original algorithm which determines the number of critical path partitions, p_{cp}, is modified to consider hardware resources

```
/* Inputs are minimal resources, o_i, and clock time, c. */
   Initialize partition count and partition time, p_cp = 1 and Tp = 0.
   Set resources left to maximum available, u_i = o_i for all i.
   Starting from top (bottom) of critical path for ASAP (ALAP) scheduling,
   while more unassigned operations remain in critical path,
      Select next unscheduled critical path operation k where type is j.
      While delay too large to fit into current partition, Tp + Delay_k > c, or
      no resources of type j are left in this partition, u_j = 0, then
         Create a new partition, p_cp = p_cp + 1.
         Reset partition time, Tp = 0.
         Reset resources used, u_i = o_i for all i.
      Endwhile /* assign operation to partition number */
      Set resource link and adjust resources left, u_j = u_j - 1.
      Compute new time used in partition, Tp = Tp + Delay_k.
      Schedule operation k into partition p_cp.
   Endwhile /* scheduling critical path */
   Increment p_cp to give actual number of partitions.
   Return total partitions, p_cp, and delay, T = (p_cp - 1) × c + T_p.
```

Figure 2.4: MAHA Critical Path Partitions (p_{cp})

as described in Figure 2.4. Since resources are tracked, a *resource link* between the operator and operation is maintained for later use in critical path allocation. This algorithm schedules the critical path operations and also determines the total delay (T) associated with this design.

The number of critical path partitions, p_{cp}, is determined by the clock cycle time and the number of resources. However, the resources were computed from the number of partitions, p. Clearly, unless $p = p_{cp}$, the critical path schedule is invalid. To overcome this problem while also finding the lowest clock cycle time for which $p = p_{cp}$, p is arbitrarily set and the set of feasible clock times is examined. Starting with the minimum clock that is theoretically possible [JMP88] and the maximum clock time, a binary search is performed to obtain the lowest clock time which gives the desired resource-sensitive partition count, where $p = p_{cp}$.

```
/*
 * Inputs are minimal resources, o_i,
 * and desired partition count, p_desired, and
 * clock array, CLK[maxclk], which is in ascending order.
 /
   Using binary search, find minimum value of l such that
      partition count for CLK[l] equals p_desired, if possible;
      p_cp is determined (Figure 2.4).
   If l exists (p_cp = p_desired), then
      Set clock cycle time c = CLK[l], and
         circuit time T = c × p_cp.
      Allocate hardware to critical path;
         (turn all resource links into resource allocations).
   Endif /* check if partition count feasible */
```

Figure 2.5: MAHA Critical Path Scheduling and Allocation

With the minimal clock time which gives $p = p_{cp}$, scheduling and allocation of critical path hardware can now be accomplished. **MAHA** sequentially assigns hardware to each operation in the critical path, keeping track of hardware usage and area. A design is generated using both top-to-bottom or as-soon-as-possible (ASAP) scheduling/allocation as well as bottom-to-top or as-late-as-possible (ALAP) scheduling/allocation. An "assigned hardware" list is generated which contains all "purchased" hardware and the time range(s) currently assigned. Since each operation has only a single component type which can be used, allocating hardware to an operation only requires analyzing the usage of previously allocated hardware. If a component of the correct type has been previously assigned, the corresponding time ranges are examined to determine if that hardware component can be reused. If not, a new piece of hardware is purchased. This maximal sharing **greedy** algorithm is performed on the dataflow graph in approximately linear time. The critical path scheduling and allocation algorithm is shown in Figure 2.5.

Assume that the partition count was set to four ($p = 4$). With 7 *effective* add operations and 4 effective subtract operations, two **adders** and one **subtractor**

are initially allocated for the example graph implementation. Using a clock time of 200 does not violate the resources, but gives $p_{cp} = 5$ which is invalid. A clock time of 210 gives 4 partitions on the critical path as shown in Figure 2.6.

In **MAHA**, hardware area is equally as important as operator delay. Consider the subgraph depicted in Figure 2.7 which is part of a larger dataflow graph. Assume the delays of operators a, b, and c are identical. By solely considering the delay time, the right path ($bcbcbc$) is 50% longer than the left path ($aaaa$). However, if only one hardware resource is available for each type, then the left path must be divided into four partitions ($a \mid a \mid a \mid a$) whereas the right path can be divided into three ($bc \mid bc \mid bc$). With this resource dependence, *the critical path (time) of a design must be recomputed whenever the resources allowed changes.* A single evaluation of the critical path based upon operator delay is insufficient.

Besides resource dependence, the critical path of a design is also affected by clock cycle time. As the clock cycle time is changed, non-critical paths may now become the critical path. To illustrate, assume operator delays of $a = 50$, $b = 30$, and $c = 30$ for the dataflow graph in Figure 2.7. For clock cycle times greater than or equal to 60, the left path is the critical path (ignoring resources for the moment). However, for clock times less than 60, the right path becomes the critical path. Thus, *the critical path of a design must be recomputed whenever the clock cycle time changes.*

Local constraints also affect the scheduling of the critical path. Before each operation is actually scheduled, it is tested to determine if any of the user-specified constraints are violated. If not, **MAHA** will accept the scheduling. Otherwise, if scheduling the operation in a later time-step results in acceptance of the constraint, the latter schedule is used. In the case where the operation cannot be scheduled without violating a constraint, a message of the form "constraint violation" is given and **MAHA** tries again with a larger partition count.

A dataflow graph may have more than one critical path. In this case, **MAHA** selects only one for performing critical path allocation. The remaining critical paths are scheduled and allocated along with non-critical path operations. Since

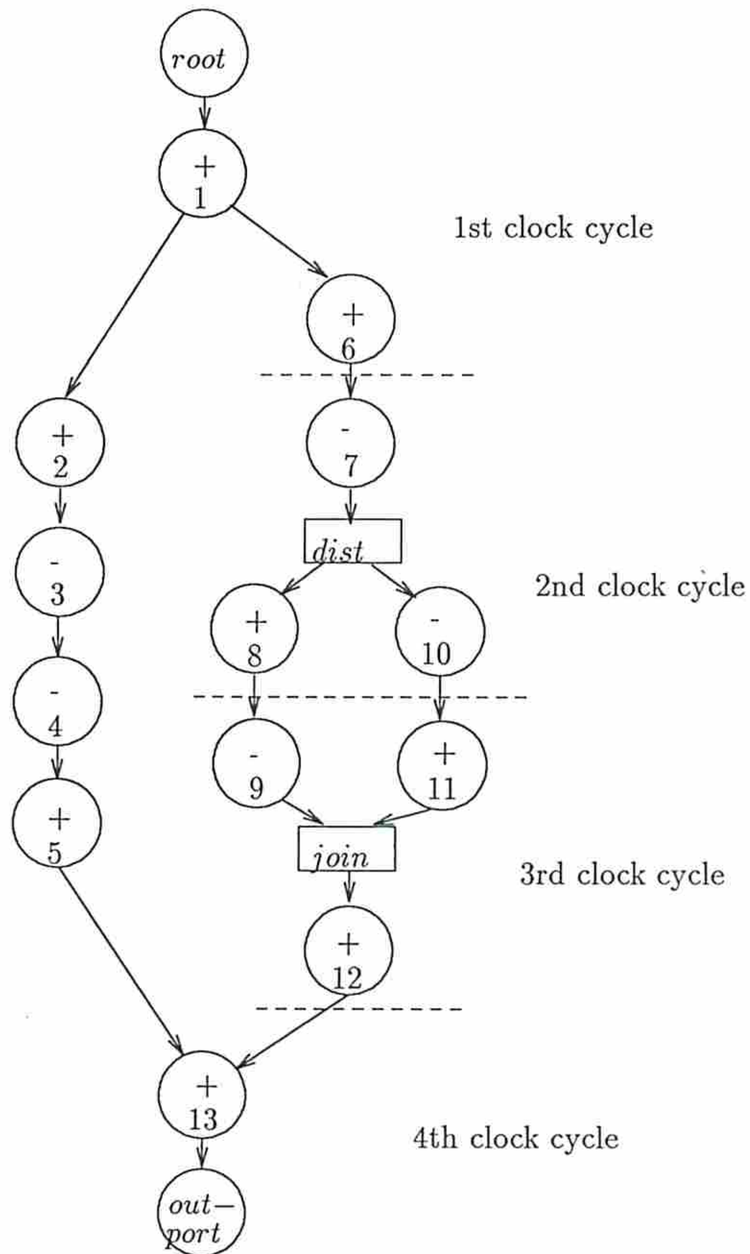


Figure 2.6: Example with Critical Path Partitioned

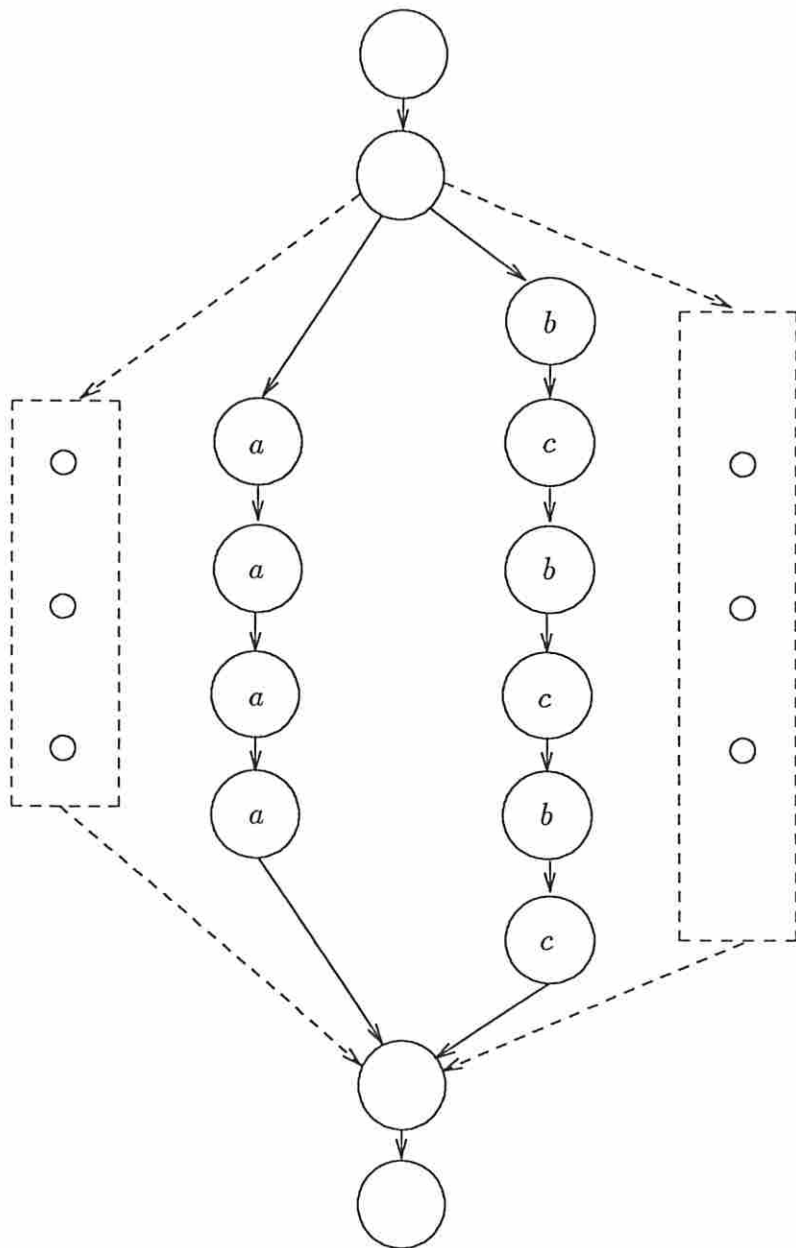


Figure 2.7: Demonstration dataflow graph # 2

the scheduling freedom of "ignored" critical path operations is limited as compared to other non-critical path operations, clearly these additional critical paths will be scheduled before other off-critical path operations.

The presence of multiple critical paths is not a major issue in non-pipelined designs. Multiple critical paths usually occur in repetitive designs, where each critical path contains the same operations in the same order. Since every critical path is identical, one can arbitrarily select any of them and obtain the same result. Furthermore, highly repetitive dataflow graphs of this nature are more aptly suited to pipelined design where **MAHA** is less likely to be used.

2.4.4 Off-Critical Path Analysis and Allocation

After completing the critical path allocation, off-critical-path operations are assigned hardware. Initially, each operation not previously allocated is analyzed to determine its freedom. Essentially, freedom is the slack time in which an operation can be performed without lengthening the critical path.

Let $\mathcal{N}_{parent}(k)$ be the set of all direct predecessors of operation k; in other words, every operation with an outgoing value connected to operation k is a member of this set. Similarly, $\mathcal{N}_{child}(k)$ is the set of all direct successors to k. Then, the earliest time $operation_k$ can be scheduled is

$$T_{early}(k) = \max_{j} \left(T_{early}(operation_j) + D(operation_j)\right) \tag{2.4.6}$$

where $operation_j \in \mathcal{N}_{parent}(k)$ and $D(operation_j)$ is the delay associated with operation j. Similarly, the latest time is

$$T_{late}(k) = \min_{m} \left(T_{late}(operation_m)\right) - D(operation_k) \tag{2.4.7}$$

where $operation_m \in \mathcal{N}_{child}(k)$.

Operations which have been scheduled have T_{early} and T_{late} set by the clock cycle and their position within that cycle. Note that previously allocated critical path operations have had their values of T_{early} and T_{late} assigned, starting with $T_{early}(root) = 0$, before off-critical path allocation occurs. Since all paths must eventually connect to a critical path operation, *root*, or *outport*, the recursive

```
/* Input is clock time, c. */
offcp_main:
   Compute freedoms of all operations not scheduled (or allocated).
   Select operation k with lowest freedom(T_late − T_early); type is j.
   Set initial indices h and l to the early and late time partitions;
      h = ⌊T_early(k)/c⌋ and l = ⌊T_late(k)/c⌋
/* Find partition to share resource, or just arbitrarily assign */
   While more partitions to check (h ≤ l), then
      Set u_j to resources used of operation type j in partition h.
      If resources of type j still left (u_j < o_j), then Break while loop.
      Else /* no resources of type j left */
         Set next partition, h = h + 1.
         If out of partitions to use (h > l), then
            Reset initial h value (h = ⌊T_early(k)/c⌋).
            Break while loop
         Endif /* out of partitions - abort while */
      Endif
   Endwhile /* determine partition to assign operation */
```

Figure 2.8: MAHA Off-Critical Path ASAP Scheduling and Allocation: Part 1 of 2

nature of Equations 2.4.6 and 2.4.7 is bounded. A list of freedom for all non-allocated operations is formed in $O(n^2)$.

The off-critical path operation with the smallest freedom (tightest constraint) is chosen for scheduling as described in the algorithm of Figure 2.8. If the freedom is so small that the operation must occur during one specific time stage, then the operation is scheduled in that stage. However, it is more common to see freedoms which allow a operation to be assigned to any one of a number of consecutive stages. In this case, the hardware allocated in the allowed stages is sequentially examined and the operation is assigned to the first stage where hardware sharing can occur. If none of the stages allows resource sharing, the earliest stage for ASAP scheduling (latest stage for ALAP scheduling) is arbitrarily chosen.

Once the off-critical path operation has been assigned to the chosen stage, either existing hardware is shared or a new operator is allocated. Figure 2.9

```
/* Found partition, now schedule and allocate */
   Assign operation k to partition h.
   Update resources used in partition, u_j = u_j + 1.
   If resources exceeded previous limit (u_j > o_j), then
      If first operation which cannot share resource (n_ocp not set), then
         Set n_ocp to the node's index (k) to identify the
            off-critical path operation.
      Endif /* determine where to stretch graph at */
      Increase limit, o_j = o_j + 1.
      Increase total area, A, by area of operation type j.
   Endif /* resource limit exceeded */
   If area constraint (if any) exceeded (A > A_u) and n_ocp set, then
      Exit routine with partial design.
   Endif /* constraint check */
   If any operations left unscheduled, then Goto offcp_main.
```

Figure 2.9: MAHA Off-Critical Path ASAP Scheduling and Allocation: Part 2 of 2

contains a description of the allocation portion of the off-critical path algorithm. Exceeding the maximum area during off-critical path allocation may result in re-partitioning of the dataflow graph. Otherwise, after allocating each chosen off-critical path operation, **MAHA** recalculates the freedoms for all operations affected by it and picks another off-critical path operation which has not been scheduled or allocated. This process continues until all operations have been allocated hardware.

2.4.5 Dataflow Graph Stretching

When a dataflow graph has been scheduled into its maximum number of time-steps along the critical path, but has not yet reached its minimal resource limit, **MAHA** lengthens the dataflow graph to reduce the likelihood of a scheduling conflict. This function can be performed with a partially synthesized design provided *at least* one scheduling conflict has occurred since an area of resource conflict has been identified.

(a) Set h to most recently assigned partition of operation n_{op};
set j to the operation type.

(b) Find critical path operation k such that:

- partition of operation k is h,
- type of operation k is j (if exists in this partition), and
- there are no other operations of type j in partition h which occur later.

(c) Insert new delay operation d into graph such that:

- All predecessors of operation k now point to delay operation d,
- a connection is made between operations d and k, and
- delay of operation d is set to push the following operations into the next partition.

Figure 2.10: MAHA Critical Path Stretching

The procedure used to stretch a dataflow graph is simplified since the critical path scheduling algorithm resolves resource conflicts along a given path. As such, the only scheduling conflict that arises is between critical path and off-critical path operations or solely between off-critical path operations. Resolution involves the insertion of a *delay* operation into the critical path at the point of overlap as described in Figure 2.10 and depicted in Figure 2.3. (The algorithm describes ASAP node insertion; ALAP is similar.) The selected critical path operation and its successors are delayed so that the off-critical path operation can be allocated to the same hardware as the conflicting operation. This *delay* operation is a **MAHA** artifact that has no associated area or "real" delay in the actual design; however, the scheduling algorithm treats this operation like any other and achieves the desired results. The delay of this operation is adjusted depending upon the clock cycle time to force the following operations into the next clock cycle.

Read in dataflow graph; check for errors. (Section 2.4.1.1)
Read in local constraint file, if any. (Section 2.4.1.3)
Read in module library; construct "average library". (Section 2.4.1.2)
Construct array of all possible clocks (c). (Section 2.4.2)
Get user global constraints, T_u and A_u. (Section 2.4.1.4)
Initialize minimum and current partition counts, $p = p_{min} = 1$.
Clear all hardware resources.
Clear first conflicting off-critical-path operation, n_{ocp}.
Assign minimum clock time, c_{min}, to delay of slowest module.

Figure 2.11: Initialization of MAHA

2.4.6 MAHA Synthesis Procedure

In order to synthesize a graph, several values must be initialized and operations performed as summarized in Figure 2.11.

The actions taken by **MAHA** are dictated by the global constraints. These are supplied by the user for both time (T_u) and area (A_u); a value of zero instructs **MAHA** to minimize that parameter with respect to a limit on the other. Assigning a zero to both forces **MAHA** to generate all possible designs.

With global constraints set, the dataflow graph is initially divided into one stage where every operation requires a distinct hardware resource except for sharing which occurs on mutually exclusive paths. **MAHA** synthesizes designs starting with this most parallel (fastest and most expensive) towards the most serial (cheapest and slowest) which has some larger number of partitions, p.

As the graph is partitioned into two or more stages (clock cycles), hardware can be shared thereby lowering the data path area. Figure 2.12 shows a simple dataflow graph that has been partitioned into two clock cycles indicated by the solid line near the middle of the graph. On the left path, the top operation a is implemented using hardware resource A. The second a must use a different hardware resource of the same type. The clock cycle ends after the second operation and a register is used to store the result.

A third a occurs in the second clock cycle. Since the results from the first and second operations are complete and the incoming value is stored in a register,

hardware resources which performed those operations are free to be reused as shown by the dashed lines. Similarly, operations b and c require two hardware resources. Note that resources B_1 and C_1 are unused during the second clock cycle.

By partitioning a graph into many clock cycles ("serializing" the dataflow algorithm), operator hardware can be reduced. This is offset by additional resources required for registers to store values and multiplexers or buses for routing of values to a given hardware resource. These lower area designs are usually slower. Depending upon the clock cycle time used (as compared to operator delays), some portions of the design may exhibit idle time. If all operations in Figure 2.12 have equal delay, then the left path is 50% idle; each clock cycle takes 4 time units, but hardware is only active for 2 of those. In addition, the overall circuit time has increased from 6 to 8 time units. There are also control area effects which will be addressed in the next chapter.

MAHA synthesizes designs by scheduling each operation into a specific clock cycle and allocating a hardware resource to perform the operation, given starting resources and number of partitions. Successive design steps increment the number of partitions until either no further designs meeting the constraints are possible, or the cheapest possible design is found (e.g. only one hardware resource for each operator type). An overview of the **MAHA** algorithm is given in Figures 2.13 and 2.14 where T is time, A is area, and T_m and A_m are the minimum time and area possible. n_{ocp} is the first off-critical-path operation encountered during synthesis which cannot share a *utilized* resource.

If a constraint on both area and time is given, **MAHA** searches for all non-inferior designs meeting these constraints. If either parameter is being minimized, **MAHA** will search for the single best design. **MAHA** exits early with the best design if time is minimized with a limit on area, this limit has been met, and the graph is about to be stretched; further partitioning can only result in slower designs.

Conversely, if area is being minimized with respect to a limit on time, **MAHA** will continue generating designs until the limit on circuit time is exceeded or the cheapest design is produced. The best design produced up to this point is chosen.

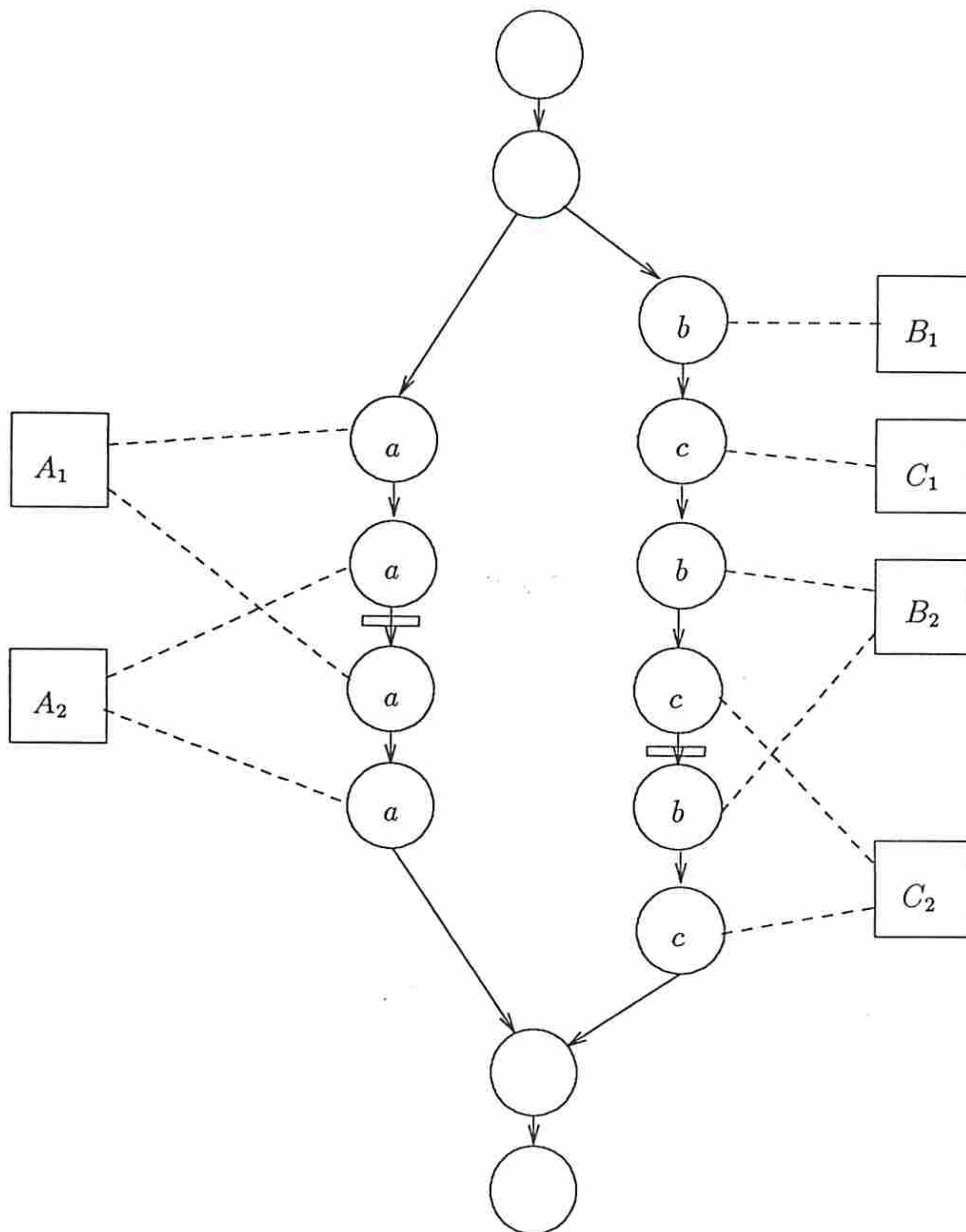


Figure 2.12: Demonstration dataflow graph # 1

```
/* Determine new resources needed and if max serialization reached */
/* Enter loop with p = p_min = 1 */
main_loop:
   Determine minimum resource area, A_m, needed given partitions, p.
   If first loop or resources changed from last loop, then
      calculate new maximum number of partitions, p_max,
      which is the number of partitions when clock time is c_min.
   Endif /* resource change or first loop */
   If graph is past maximum serialization, p > p_max, then
stretch_entry:
      If resources at minimum possible (one of each type), then
         display best design(s) and exit.
      Else if an off-critical-path operation was allocated a
      new resource in the last iteration (n_ocp), then
         Stretch the dataflow graph.
         Set partition count p to its minimum possible value, p_min.
      Else increment the number of partitions (p = p + 1)
      /* Either an n_ocp will be found or minimum resources will be reached */
      Goto main_loop.
   Endif /* graph at maximum serialization */
/* Now perform preliminary check to see if circuit exceeds constraints */
   If area constraint (A_u) provided and is exceeded (A_m > A_u), then
      Increment the number of partitions (p = p + 1); Goto main_loop.
   Endif /* area constraint check */
   Calculate minimum circuit time (T_m = c_min × p).
   If time constraint (T_u) provided and is exceeded (T_m > T_u), then
      Goto stretch_entry. /* stretch graph (if possible) */
   Endif /* time constraint check */
```

Figure 2.13: Overall operation of MAHA: Part 1 of 2

```
/* Critical path synthesis and constraint check */
   Perform critical path scheduling and allocation (Section 2.4.3);
      produces partition count p_cp and circuit time T.
   If the partition count p not reachable (p_cp ≠ p), then
      If partition count at minimum (p = p_min), then
         Set new minimum (p_min = p + 1) due to graph stretching.
      Endif /* increase minimum partition count */
   Else /* Off-critical path synthesis and constraint check */
      If time constraint (T_u) provided and is exceeded (T > T_u), then
         Goto stretch_entry. /* stretch graph (if possible) */
      Endif /* time constraint check with actual time, not estimate */
      Perform off-critical path scheduling and allocation (Section 2.4.4);
         produces circuit area A and may set n_ocp.
      If area does not exceed constraint, if any, (A ≤ A_u), then
         Update list of feasible designs.
      Endif /* final area constraint check */
   Endif /* partition count not reachable check */
   Increment the number of partitions (p = p + 1); Goto main_loop.
```

Figure 2.14: Overall operation of MAHA: Part 2 of 2

Finally, constraints are tested during both critical path and off-critical path synthesis. Thus, **MAHA** may only have synthesized a partial design before detecting a constraint violation and terminating construction of the current design.

Upon completion, **MAHA** displays the component types used and all instantiations of them including the clock cycles where they were utilized. As an aid to other portions of the ADAM system, **MAHA** can generate a file of the clock cycle assigned to each operation, hardware resources used, and conditional coloring.

2.4.7 Runtime Analysis

In a dataflow graph with N nodes (operations) and E edges (values), the number of *unique* edges is bounded by the number of operations in an acyclic representation. (For this analysis, edges arising from the same source and terminating at the sink are not *unique*.) The first node may have $N-1$ edges emanating from it, but the second node can have only $N-2$ edges since a "loop" back to the first node is prohibited. By expansion and series substitution,

$$E \leq \frac{N \times (N-1)}{2} \tag{2.4.8}$$

or $E \propto N^2$.

Prior to synthesis, **MAHA** generates a list of all possible clock times. This consists of the union of the maximum path between two nodes over all node pairs. The maximum path from a single node to every other node is found in $O(E)$ time. Thus, construction of all possible $(N \times (N-1))$ clock times takes $O(N \times E)$ or $O(N^3)$. These clock times are then sorted which has a complexity of $O(N^2 \log N)$.

During the first program loop, or when either the resources or the clock cycle time change, the critical path must be determined, which takes $O(N^2)$ time. The critical path is scheduled and has hardware allocated in linear time. Freedom of off-critical path operations is found in $O(N)$ time; in the worst case, scheduling and allocation of all off-critical operations nodes is accomplished in $O(N^2)$. With

up to N designs possible from the most serial to most parallel design, synthesis of a *fixed dataflow graph* may take $O(N^3)$.

Construction of the entire design space, which might be necessary to find the best single design meeting some constraints, may involve insertion of delay operations to "serialize" the design. This alters the dataflow graph. In the worst case, where a dataflow graph consists of N parallel paths with one operation in each path, up to $N-1$ delay operations might be inserted. With synthesis of a fixed dataflow graph taking $O(N^3)$, worst case synthesis of all possible designs is accomplished in $O(N^4)$ time.

2.5 Loop Transformation in Data Path Synthesis

Synthesis tools for generating RT-level designs from behavioral descriptions often simplify the problem to make it more manageable. One simplifying assumption prevalent in data path synthesis is the removal of loops. With the exception of an implicit outermost loop for a given design, loops internal to a description, to our knowledge, do not share resources outside of the loop in any of the currently published systems [GK84, PK87, Cam85, McF86, PPM86]. Rather, the loop is either ignored or modified by a human to remove feedback. A more robust solution is needed.

For approximating the area and time, removal of loops is a reasonable simplification for data path synthesis. After all, a loop is a control path construct and may have no data path hardware associated with it. However, if one wishes to synthesize a correct design which considers secondary effects such as multiplexers, buses, registers and, most importantly, a controller, then implementation and control of loops must be considered.

This section contains a method for transforming a cyclic dataflow graph. Such a mechanism is useful since current synthesis programs which determine the critical path or "color" the graph to detect mutually exclusive operations rely upon the acyclic nature of the dataflow graph [PP86, PG87]. However, a robust synthesis system must be fully cognitive of loops.

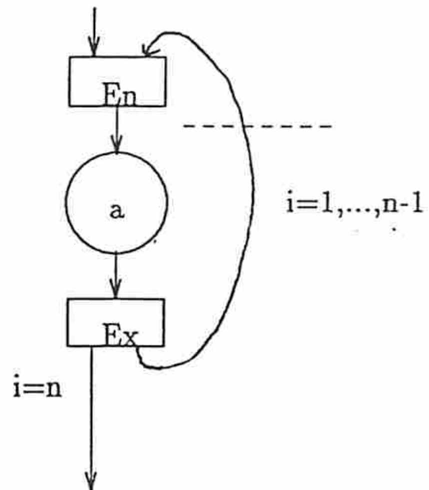
2.5.1 Simple Loop Transformation

Loops are explicitly represented in behavioral descriptions such as ISPS [Bar73]. (They are also easily detected using methods described in Aho [AHU74].) The difficulty lies in transforming the cyclic subgraph into an acyclic one suitable for data path synthesis while preserving the behavior. Even for the simple loop of Figure 2.15a which has a behavior of a^* (where a^* signifies that a is executed one or more times), disregarding the feedback edge may lose important control information. (**En** is the loop entrance and **Ex** is the loop exit.) Also, there is a precondition which forces the placement of a register somewhere in the feedback path as shown by the dashed line. This restriction could be extended to include the binding of values to registers at the top of the loop; the value(s) clocked at each iteration should use the same registers to reduce controller complexity. However, as will be shown later, by considering the expected execution of the loop, registers will be properly placed.

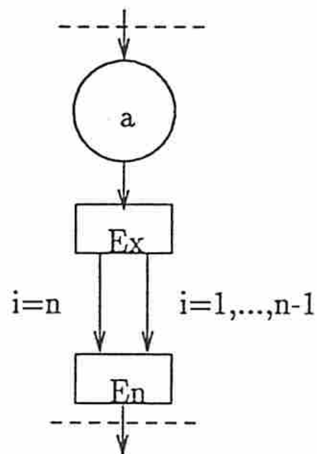
One hardware description language used by many of the current synthesis programs is ISPS. ISPS is easily converted into a dataflow graph through an intermediate representation known as the Value Trace (VT). Translation of ISPS into the VT cleverly avoids the problem of loops by instantiating each loop into its own VT-body (or subroutine) [McF78]. Each VT-body can be synthesized individually (since an outermost loop is implicit) before merging with larger pieces of the behavior. However, if the behavior is flattened to allow optimizations over multiple levels of the Value Trace, another approach is necessary to accommodate loops.

The transformations proposed here treat a loop as a dataflow graph having mutually exclusive branches. One branch is taken for the loop continuation (**restart** in ISPS) and the other is taken for loop exit (**leave** in ISPS). It is assumed that there are no hidden side effects for any operations within the loop. Thus, the portion of the subgraph containing the loop can be analyzed in isolation.

Coloring of a graph is used to detect mutually exclusive operations. Special operations in the dataflow graph called *distribute* and *join* signify the start and finish of one or more mutually exclusive paths. They are special operations being



(a) Original dataflow



(b) Transformed dataflow

Figure 2.15: Simple loop

merely "place holders" so that the dataflow is unbroken and dependencies are readily determined.

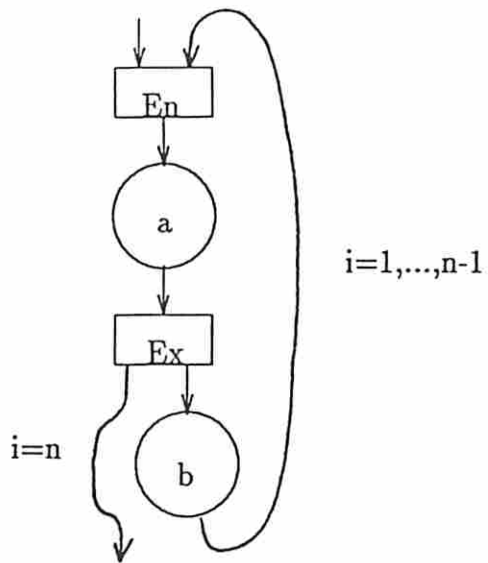
The analogy between loops and conditional paths is plain. At the top of the loop, values are either entering the loop for the first time or are the results from a previous loop; this selection is analogous to a conditional *join*. At the bottom of the loop, values are either propagated to the next operation or fed back to the top of the loop which is the same as a conditional *distribute*. Hence, one can substitute *join* for loop entrance and *distribute* for loop exit as they effectively represent the same operations. Since this leaves a mismatched conditional branch (a join precedes a distribute), the entrance (join) operation is moved *after* the exit (distribute) operation as shown in Figure 2.15b. This is only a modification to the dataflow graph; since no operations have been transposed, the order of operations is retained. Furthermore, since the a^* loop behavior is determined by the controller - which will still execute a one or more times - the behavior is unchanged. Note that the feedback register cuts lie on either end of the new subgraph. This would be the same physical register as values produced at the bottom of the loop are then implicitly available at the top of the loop.

Since the behavior in both branches of the **Ex**/**En** conditional is identical (no-op), in this particular example it can be collapsed into a single edge in the dataflow graph. Execution differences and register clocking are handled by the controller.

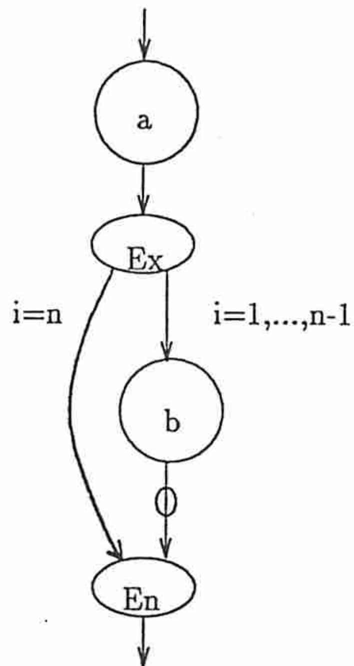
Transformation of a more complicated loop is shown in Figure 2.16a. The behavior is written $a(ba)^*$; **En** is moved resulting in the dataflow graph of Figure 2.16b. As in the first example, values produced at **En** are implicitly fed back to the top of the loop via use of the same register prior to a and after **En**.

Loops which have their exit condition within a nested conditional path, such as that shown in Figure 2.17, are a special case. Clearly, the single transformation results in two intertwined conditional paths as depicted in Figure 2.18; such a graph cannot be colored in order to determine mutual exclusion properly.

The transformation can be completed properly by noting that distribute and join operations as well as entrance and exit operations are place holders and represent no physical behavior. Thus, the dataflow edge which bypasses the **J** operation can be rerouted through it. Behavior is retained by duplicating the



(a) Original dataflow



(b) Modified dataflow

Figure 2.16: Loop with mid-graph exit condition

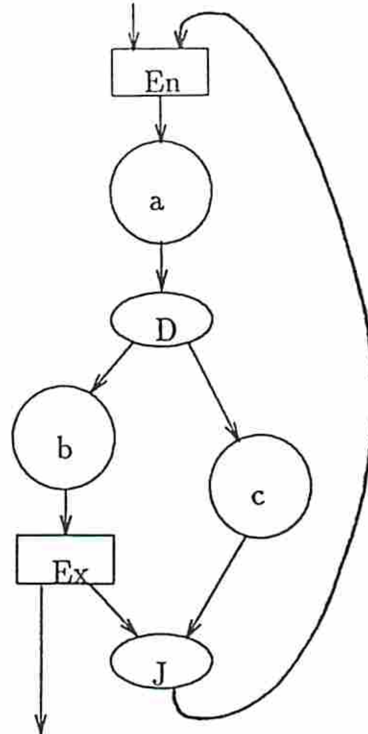


Figure 2.17: Loop with mutual exclusion and mid-graph exit

exit/entrance pair on either side of the **J** operation as shown in Figure 2.19. Similar to the first example where adjacent branches of a conditional path are identical, a single edge can be substituted resulting in the dataflow graph of Figure 2.20. The controller must track whether the loop is active or being terminated and operate the data path accordingly.

2.5.2 Transformation Algorithm

Figure 2.21 contains the algorithm for converting a cyclic dataflow description into an acyclic graph. The outermost repeat isolates each loop and writes the loop identifier and loop control parameters such as entrance, exit, and feedback edge to a loop control file. (This information is used for control synthesis and estimation; it could also be used for expected execution analysis.) The inner repeats break overlapping mutually exclusive branches (dist/join with Ex/En) and remove any useless Ex/En pairs. Any new entrance/exit pairs are written to the control file with the loop identifier. The list of loop transformations

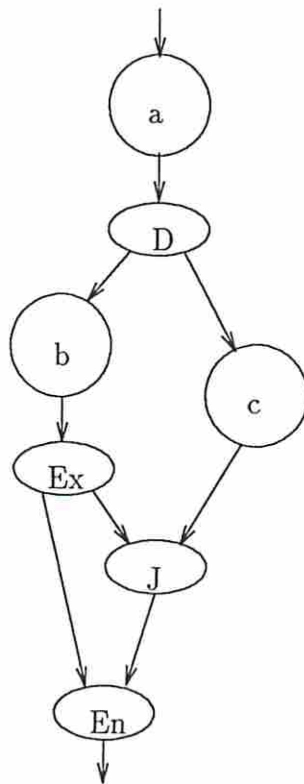


Figure 2.18: Loop after simple transformation

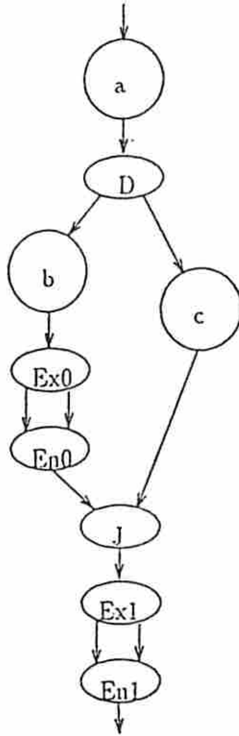


Figure 2.19: Disconnection of mutually exclusive branches

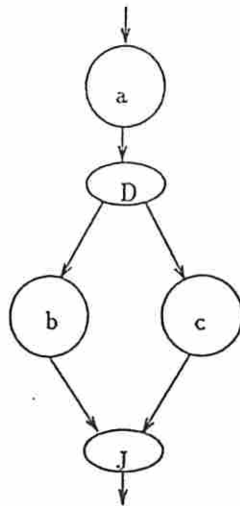


Figure 2.20: Completed loop transformation

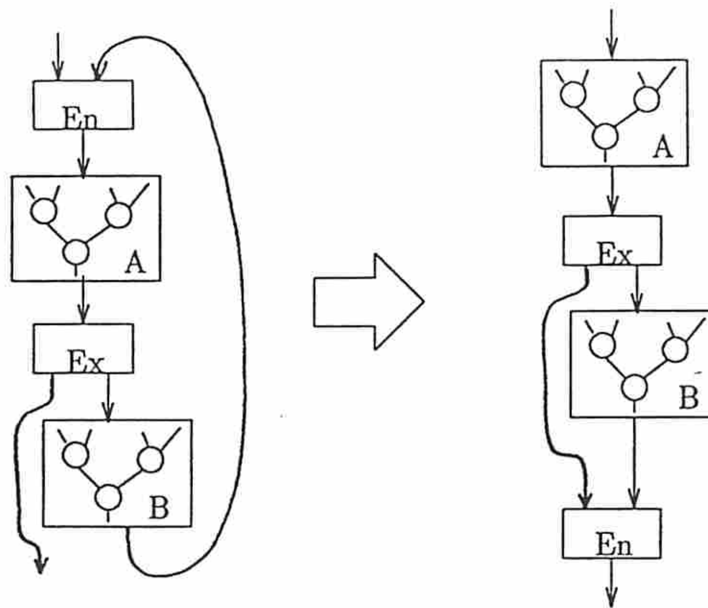
```
procedure break_loop(G)
begin
  repeat
    Find a feedback edge. If none, exit repeat.
    Mark feedback and loop entrance edges.
    Find all matching loop exit edges. If more than one unique edge, exit with error.
    Write loop id and entrance, exit, and feedback edges to loop control file.
    Apply transformation of Figure 2.22a.
    repeat
      Find overlapping dist/join-entrance/exit pairs. If none, exit repeat.
      Apply transformation of Figure 2.22b to remove overlap.
      Write entrance/exit pair and loop id to control file.
    end repeat
    repeat
      Find exit/entrance pair with matching no-op edges. If none, exit repeat.
      Apply transformation of Figure 2.22c.
      Write entrance/exit pair and loop id to control file.
    end repeat
  end repeat
end
```

Figure 2.21: Procedure for Transforming Cyclic into Acyclic Dataflow

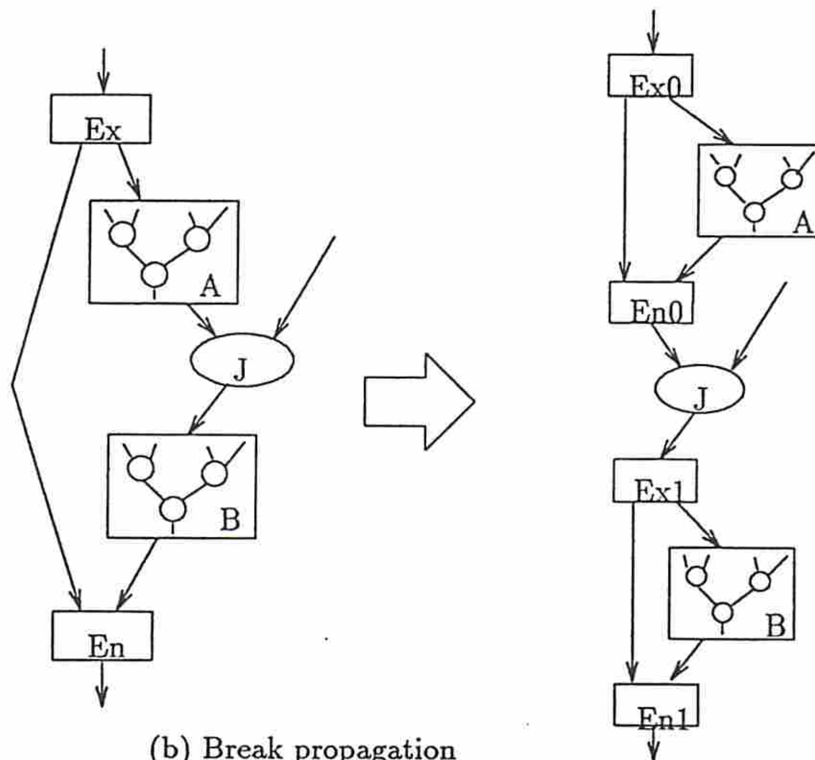
applicable is shown in Figures 2.22a through c. The transformation shown in Figure 2.22b is applied recursively until all distribute/join and entrance/exit dataflow subgraphs are "untwined".

With the transformations complete, any acyclic data path synthesis program can process the graph. A control path synthesis program or estimation tool can utilize the loop information to operate the hardware properly as it can be given all information regarding which conditionals are part of a specific loop.

Although data path synthesis of cyclic graphs is possible after performing the loop transformations to make them acyclic, good designs are not necessarily produced by the synthesis tool. This is an unfortunate artifact of any acyclic transformation of loops; from the view of data path synthesis, every operation in the graph is only executed once. As a result, hardware and register allocation

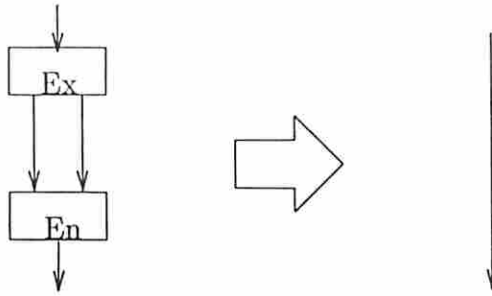


(a) Breaking the loop



(b) Break propagation

Figure 2.22: Loop Transformations



(c) Break compaction

Figure 2.22: Loop Transformations (cont.)

may be non-optimal and timing estimates for the graph are minimal rather than typical values.

A method which reduces the adverse effect of loop transformations is unrolling the loop into two or more iterations prior to synthesis. Unrolling of simple graphs merely entails replicating the acyclic loop region. However, for more complex graphs having exit conditions inside mutually exclusive branches, replication of the subgraph being iterated occurs along the feedback edge as indicated by the ellipse-marked edge in Figure 2.16b.

As the loop is unrolled and designs evaluated, register and hardware allocation quickly fall into place, e.g. registers appear at the loop entrance or exit and feedback edges, while hardware is shared between loops. By increasing the number of unrolled loops until little change is detected in the loop hardware allocation, the best hardware allocation and register placement can be determined for the loop.

The loop transformation software is a separate package for preprocessing a potentially cyclic graph; the output is usable by either the non-pipeline (**MAHA**) or pipeline (**Sehwa**) synthesis packages for the data path. In addition, the control area estimation package utilizes the loop control information generated. The effect of loop transformation was determined experimentally by running **MAHA** and analyzing the results manually. This could be integrated as an automatic extension to the transformation software in the future.

2.5.3 Runtime Analysis for Loop Transformation

Each feedback edge in procedure *break_loop* can be found in $O(E)$ time where E is the number of edges in the graph. N is the number of nodes. Backtracking to find the loop entrance and exit edges may take an additional $O(E)$ steps. Each isolated subgraph which comprises the loop has N' nodes and E' edges where $N' < N$ and $E' < E$, when the maximum number of exit/entrance and dist/join pairs is one per node. From Equation 2.4.8, a worst case conversion time of $O(N^3)$ occurs for a graph entirely composed of conditional operations.

2.6 Examples and Synthesis Results

A group of example data flow graphs which have varying degrees of parallelism, sharable operators, conditional paths, and operator complexity was selected for validating **MAHA**. Some of these graphs are small enough that the results generated by **MAHA** could be verified using exhaustive search. For the remainder, other tools were used to search for good designs. (Samples of **MAHA** program input and output can be found in Appendix B.)

The first example is a simple one to demonstrate the power of the enhanced version of **MAHA**, as shown in Figure 2.23. Each operation produces either the sum or product of its inputs and is 16-bits wide for both input and output. (For clarity, some input edges external to the figure are not shown.) Clearly, the most expensive (and fastest) design of Figure 2.23 would require the purchase of five **add** and three **mul** operators; the cheapest would be the purchase of a single **mul** and **add**. Table 2.1 shows the results from the most parallel to the most serial implementation which are individually shown in Figures 2.24 through 2.26. Only the non-inferior designs are listed. (Design A is inferior to design B if A has an area and time greater than or equal to that of B, excepting where both parameter pairs are equal.) Time and area are in nanoseconds and square mils, respectively, and area does not include wiring. Modules used are based upon the RCA 3-micron CADDAS library with parameters listed in Table 2.2. [2.1].

[2.1]The addition delays in this module library are somewhat pessimistic due to a historical error in reading a graph, and should not be taken as representative of actual modules.

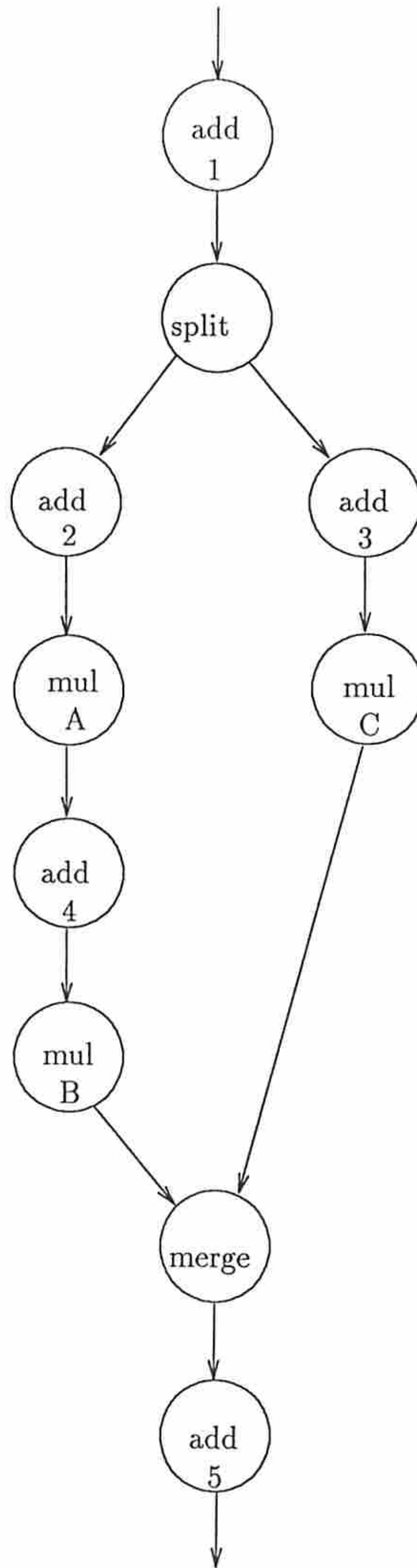


Figure 2.23: Demonstration data flow Graph

Table 2.1: MAHA Results for Simple Example

Time Partitions	Basic		With reg		With reg + mux	
	Time	Area	Time	Area	Time	Area
1	2110	168000	2115	168512	2115	168512
2	2110	110600	2120	112136	2128	113288
3	2145	106400	2160	108960	2184	110688
6	2250	53200	2280	57808	2328	60688

Table 2.2: Modules Used for Synthesis

Type	Bitwidth	Area (mil^2)	Prop. Delay (ns)
add	16	4200	340
sub	16	4200	340
mul	16	49000	375
cmp	16	4200	340
and	2	5	3
shift	16	32	2
inv	1	2	2
bufpad	1	26	4
reg	1	32	5
mux	1	18	4

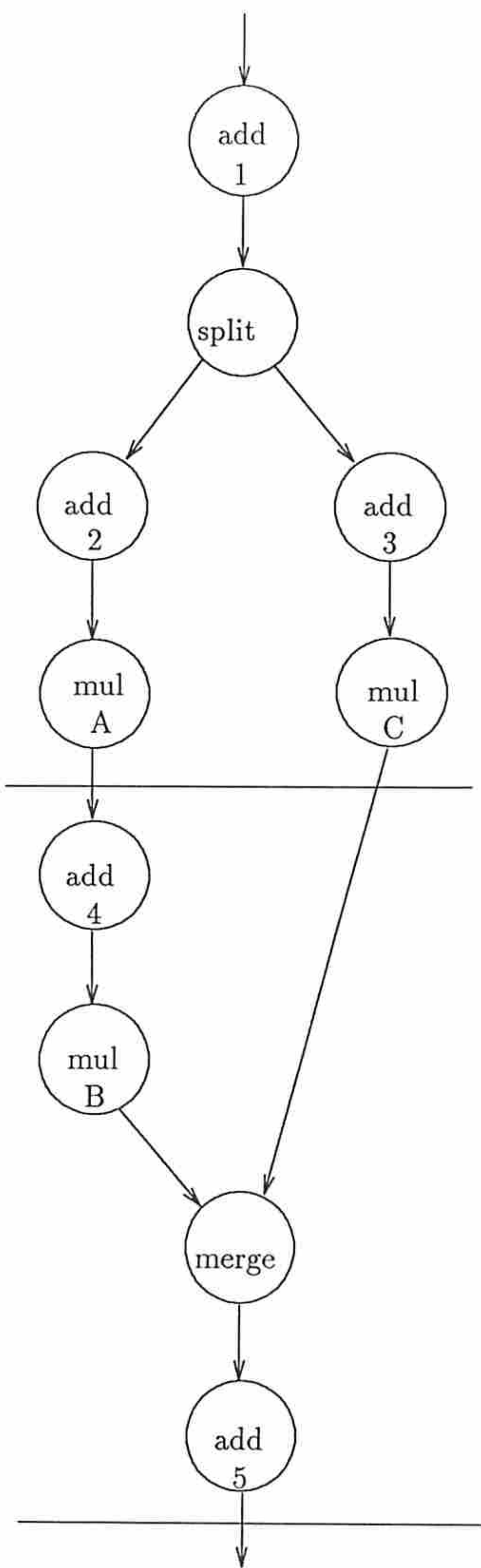


Figure 2.24: MAHA Results for 2-stage Design

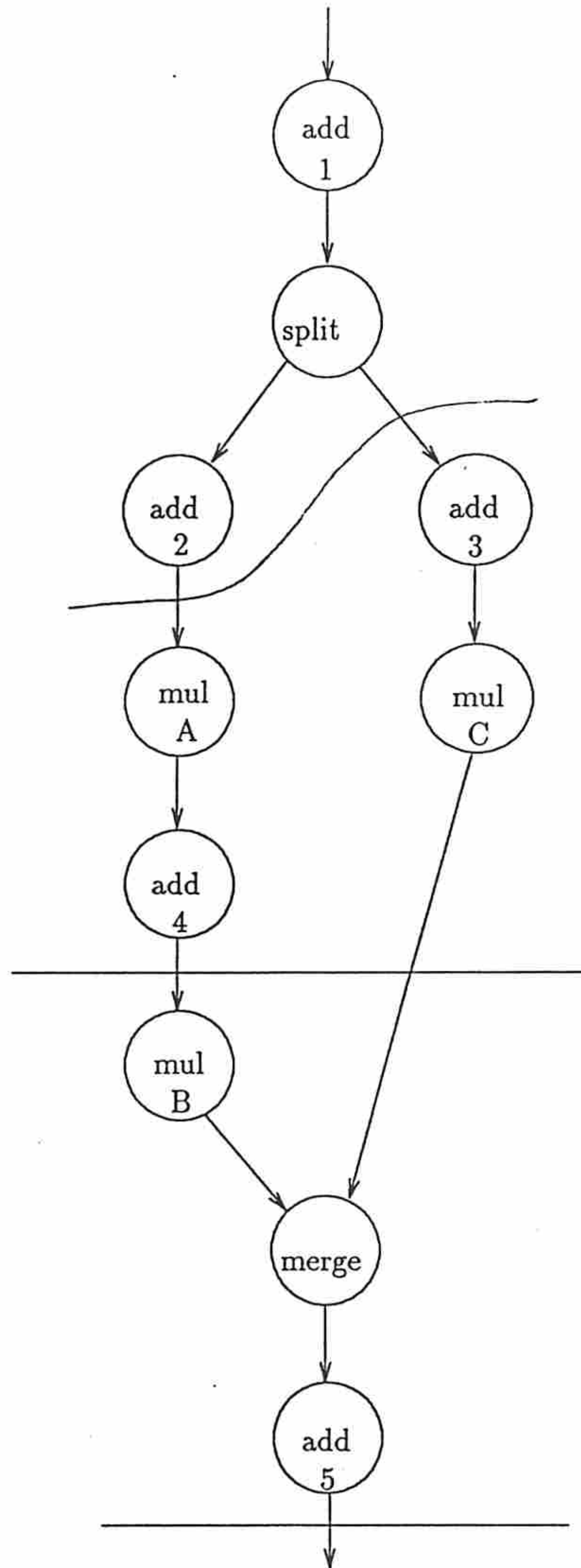


Figure 2.25: MAHA Results for 3-stage Design

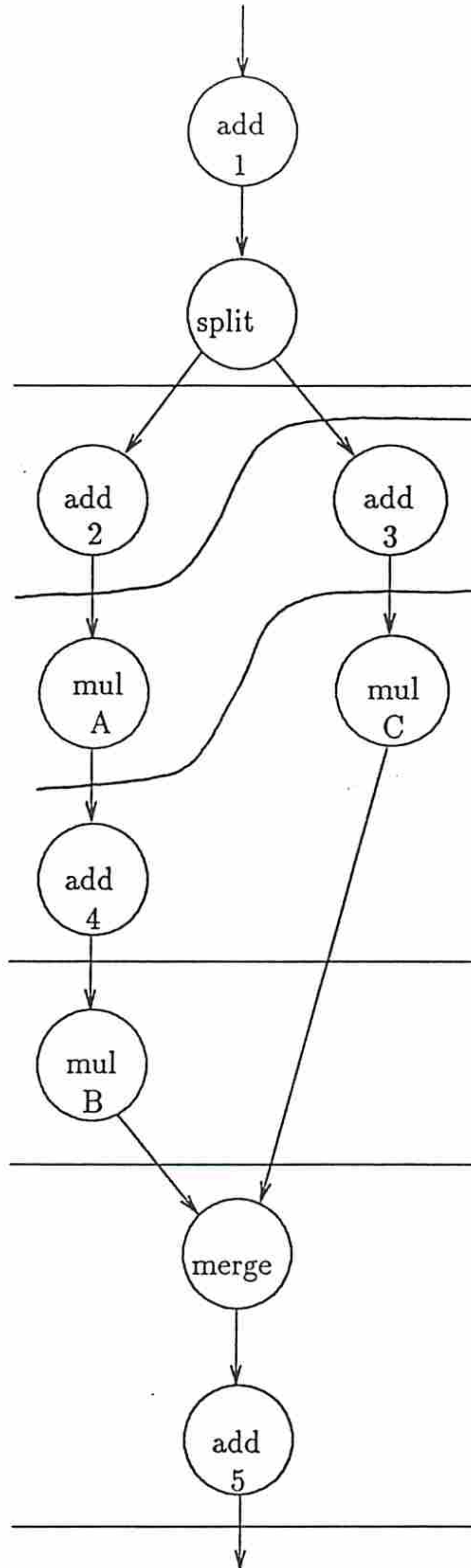


Figure 2.26: MAHA Results for Cheapest Design

Table 2.3: Sehwa Results for Simple Example

Basic				With reg			
Stages	Init. Intrvl	Time	Area	Stages	Init. Intrvl	Time	Area
6	1	2250	168000	3	1	2160	170560
10	5	3750	53200	7	5	2660	59856

In the table, **Basic** results are for operator scheduling and allocation only. When **MAHA** is instructed to include registers (**With reg**), a fixed delay of 5 nanoseconds is added per clock cycle. For the fastest design, a single 16-bit register is needed after *add5* which impacts the circuit area. Finally, when **MAHA** allocates both registers and multiplexers, circuit time and area is impacted further.

The results for the first example were verified manually; however, such a process is tedious and the possibility for error exists. Fortunately, several alternative tools are available for verifying the results: **Sehwa**, **optsyn**, and **randsyn**. **Sehwa** is a pipeline synthesis program written by Park as part of his dissertation on pipeline synthesis [PP86]. Results produced by **Sehwa** are shown in Table 2.3; "Time" in this table represents total time from top to bottom as opposed to initiation interval. **Sehwa** does not handle registers in the same fashion as **MAHA** due to the pipelined architecture. In addition, multiplexers are not addressed in **Sehwa**, so a comparison against **MAHA** is restricted to the basic case (no registers or multiplexers).

Since the fastest design implies a 1:1 mapping of operators onto operations, results from the two synthesis programs have identical area in the basic case. However, the times are different due to the difference between pipelined and non-pipelined architectures. The *overall* input-to-output delay for the fastest pipelined design is often slower than the comparable non-pipelined design due to the penalty of partitions being equal size even though operator times vary; however, *throughput* of the pipelined design is far better than the non-pipelined result. In the example of Figure 2.23, the *first* output value takes 2250 ns and 2110 ns to compute for the pipelined and non-pipelined designs, respectively. This 2110 ns value is fixed for any output values in the non-pipelined design.

Conversely, the pipelined design produces a new result each 375 ns (after the initial 2250 ns delay) until the pipe is flushed. The fastest time computed by **MAHA** should *never* be slower than the fastest pipelined time, which provides a comparison point.

Converse to the fastest design, only the area is identical between **MAHA** and **Sehwa** results for the cheapest design. When the cheapest pipelined and non-pipelined designs have the same number of partitions, the time computed should also match. The cheapest design of Figure 2.23 is depicted in Figures 2.26 and 2.27 as produced by **MAHA** and **Sehwa**, respectively. For the basic design, the areas are the same.

A time comparison between **MAHA** and **Sehwa** is more difficult for the cheapest design. The time shown in the table is **Sehwa**'s begin-to-end delay for a single dataset and not the initiation interval; hence, it is possible for **MAHA** to produce a better overall time than **Sehwa**. Despite these differences, **Sehwa** is an effective tool for comparing both the cheapest and fastest designs in many cases. Table 2.4 lists the results for the data flow graphs shown in Figures 2.23 through 2.35. Only the basic module area is used; registers and multiplexers are not considered.

Additional synthesis tools, **optsyn** and **randsyn**, were developed specifically to verify **MAHA** synthesis results. Both of these tools determine the operator scheduling given the *exact* hardware resources. **Optsyn** performs optimal synthesis by constructing every legal schedule permutation within the resource limitations. Unfortunately, computation time is exponential allowing only the smaller graphs to be verified with this procedure. **Randsyn** generates legal schedule permutations using a Monte-Carlo approach; here, the optimal design cannot be guaranteed. However, given a sufficient design sample size, the probable best design is indicated.

Given other synthesis utilities and some human designers by which to compare **MAHA**, a number of descriptions were attempted. Behavior ranged from irregular to regular structures and included conditional path examples; results are shown in Figures 2.28 through 2.35. Note that the human designers produced excellent designs in all but a few cases.

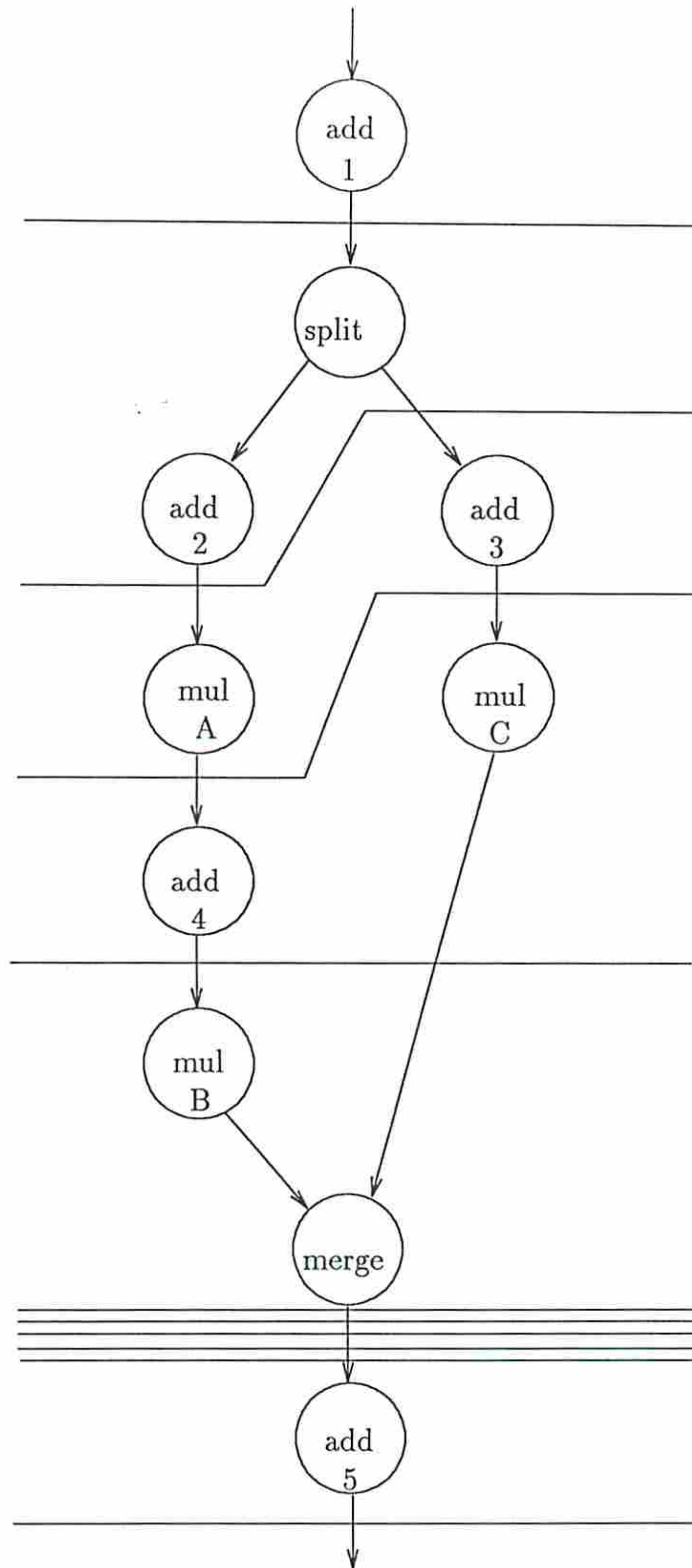


Figure 2.27: Sehwa Results for Cheapest Design

Table 2.4: Comparison of Synthesis Results: Sehwa

Description	Figure	Design	MAHA		Sehwa	
			Time	Area	Time	Area
Simple Example	2.23	Fastest	2110	168000	2250	168000
		Cheapest[1]	2250	53200	3750	53200
Parallel Example	2.28	Fastest	2720	67200	2720	67200
		Cheapest[3]	2720	8400	3060	8400
Temperature Controller	2.29	Fastest	1711	46343	2380	46343
		Cheapest	4152	12663	3740	12663
Multiplier Example	2.30[3]	Fastest	2075	217000	2250	217000
		Cheapest	2625	53200	2625	53200
FIR Filter	2.31[3]	Fastest	3095	455000	3375	455000
		Cheapest	5625	53200	5625	53200
AR Lattice Filter	2.32	Fastest	2825	834400	3000	834400
		Cheapest	6750	53200	7125	53200
Random Graph	2.33	Fastest	2075	207200	2250	207200
		Cheapest	6000	57400	6000	57400
Simple[4] Conditional	2.34[3]	Fastest	1360	16800	–	–
		Cheapest	1360	8400	–	–
Large Conditional	2.35[3]	Fastest	1700	46200	1700	46200
		Cheapest	2380	8400	2040	8400

[1] Begin-to-end data path time is slower for **Sehwa** in the cheapest case due to a number of inactive clock cycles added to achieve pipelining. Only six clock cycles (2250 ns) showed module activity.

[2] Time is slower for **Sehwa** in the cheapest case due to the additional step it added for a **join** node which has no delay.

[3] Figures 2.30, 2.31, 2.34 and 2.35 originally appeared in Park's dissertation [PP86].

[4] **Sehwa** failed to give an answer for this graph. Since the graph is small, results were verified manually.

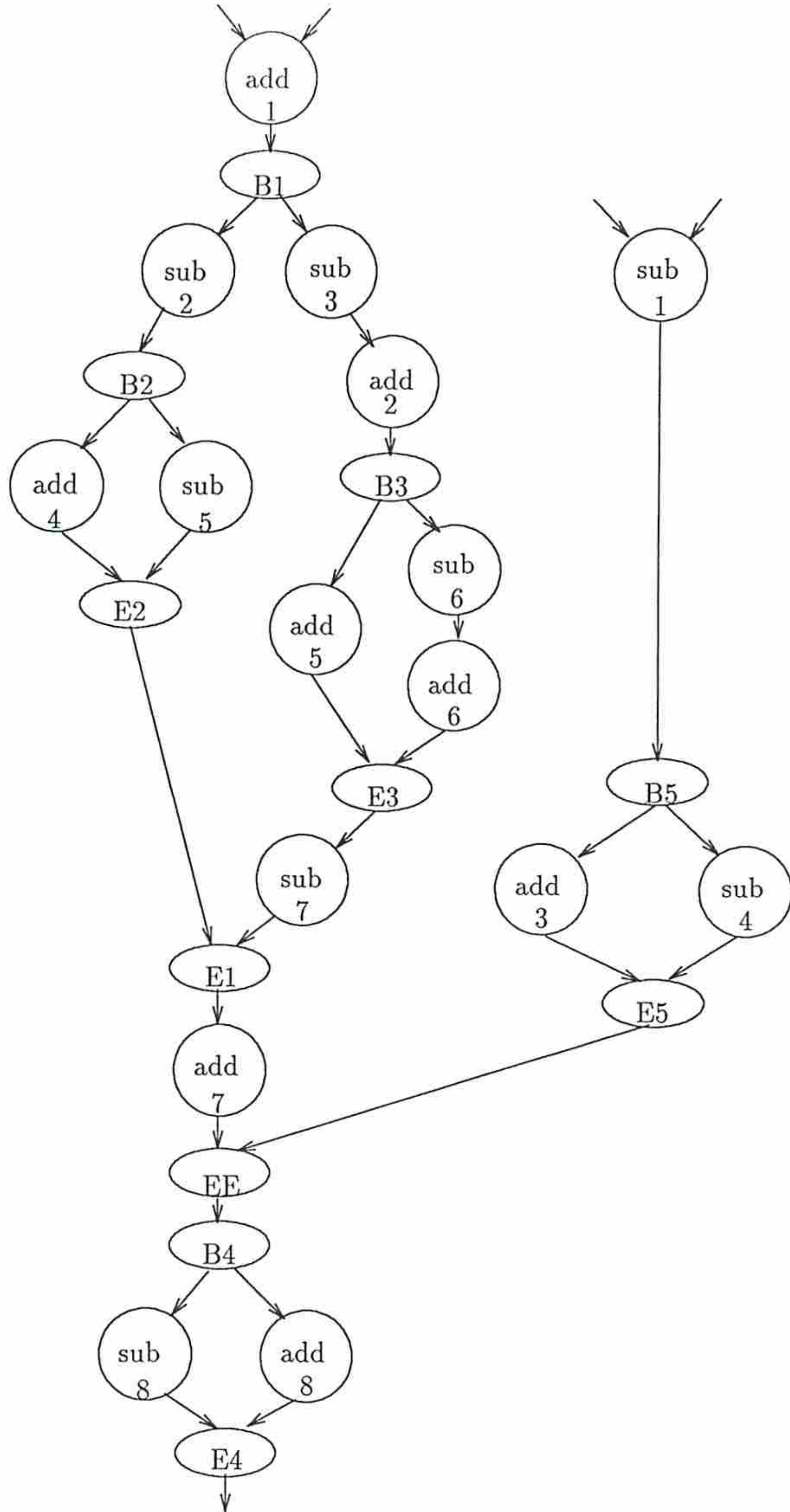


Figure 2.28: Parallel Example

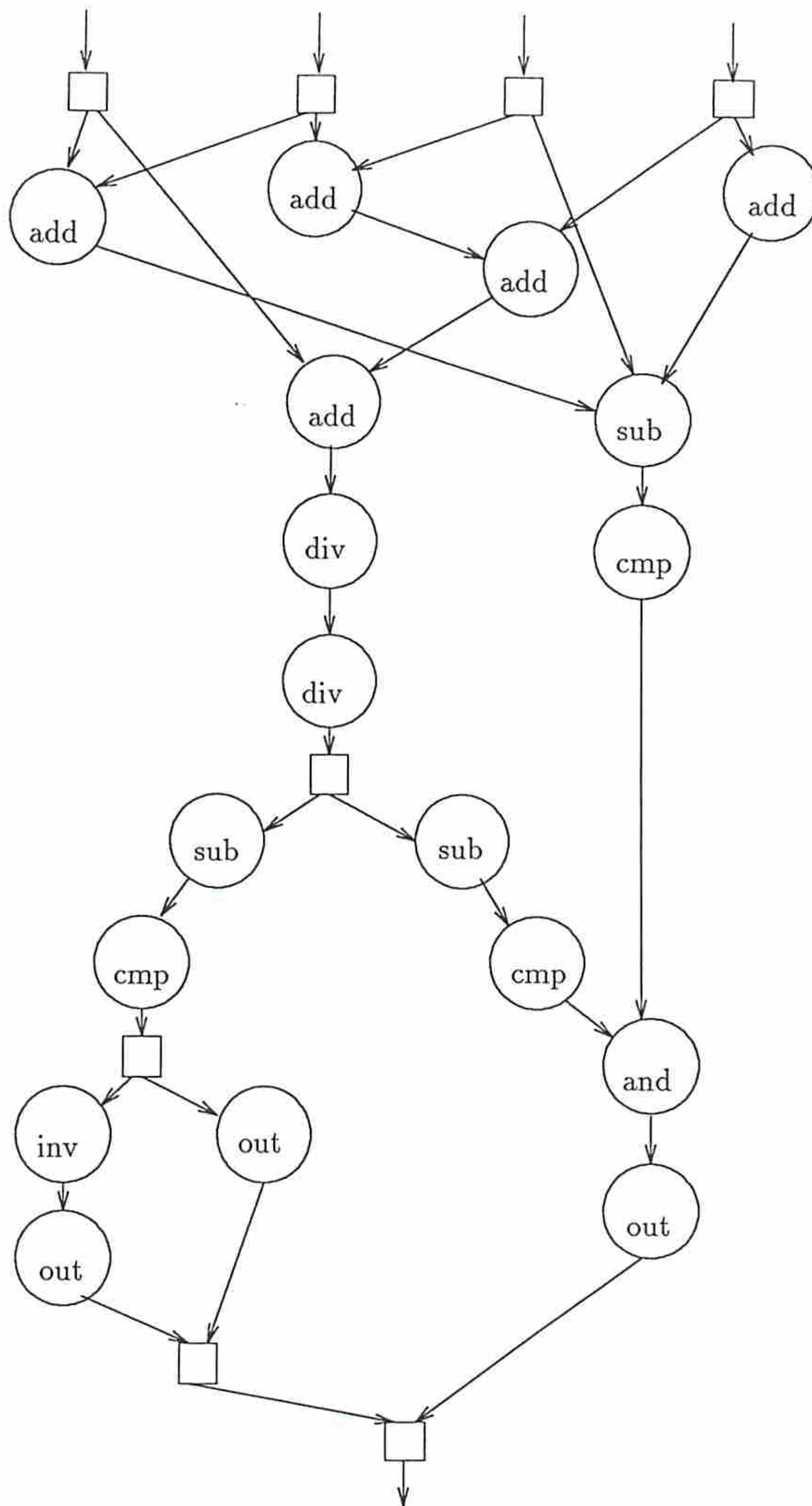


Figure 2.29: Temperature Controller

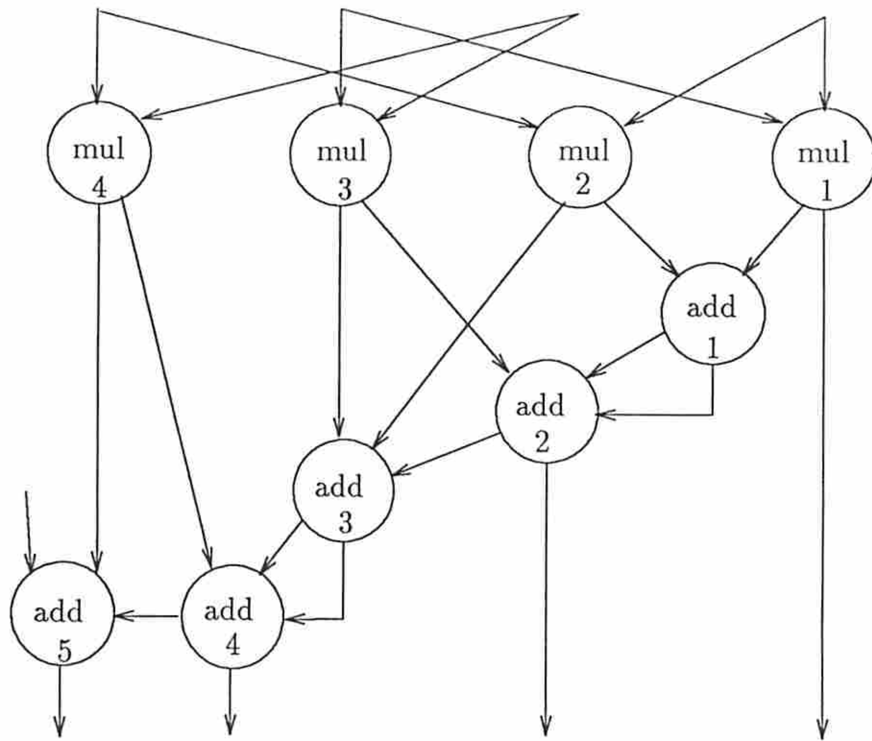


Figure 2.30: Multiplier Example

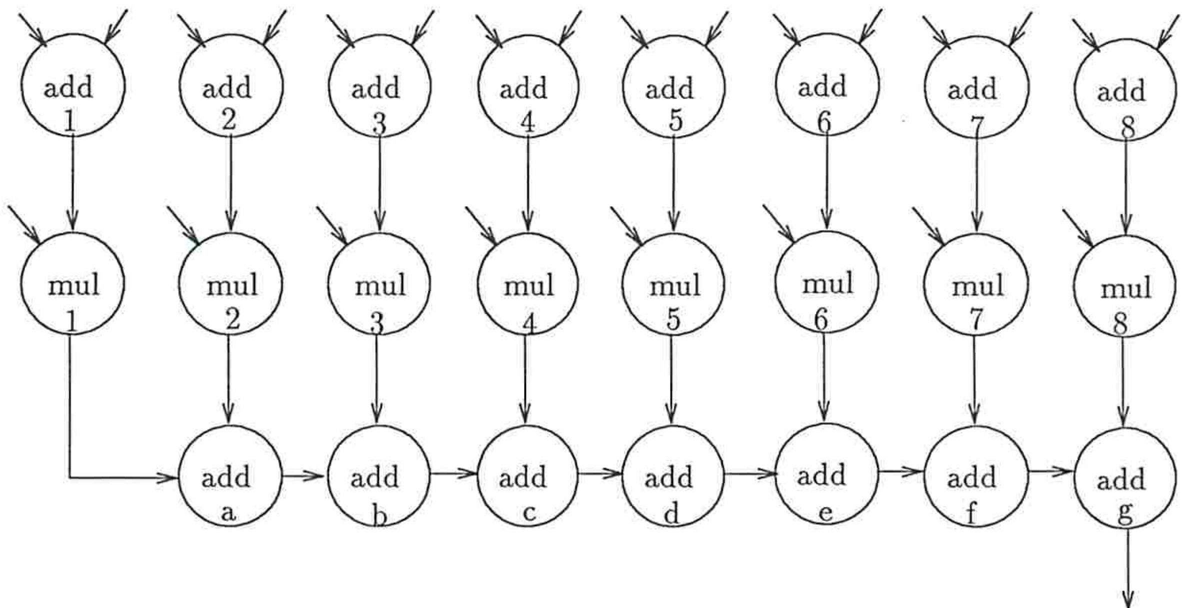


Figure 2.31: FIR Filter

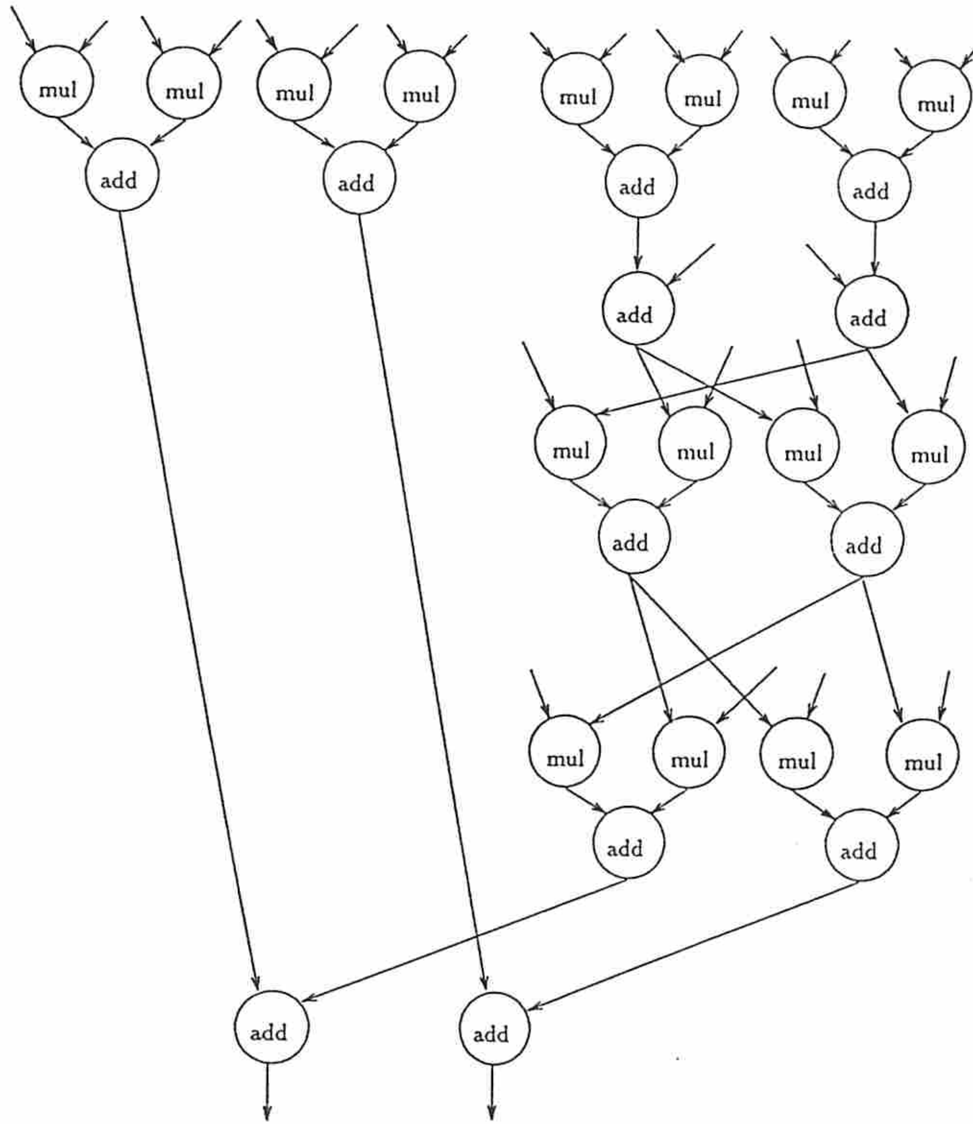


Figure 2.32: AR Lattice Filter

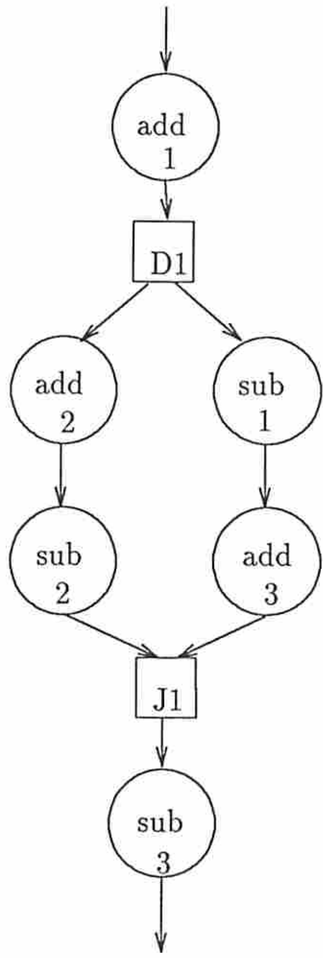


Figure 2.34: Simple Conditional

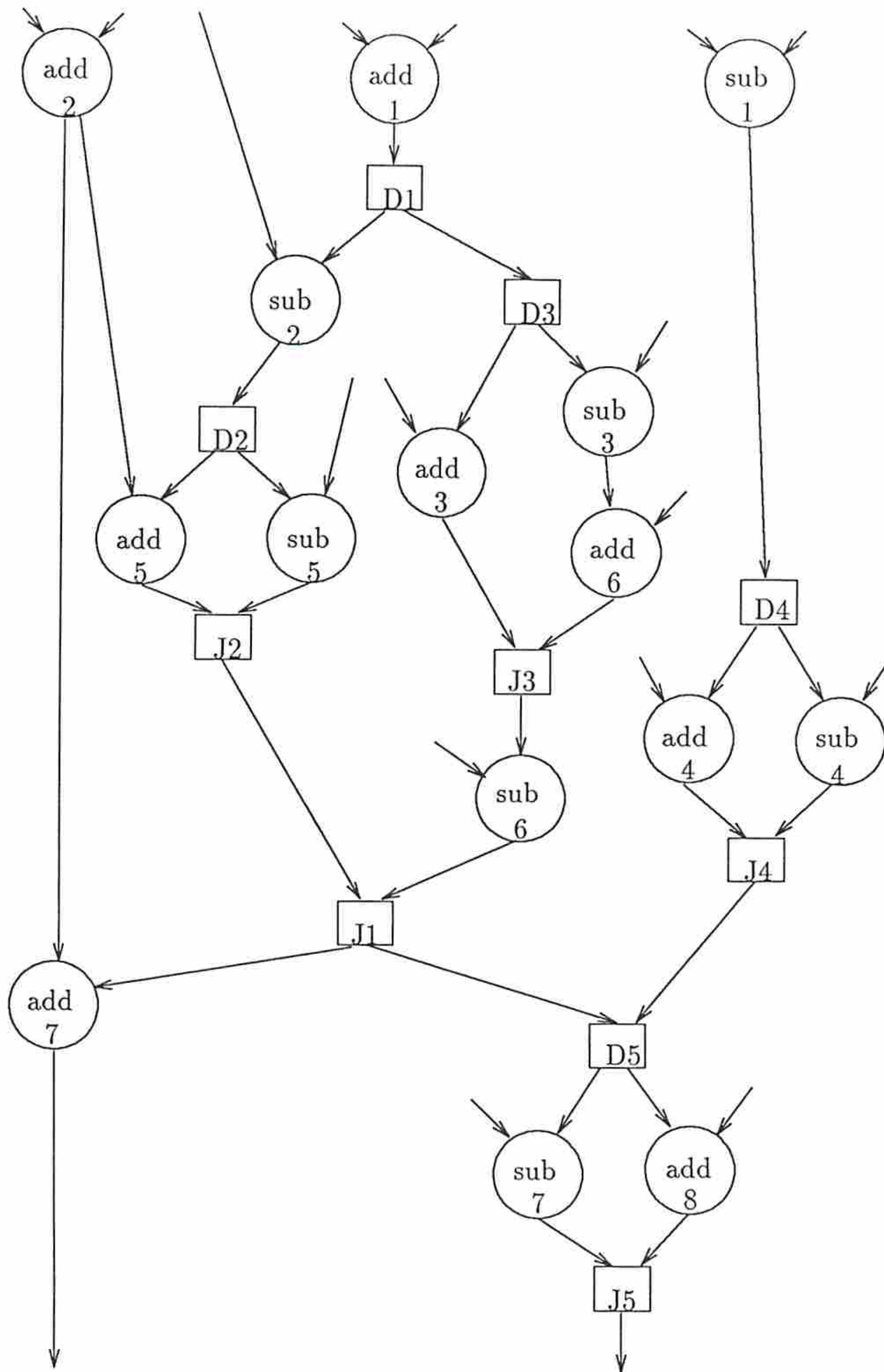


Figure 2.35: Large Conditional

As an additional check, any comparisons between **MAHA** and **optsyn** or **randsyn** synthesis results which differed were checked against an exhaustive search design from **Sehwa**. In essence, **Sehwa** was used to confirm the validity of **optsyn** and **randsyn**. Overall errors of the five graphs for which optimal and random synthesis designs were generated as compared to **MAHA** are less than 2%. Only 2 of the 27 total designs produced by **MAHA** using this module set were shown to be inferior. Errors are sufficiently small to confirm that **MAHA** is operating in the intended fashion; a more fundamental examination of the algorithm is included in the next subsection.

In addition to validation of **MAHA** for basic synthesis, verification of register and multiplexer extensions was performed manually for the cheapest designs. (The cheapest design was chosen since it has the lowest operator complexity giving a reasonable design for a human to optimize.) Table 2.5 summarizes the applicable examples. In all cases, the register and multiplexer values met the expected non-optimized costs (e.g. no sharing of register and multiplexer resources).

Since **MAHA** was developed, another utility called **MABAL** has appeared. This tool accepts the scheduling and allocation from **MAHA** and attaches the routing and storage hardware. Unlike **MAHA**, **MABAL** is capable of allocating both multiplexers and buses as well as registers. **MABAL** uses heuristics to minimize these values. Thus, the register and multiplexer results are better than **MAHA** and the usefulness of register and multiplexer allocation within **MAHA** is diminished. However, **MAHA** does provide a useful "first cut" at the completed data path, and is useful for non-pipelined designs with inner loops, which **MABAL** cannot handle.

Register and multiplexer cost can contribute a significant value for heavily serialized designs as shown in Table 2.5. A shift in both the area and time of the designs is noted. In fact, for the FIR filter, inclusion of register and multiplexer costs causes a *different* implementation to be the cheapest. Whereas the basic design used 15 stages, a design which included routing and storage used 8 stages. Introducing register and multiplexer effects favors the lower partition count in cost and also resulted in a lower time than one would expect, an observation also recognized by McFarland [McF87]. This example also illustrates the problem

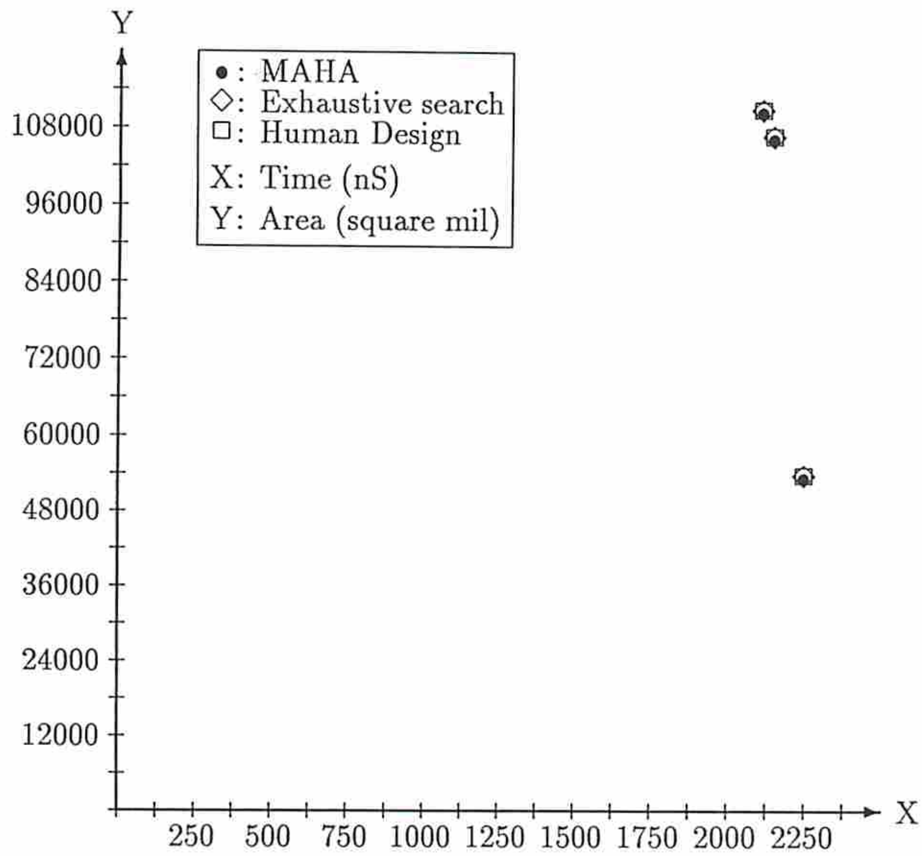


Figure 2.36: Synthesis Results for Small example

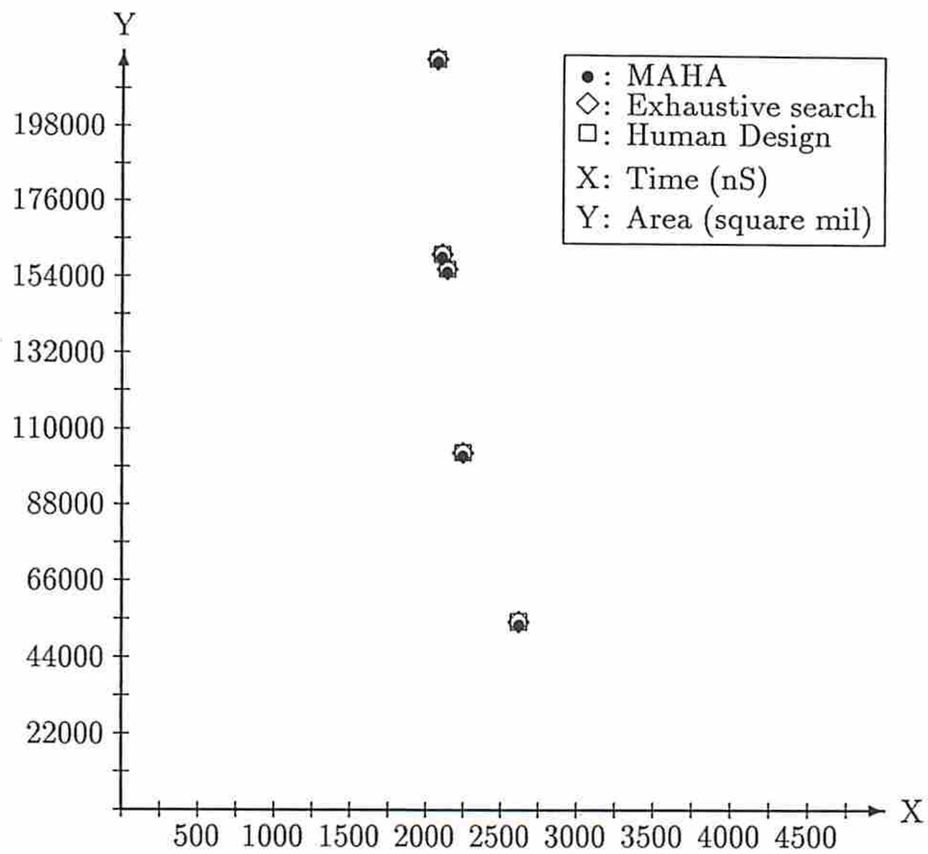


Figure 2.37: Synthesis Results for Multiplier

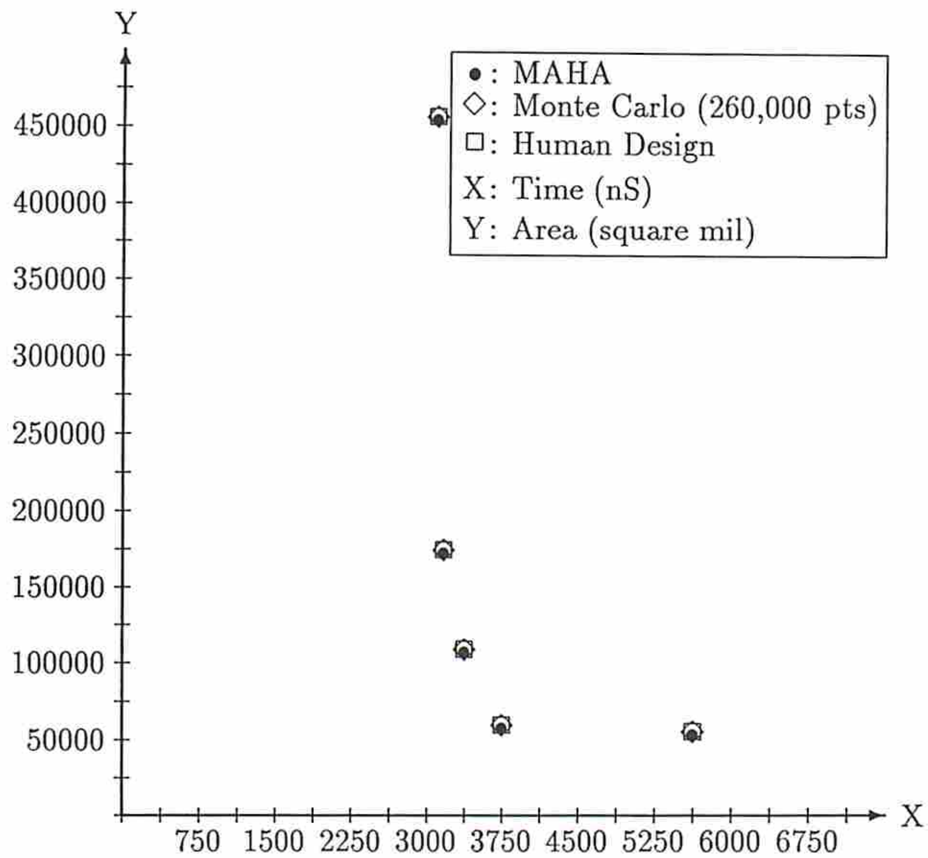


Figure 2.38: Synthesis Results for FIR Filter

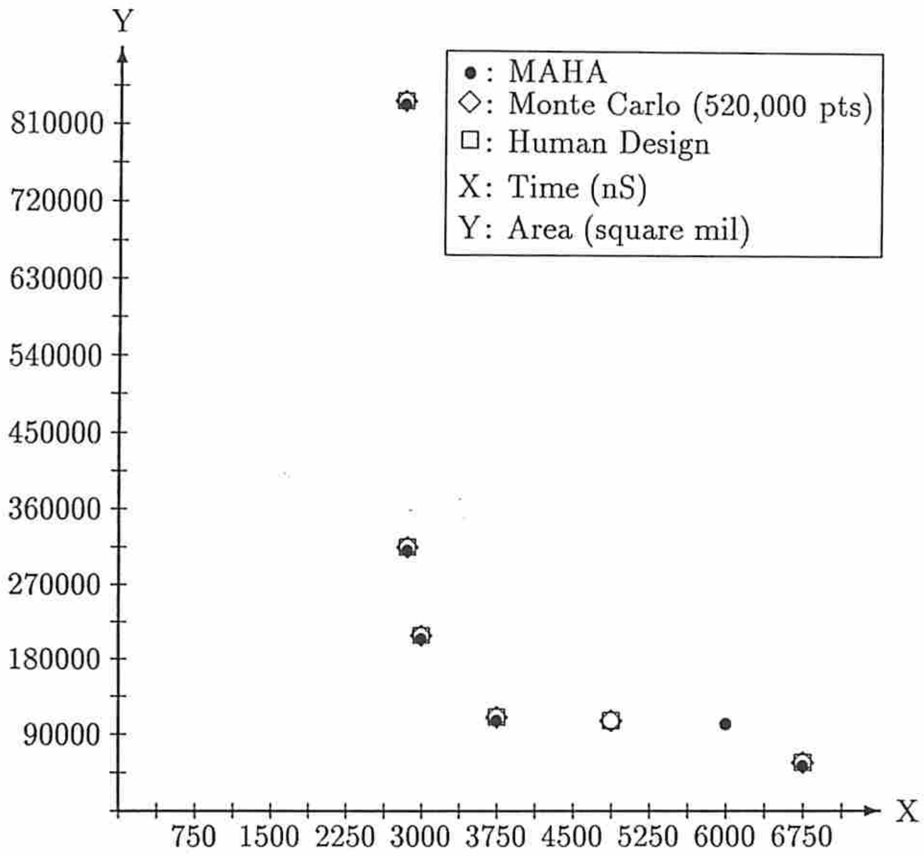


Figure 2.39: Synthesis Results for AR Lattice Filter

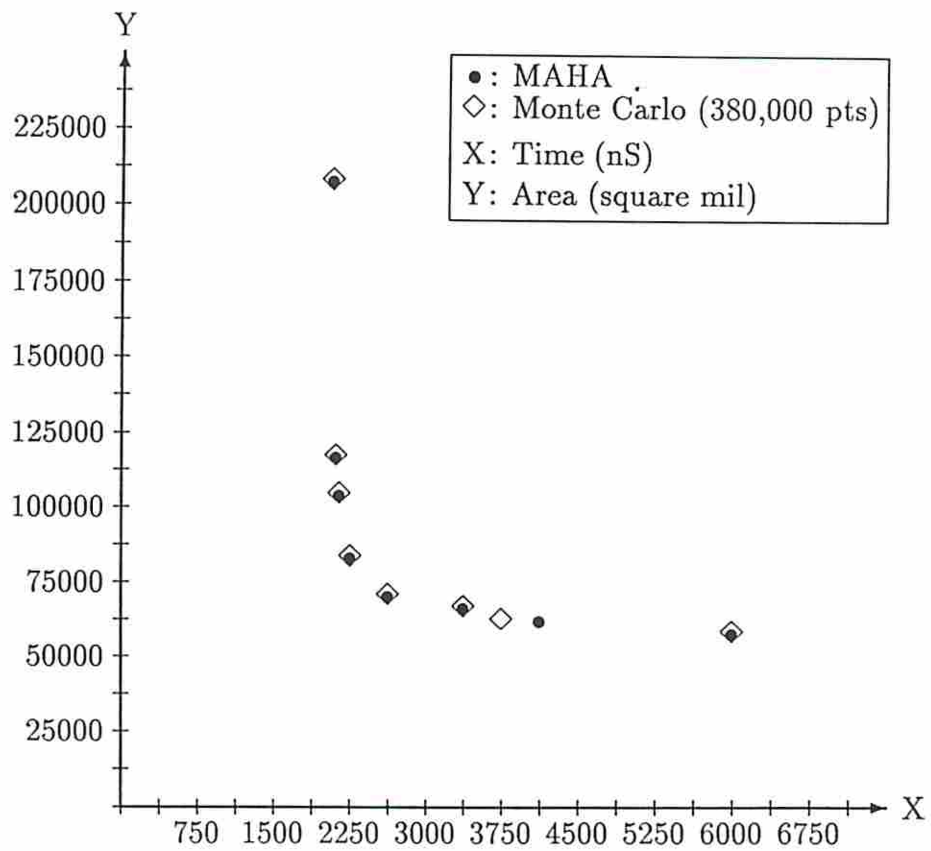


Figure 2.40: Synthesis Results for Random Graph

Table 2.5: Summary of Register and Multiplexer Area for Cheapest Design

Description	Figure	Synthesis	Basic	With reg.	With reg + mux
Simple Example	2.23	Time	2250	2280	2328
		Area	53200	57808	60688
Parallel Example	2.28	Time	2720	2760	2856
		Area	8400	16592	20624
Temperature Controller	2.29	Time	4152	4212	4308
		Area	12663	18039	23583
Multiplier	2.30	Time	2625	2660	2716
		Area	53200	57168	58680
FIR Filter	2.31	Time	5625	5760	5856
		Area	53200	65592	77112
AR Lattice Filter	2.32	Time	6750	6840	7128
		Area	53200	74704	85360
Random Graph	2.33	Time	6000	6080	6144
		Area	57400	80440	94840
Simple Conditional	2.34	Time	1360	1380	1412
		Area	8400	11472	12624
Large Conditional	2.35	Time	2380	2415	2499
		Area	8400	13520	15392

with data path synthesis: unless *all* steps of the synthesis process are done simultaneously, one may not achieve optimal or near-optimal results.

2.7 Limitations of MAHA

As a non-pipelined synthesis program, can **MAHA** be used in higher-level synthesis as a basic analysis tool? In order to answer this question, an understanding of the limitations of **MAHA** is necessary.

It has been shown in the previous section that **MAHA** produces designs that are not necessarily optimal. It is therefore useful to know the frequency and extent by which **MAHA** "misses" the optimal design. There are several factors which can affect the synthesis results including

- the size of the dataflow graph,
- the number of operation types in the graph, and

- the selected module set areas and delays.

First, specific errors noted in the previous section will be detailed. There are two cases where **MAHA** generates designs which are inferior to Monte Carlo and human designs.

One design where **MAHA** generates an inferior result is for the AR lattice filter where the resources are 2 multipliers and 1 adder. The actual partitioning results are shown in Figures 2.41 and 2.42. Note that although the clock cycle time is identical, the human design took three less partitions. This discrepancy is due to both the graph itself and the technique by which **MAHA** serializes it. The graph has sixteen unique but *identical* critical paths which are interwoven. Since **MAHA** only uses a single critical path, one is arbitrarily chosen. The remaining "non-"critical paths will be scheduled and allocated once the critical path is completed. This path will also be stretched in order to realize the more serialized designs.

It is in the stretching of this graph with the given resources that **MAHA** runs into a problem. With so many identical critical paths, the one chosen runs through the two adders which are marked **ADD** in Figure 2.41. Each of these particular **add** operations is pushed to a later time whenever a conflict occurs. These operations cannot be performed until all other competing **add** operations have completed, contributing a clock cycle.

This effect also extends to the surrounding **mul** operations. In particular, with four multipliers contending for two slots, only one delay should be needed. However, since **MAHA** inserts a critical path delay against the *first* non-critical path resource conflict, graph symmetry is not considered and delays are not inserted optimally. Furthermore, in future synthesis, that non-critical path operation is forced to occur in the same timestep as that delay node on the critical path. Each cluster bounded by the boxes shown is scheduled in one additional clock cycle each, giving the three clock cycle difference.

The other design encountered which is non-optimal is the random dataflow graph having 16 **add**, 10 **sub** and 2 **mul** operations. Resources consisted of one subtractor, one multiplier, and two adders. In this case, stretching the dataflow

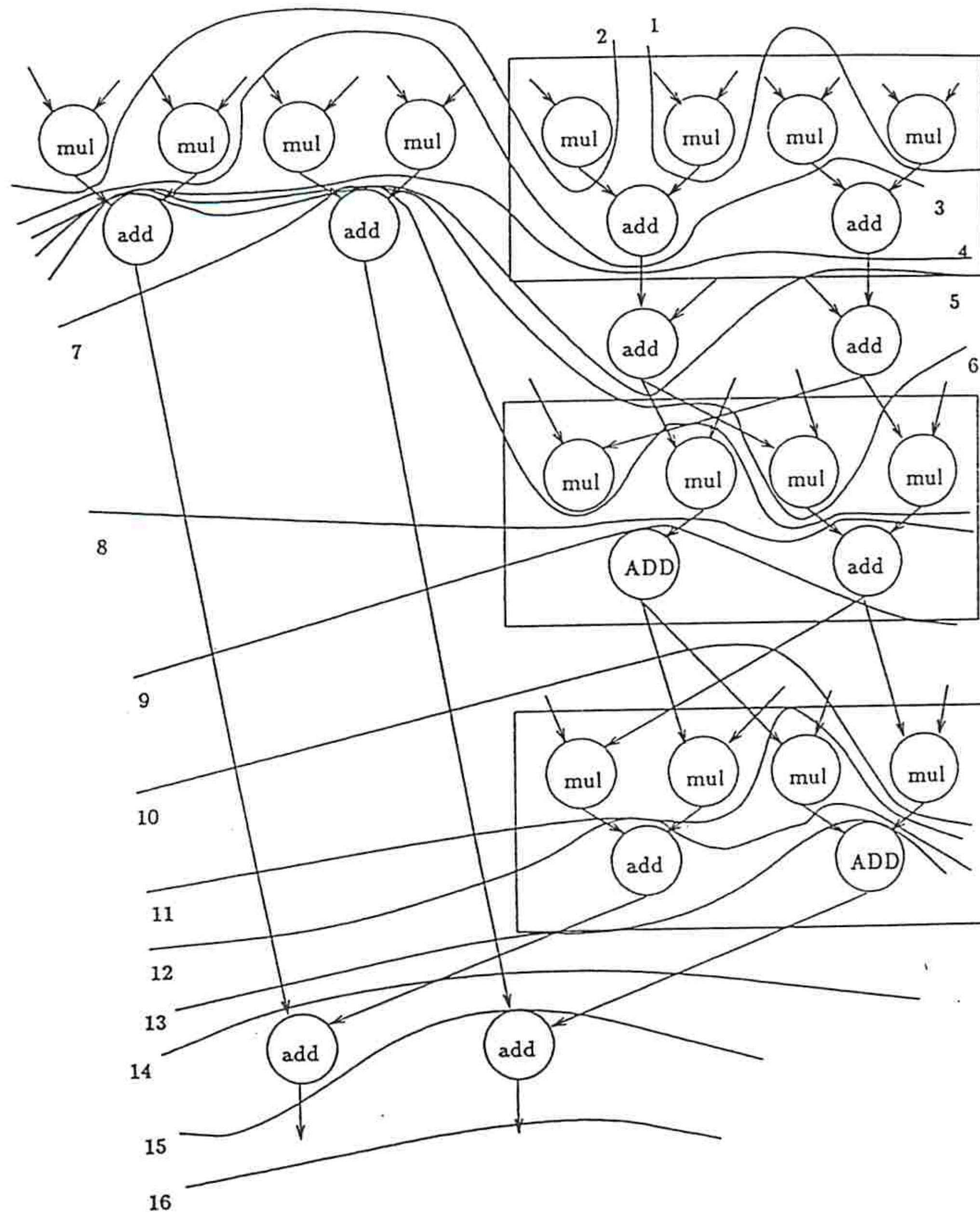


Figure 2.41: MAHA design of AR filter

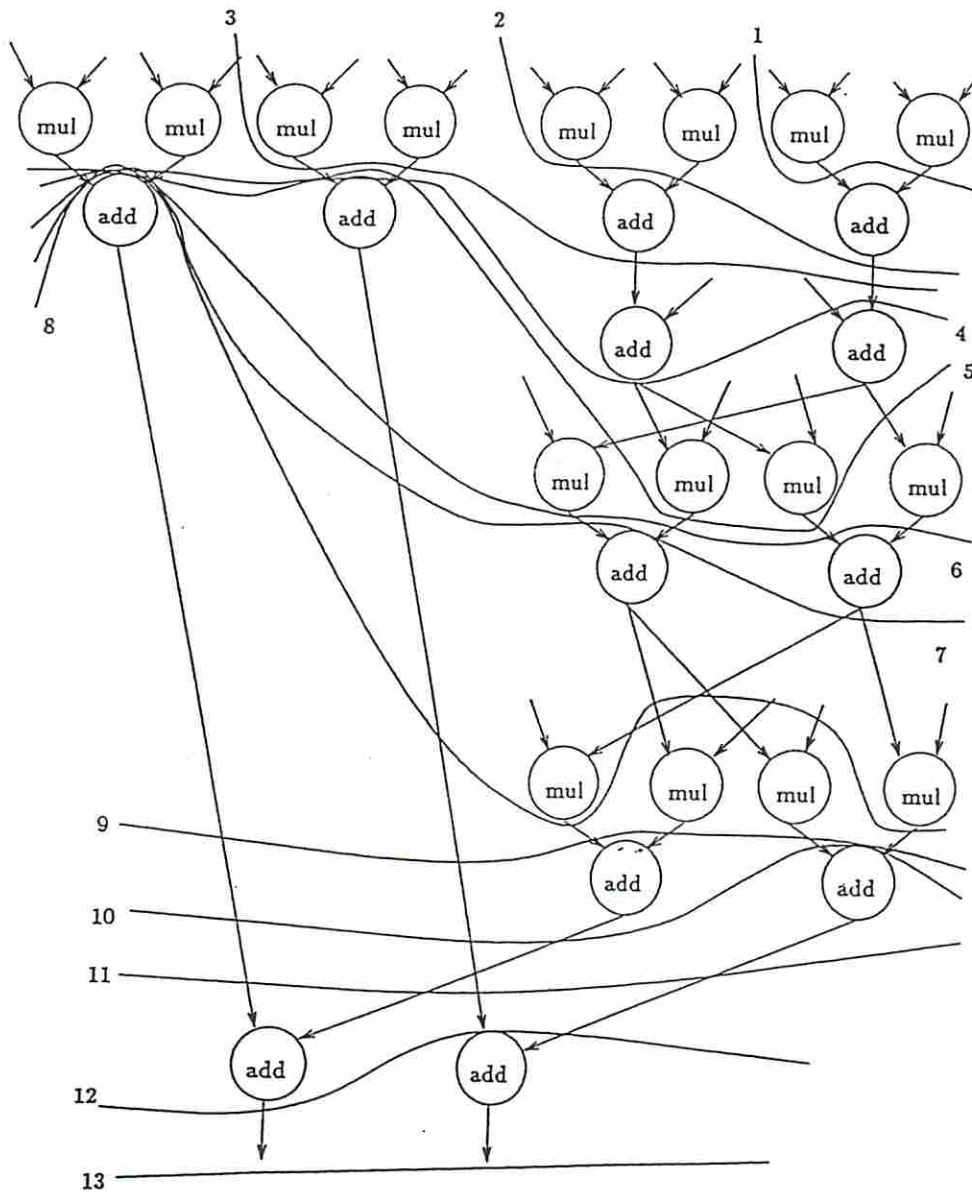


Figure 2.42: Human design of AR filter

graph resulted in a single stage where a subtractor was prevented from being used, causing an additional clock cycle to be required.

It should be noted that the AR lattice filter is probably a worst case example for non-pipelined design. Clearly, this highly symmetrical graph is best implemented using a pipelined architecture. However, both examples highlight one potential weakness of this algorithm. In stretching the dataflow graph to produce more serial designs, there is no understanding of shapes and layout of the graph. In all cases, a human designer could observe the pattern and exploit the symmetry. Graph analysis and pattern detection at a high level would improve the intelligence of the algorithm. However, given the complexity of such a task and the present state of **MAHA**, it is not clear such an activity would greatly improve the overall quality of the the synthesis results.

In general, synthesis errors on small dataflow graphs are greatly pronounced since a small change has a large impact. However, optimal (exhaustive search) techniques can be employed in such cases as needed without excessive computation time. Human designers also excel in this area. Conversely, large designs are difficult for a human designer to do well (given a limited time to produce a circuit) and exhaustive techniques are not feasible. Given the heuristic nature of synthesis, one would expect most designs to be non-optimal, but still acceptable. Unfortunately, criteria for what is acceptable is usually subjective due to external design factors such as processing costs, production run size, and the application environment.

The number of module types and their areas and delays also affect the synthesis process. A graph with a wide range of operation types may limit the flexibility of **MAHA**. However, if many of these operations are available in a single complex module such as an ALU, then sharing can be improved. This is a module-set tradeoff problem which can greatly impact the results. Also, in the previous section, one type of add and multiply were chosen. If a number of adders and multipliers are available, which set should be chosen? Clearly, this is a separate topic of research in which some progress has been made [Jai89].

Given the RCA 3um CADDAS library, several types of multipliers and adders were constructed; one set was shown in earlier. Additional modules are shown in

Table 2.6: Module Library of Adders and Multipliers

Type	Bitwidth	Area (mil^2)	Prop. Delay (ns)
add-f	16	4200	340
add-m	16	2880	530
add-s	16	1200	1510
sub-f	16	4200	340
sub-m	16	2880	530
sub-s	16	1200	1510
mul-f	16	49000	375
mul-m	16	9800	2950
mul-s	16	7100	7370

Table 2.6 [2.2]. Using the same dataflow graphs given in Section 2.6, other module sets were constructed and used by **MAHA** for synthesis. (No attempt was made to use two different modules of the same type in a single design.)

Since the programs **optsyn** and **randsyn** accept a quantized module set as input, a direct comparison between them and **MAHA** is possible. To simplify the analysis, every **MAHA** design point was distinguished as follows:

1. For each **MAHA** design point, the same module set and quantities were input into **optsyn/randsyn** to compare the time.

2. After completing step (1), the module set quantities were lowered to ascertain whether the same circuit time could be met using a smaller module set. The area difference, if any, was then compared.

With this approach, each design point produced by **MAHA** could be compared in both area and time against the verification utilities. Three module sets chosen by **SLIMOS** as being the most useful were used and are listed in Table 2.7 [JPP88]. These module sets were applied against a number of dataflow graphs to ascertain errors in both area and time as shown in the first two columns of Tables 2.8 and 2.9. (An elliptical wave filter shown in Figure 2.43 was included so that a comparison could be made with other synthesis tools.) **MAHA** was

[2.2]The addition delays in this module library are somewhat pessimistic due to a historical error in reading a graph, and should not be taken as representative of actual modules.

Table 2.7: Module Sets Evaluated by MAHA

Name	Hardware Modules		
fast	mul-f	add-f	sub-f
medium	mul-m	add-s	sub-s
slow	mul-s	add-s	sub-s

then executed; the total number non-inferior designs obtained is listed in the third column. The quantity of these designs which exhibited optimal area and time are shown in Tables 2.8 and 2.9, respectively. Optimal area means that no design of the same (or lower) time could be realized with the less hardware; optimal time means that no design of the same (or less) area could be constructed with a smaller circuit delay. From these results, the maximum and average error over all *non-optimal* designs is computed as well as the average error over all designs (both optimal and non-optimal).

At first glance, it might appear that the synthesis program is not producing good results for some descriptions. In particular, the elliptical wave filter **MAHA** designs diverge from the Monte Carlo results by more than a single module. Area differences are due to the heuristic nature of the scheduling algorithm. Unfortunately, time delay error is magnified in this dataflow graph, since time is the product of clock cycle time and the number of partitions. For example, the most serial elliptical filter design had 28 time steps. The worst case arises for 16 partitions when the clock is off by only 11%.

A comparison of the **MAHA** elliptical wave filter designs against other synthesis packages is plotted in Figure 2.44. Other tools include HAL [PK87], EMUCS [MPC88], SPAID [HE89], CATREE [GE88], and SPLICER [Pan88]. In this example, **MAHA** not only produced better designs when the same module sets are used, but it also explored a larger portion of the design space.

Another approach to determine the usefulness of **MAHA** is to analyze the data to determine *what* causes the synthesis errors. Table 2.10 contains a summary of the comparison ranked by graph size (the number of nodes in the dataflow graph) as compared to the time and area variance in *module* area and delay units. If the number of designs that are optimal in area and time are

Table 2.8: Comparison of MAHA results to Human/Random: **Area**[1]

Dataflow Graph	Module Set	Total Designs	Optimal Designs	Max. Err. (%)[2]	Ave. Err. (%)[3]	Overall Err. (%)
Parallel	fast	4	4	0.0 (0)	0.0	0.0
	medium	4	4	0.0 (0)	0.0	0.0
	slow	4	4	0.0 (0)	0.0	0.0
Multiplier	fast	5	5	0.0 (0)	0.0	0.0
	medium	5	5	0.0 (0)	0.0	0.0
	slow	6	5	20.0 (*)	20.0	3.3
FIR Filter	fast	5	5	0.0 (0)	0.0	0.0
	medium	8	7	42.0 (1)	42.0	5.3
	slow	10	8	15.5 (*)	12.3	2.5
AR Filter	fast	6	6	0.0 (0)	0.0	0.0
	medium	9	7	55.5 (1)	50.0	11.1
	slow	10	8	15.5 (*)	12.4	2.5
Random Gr.	fast	8	8	0.0 (0)	0.0	0.0
	medium	7	7	0.0 (0)	0.0	0.0
	slow	10	10	0.0 (0)	0.0	0.0
Simple Cond.	fast	2	2	0.0 (0)	0.0	0.0
	medium	2	2	0.0 (0)	0.0	0.0
	slow	2	2	0.0 (0)	0.0	0.0
Large Cond.	fast	2	2	0.0 (0)	0.0	0.0
	medium	2	2	0.0 (0)	0.0	0.0
	slow	2	2	0.0 (0)	0.0	0.0
Ellip. Filt.	fast	7	5	17.4 (*)	10.6	3.0
	medium	12	5	91.1 (4)	58.8	34.2
	slow	12	7	66.3 (1)	38.0	15.8

[1] A comparison of area between the designs is made where the times are identical.
[2] The number in parentheses is the number of extra modules purchased. A value of "*" means that the total number of modules is identical, but the individual module type counts are different.
[3] The average error is the average of all designs which did not have the optimal area.

Table 2.9: Comparison of MAHA to Human/Random: **Time**[1]

Dataflow Graph	Module Set	Total Designs	Optimal Designs	Max. Err. (%)[2]	Ave. Err. (%)[3]	Overall Err. (%)
Parallel	fast	4	4	0.0 (0)	0.0	0.0
	medium	4	4	0.0 (0)	0.0	0.0
	slow	4	4	0.0 (0)	0.0	0.0
Multiplier	fast	5	5	0.0 (0)	0.0	0.0
	medium	5	5	0.0 (0)	0.0	0.0
	slow	6	4	20.5 (*)	11.9	4.0
FIR Filter	fast	5	5	0.0 (0)	0.0	0.0
	medium	8	7	16.7 (1)	16.7	2.1
	slow	10	8	16.7 (1)	13.7	2.7
AR Filter	fast	6	4	23.1 (3)	16.5	5.5
	medium	9	5	25.0 (2)	15.9	8.8
	slow	10	5	20.5 (1)	18.4	9.2
Random Gr.	fast	8	7	10.0 (1)	10.0	1.3
	medium	7	6	10.0 (1)	10.0	1.4
	slow	10	9	2.0 (*)	2.0	0.2
Simple Cond.	fast	2	2	0.0 (0)	0.0	0.0
	medium	2	2	0.0 (0)	0.0	0.0
	slow	2	2	0.0 (0)	0.0	0.0
Large Cond.	fast	2	1	16.7 (1)	16.7	8.3
	medium	2	1	16.7 (1)	16.7	8.3
	slow	2	1	16.7 (1)	16.7	8.3
Ellip. Filt.	fast	7	4	7.1 (1)	5.7	2.4
	medium	12	4	44.4 (3)	27.4	18.3
	slow	12	4	26.9 (2)	16.3	10.9

[1] A comparison of time between the designs is made where the areas are identical.
[2] The number in parentheses is the *total* time difference expressed as the number of extra minimum clock delays. A value of "*" means that the delay is the less than the minimum clock (i.e. the faster module).
[3] The average error is the average of all designs which did not have the optimal time.

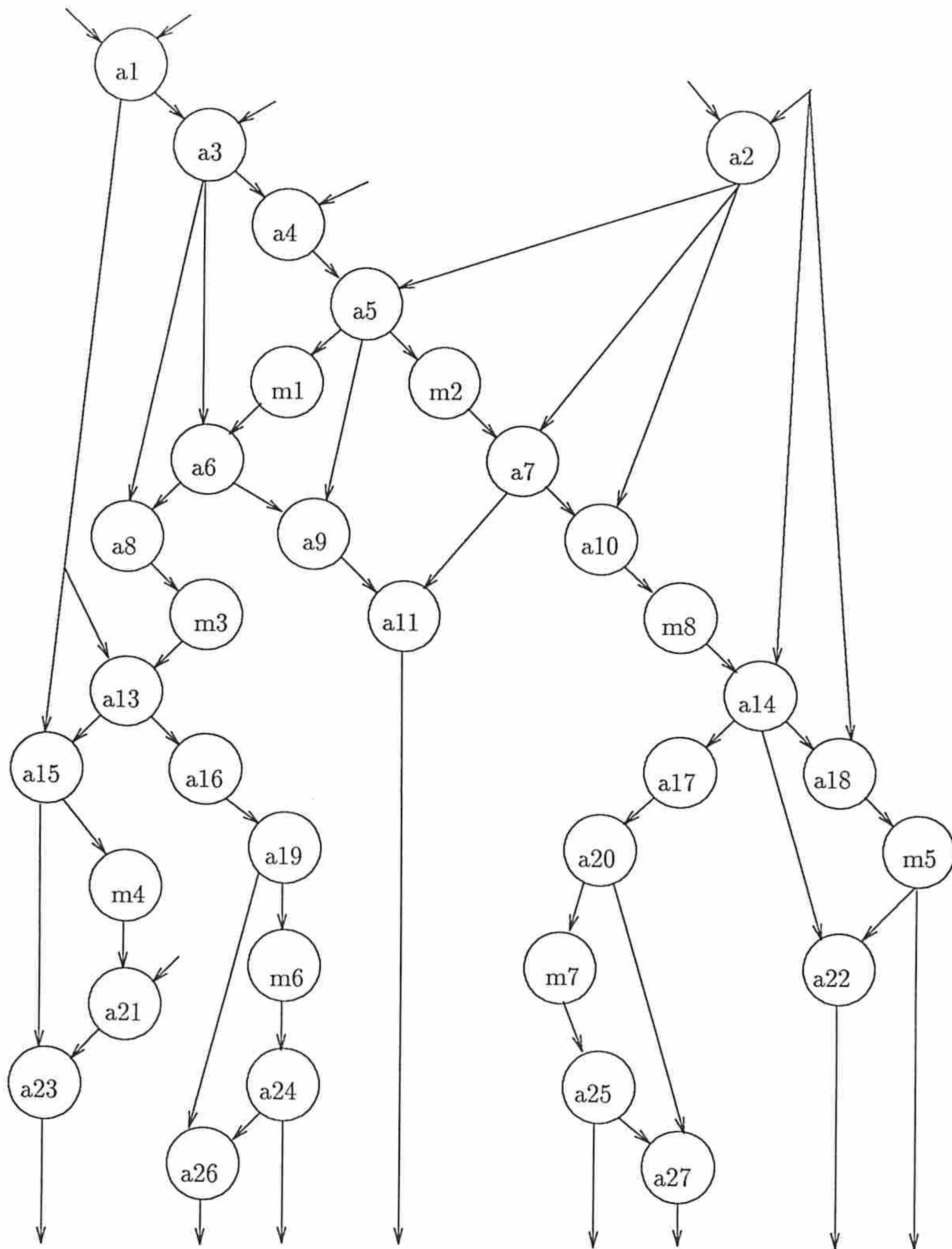


Figure 2.43: Elliptical Wave Filter

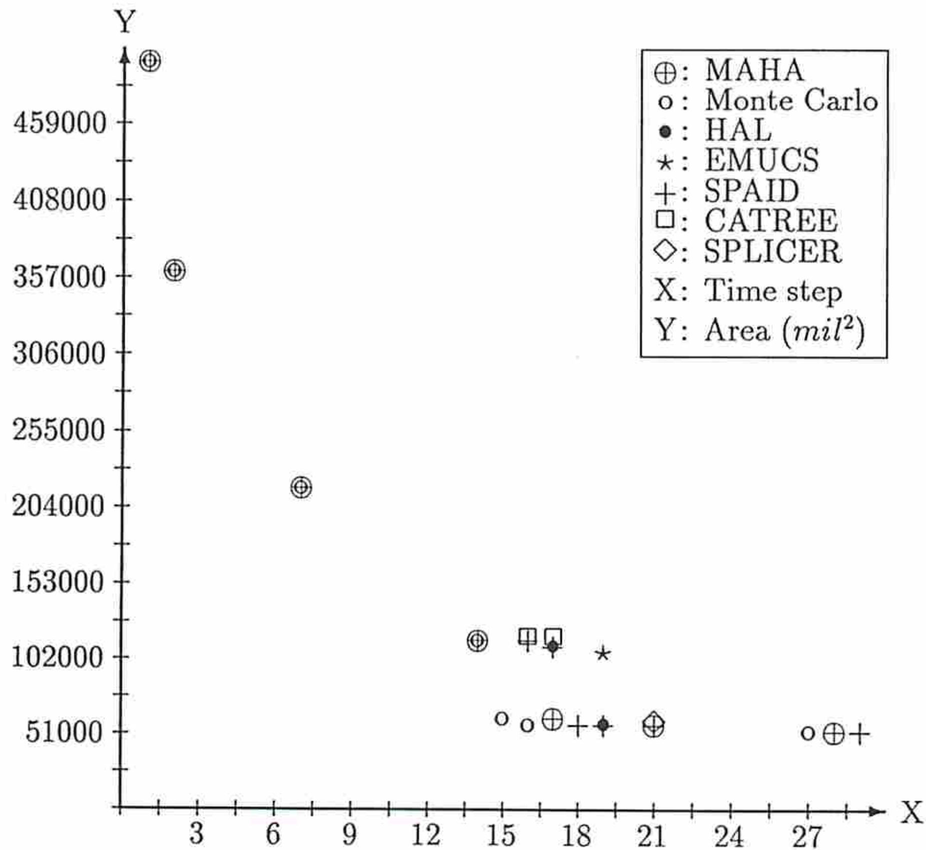


Figure 2.44: Elliptical Filter Designs of Several Synthesis Systems

Table 2.10: Summary of MAHA Validation

Description	Graph Size	Total Designs	Optimal Designs	Worst Case (modules)	
				Area Increase[1]	Time Increase[2]
Simple Cond.	6	6	6	0	0
Multiplier	9	16	15	*	*
Large Cond.	15	6	6/4	0	1
Parallel	16	12	12	0	0
FIR Filter	23	23	20	1	1
AR Filter	28	25	21/16	1	2
Random Gr.	28	25	25/22	0	1
Ellipt. Filt.	36	31	17/15	4	3

[1] The number indicates the quantity of extra modules purchased. A value of "*" means that the total number of modules is identical, but the individual module type counts are different.
[2] The number is the *total* time difference expressed as the number of extra minimum clock delays. A value of "*" means that the delay is the less than the minimum clock (i.e. the faster module).

different, these values are separated by a slash in the *Optimal Designs* column. This second perspective of the non-pipelined synthesis program also indicates that **MAHA** is producing quality designs.

Based upon the analysis, the operation of **MAHA** can be summarized as follows:

- The number of time steps (partitions) or module difference between the optimal and **MAHA** results appear to be weakly influenced by the dataflow graph size.

- Small dataflow graphs and conditional path graphs are subject to greater error in actual area. This is due to the enormous impact of being off the optimal design by even a single module.

- Similarly, as a given dataflow graph is partitioned into more time steps, its likely error increases. Serializing a design involves many more possibilities and, hence, a greater risk that the algorithm will select a non-optimal

design. As the area shrinks and the number of time steps rises, choosing the wrong value has significant impact on the resultant design.

- Finally, the impact on time is much more pronounced than the impact on area for serialized designs. This is due to the discrete nature of time-step scheduling; being off by a slight amount on each clock cycle results in an error magnified by the number of clock cycles.

Another aspect of data path synthesis will demonstrate additional non-optimality as a result of the allocation process. In data path synthesis, the way in which a dataflow graph is partitioned into timesteps can affect the results for a complete design which includes registers and multiplexers. Figures 2.45 and 2.46 illustrate the problem. In this example using the FIR filter, the two different partitioning schemes result in the same time and primary hardware allocation: an eight time-step implementation utilizing a **multiplier** and two **adders**; both of these are *optimal* designs. In Figure 2.45, the **REAL** optimal register allocation scheme utilizes 10 registers. However, in Figure 2.46, only 2 registers are needed. The total area difference between the different implementations due to register costs contributes nearly another **adder** in area. However, the additional multiplexer area due to increased register sharing might overwhelm any savings noted in register area.

In dynamic schemes where register cost is negligible, the register count difference would not present a problem. However, for the static case, the area impact is not negligible. Unfortunately, unless one performs *exhaustive* analysis including all contributing effects, it is unlikely that such design differences will be detected. In short, the non-optimal designs produced by **MAHA** are the same order of magnitude as the secondary effects due to register allocation and partitioning. Until a method which accounts for secondary effects is introduced, the fundamental **MAHA** algorithm is adequate.

2.8 Summary

MAHA is a design tool which assigns operations in a dataflow representation to hardware operators. The resulting bindings can then be passed to a placement

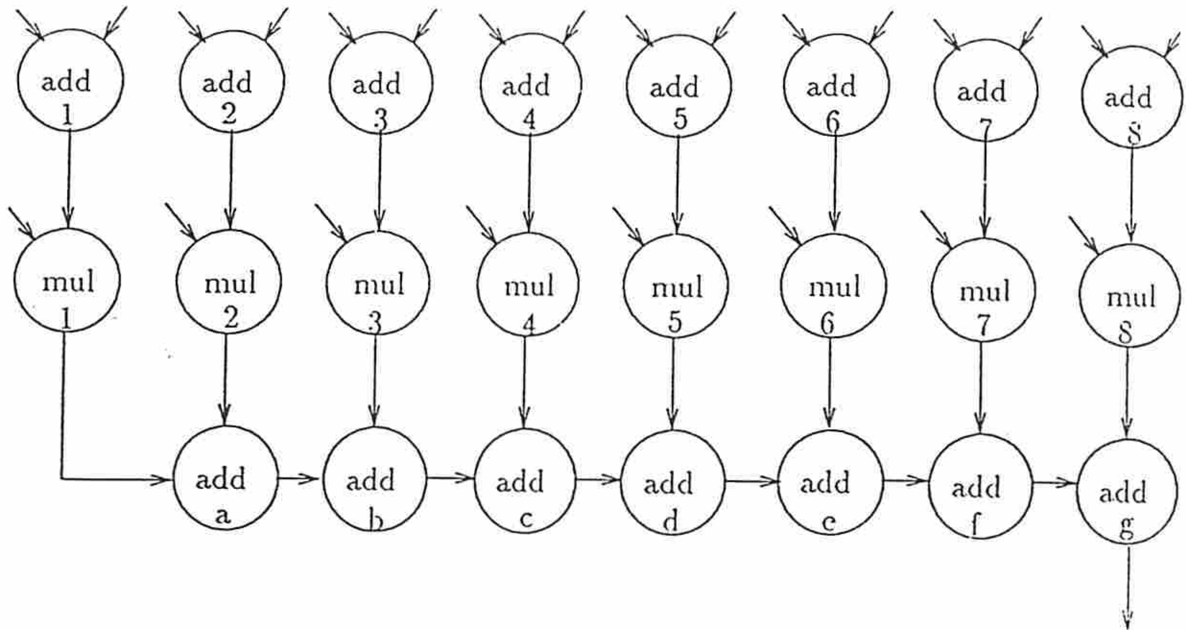


Figure 2.45: Sample Schedule of FIR Filter

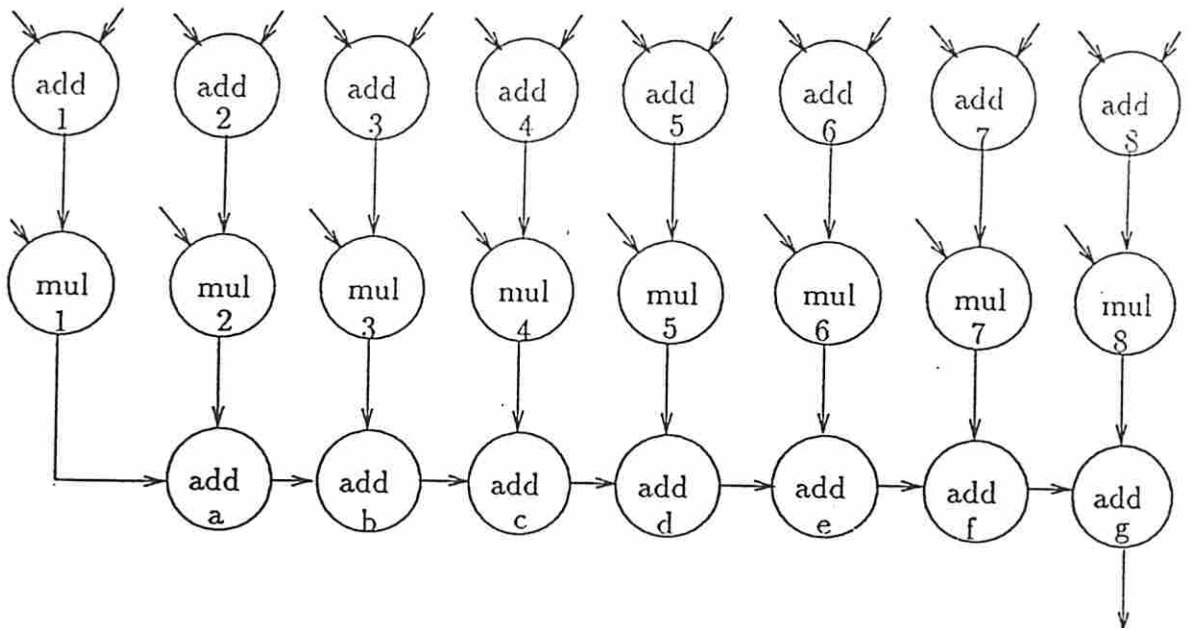


Figure 2.46: Identical Delay/Cost of FIR Filter with Fewer Registers/Muxes

and routing program to produce silicon. As an example, using the MP2D cell library from RCA as the hardware module library, a dataflow representation can be taken to silicon using **SLIMOS** to select the module set, **MAHA** to perform the partitioning and allocation, and **MABAL** to assign the routing and storage hardware [KP90]. The resulting RT level design would be fed to MP2D to produce the final layout.

MAHA illustrates a flexibility not found with other synthesizers, including its adaptability to either area or speed constraints, depending on the application. **MAHA** currently assigns operations to operators, schedules when the operations should or must occur, and allows exploration of the design space given the constraints of the user. Registers are assigned to the data paths as necessary, although **MAHA** does not attempt to share registers during free clock cycles. At present, multiplexers are indicated where needed, with no consideration of when it would be advantageous to use a bus or other types of control for signal paths.

Chapter 3

Area Estimation of Data Path Controllers

3.1 Introduction

Control path synthesis has been heavily researched in the past, but today receives little attention. Due to the regular structures of controllers (both PLAs and microcode), synthesis issues have been mostly resolved excepting area optimization. Current control path literature concentrates upon either minimizing the product terms in PLAs or construction of microcode and nanocode controllers to produce smaller control area [KC86, GR83, WC88]. At present, PLAs are the predominant method used for control within VLSI chips.

Past research has resulted in a number of automated synthesis tools for PLAs [Ham83, DSVV83, KY85]. With a finite state machine (FSM) description as their inputs, these tools produce a PLA layout. As indicated in Figure 3.1, these PLAs consist of two regions. On the left side, one or more of the **inputs** (or their inverse) are ANDed together to form each **product term**. On the right side, each **output** line is driven by one or more product terms which are ORed together. Thus, given a set of active inputs, the PLA will drive some set of output lines. An FSM can be realized by having a subset of the input and output lines represent the *current state* and *next state*, respectively. These lines are then connected through a feedback register as shown in the figure.

During circuit design, the data path is produced, and control of the operators, registers, and buses/multiplexers is determined. Thus, control path synthesis is performed as a distinct step following data path synthesis. However, in the course of evaluating a large design space or complex circuits where multiple designs are

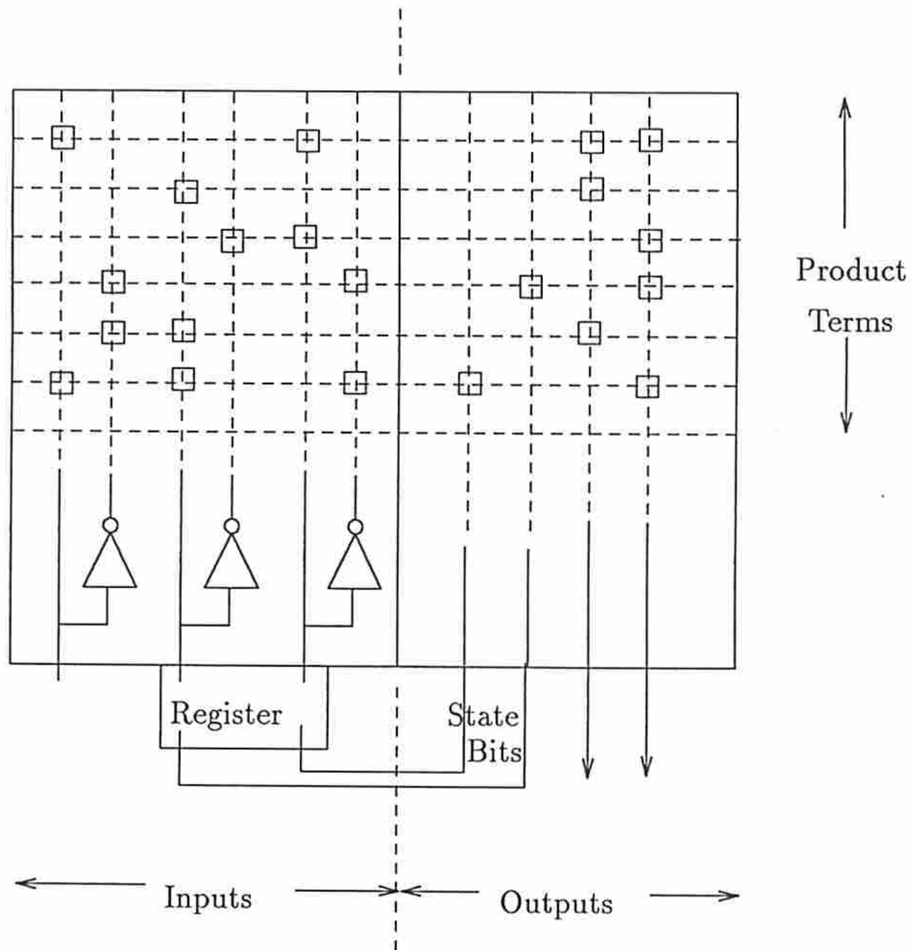


Figure 3.1: PLA Finite State Machine

constructed, such methods are computationally expensive. As shown in Figure 3.2, the PLA synthesis execution time increases exponentially as the number of control states is increased. The existence of a control area predictor would greatly enhance the ability of a system or designer to rapidly explore the design space.

Control area is affected by the data path hardware allocation and scheduling which, in turn, is determined by the dataflow algorithm being synthesized, resource library, and user constraints. A PLA generation tool is usually used after forming the data path using these input parameters. These data path input parameters might also be provided by prediction tools. In this case, specific scheduling and allocation of *individual* operations will not be known; hence, a PLA synthesis tool cannot be used at that time for building a controller. However, a control area predictor may be capable of utilizing data path estimates.

The availability of a predictor based solely upon data path parameters should be capable of integration into data path synthesis programs to immediately assess design changes after partial synthesis. It could also be used in conjunction with data path estimation tools to more quickly explore the design space. With such a prediction, a method is available for evaluating control path/data path tradeoffs prior to data path synthesis. Hence, the research problem described in this chapter is to predict control area from either synthesized or estimated data paths.

In this chapter a theoretical analysis of the area of PLA finite state machines is presented. The solution approach entails:

- describing the PLA area in terms of its control parameters (inputs, outputs, and product terms),
- predicting these parameters using data path attributes,
- considering control of pipelined designs as a special case,
- extending the predictive model to folded PLAs, and
- validating the models using actual designs.

The PLA area estimation technique examines the dataflow graph, its scheduling into timesteps, and register and bus/multiplexer control requirements.

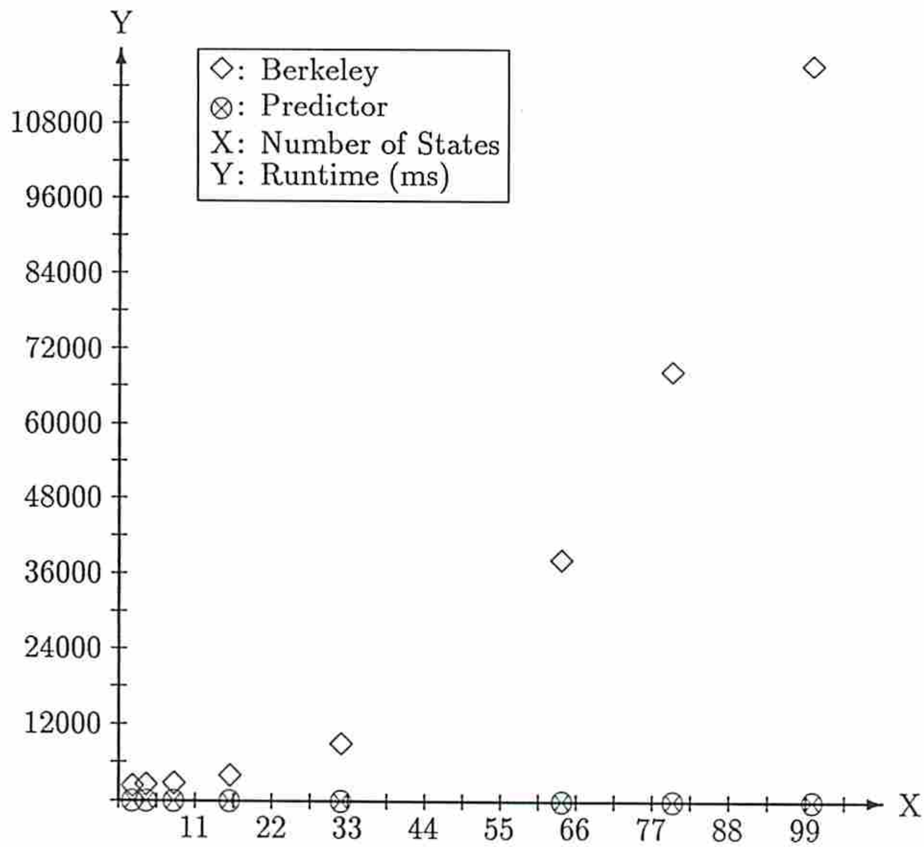


Figure 3.2: Computer Runtime

Although prior discussion has focused upon area, operation time of a PLA cannot be neglected. However, advances in technology have resulted in PLA controllers being used effectively up to 40 MHz. This is reflected in actual designs where high speed circuits have previously used other control methods such as random logic or timing generators. The use of PLA controllers is not believed to be a limitation in most digital designs. Therefore, PLA delay will not be examined.

In the next section a basic PLA controller area prediction model is presented, followed by an analysis of loops and conditional paths. Estimation of folded PLAs and a brief discussion of pipelined control are also included. Finally, experimental results of the complete model are presented.

Several assumptions are made in the development of this control area prediction model.

- The control area estimated is specific to finite state machines.
- The PLA state feedback register is considered part of the control area.
- Hardware external to the PLA which is used for control, such as a counter for loops, is considered part of the control area model. Any hardware not specifically part of the controller is excluded.
- Wiring area within a controller is not fully analyzed. The area of a PLA inherently includes internal wiring connectivity; however, wiring between the PLA and any additional external control hardware, including the feedback register, is ignored. It is presumed this area is a small fraction of the PLA and external hardware, but could be taken into account during wiring area analysis of combined data path and control.

This chapter focuses upon the PLA as the mechanism for control. However, this does not preclude use of the underlying model for other control methods such as multi-level logic.

3.2 A PLA Control Area Model

In a finite state machine the controller performs state transitions based upon the current state and, where needed, external inputs. The Moore state machine model is used here.

Control area is determined by three major parameters:

1. the number of inputs, i, some which are used to represent the *current state* and others as input status from the data path or external control path hardware;

2. the number of outputs, o, some which are used to represent the *next state* plus others for controlling external control path or data path hardware; and,

3. the number of product terms, p, in the PLA.

The PLA parameters i, o, and p are driven by the number of data path states, which is the number of partitions in non-pipelined design and the initiation interval for pipelined design. Other factors such as conditional execution and loops may also affect the external control area, as in the use of a counter for loops, but they predominantly influence the inputs, outputs, and number of control states in the PLA itself.

3.2.1 General Model

The number of FSM states, ζ, determines the number of state bits (the minimum count on the number of input and output lines) which is $\lceil \log_2 \zeta \rceil$.

Assuming that no inputs, outputs, or product terms are redundant is important in estimating these parameters of a PLA. The presence of redundancy would unnecessarily increase the size of the PLA. Unfortunately, the presence of redundant product terms is not uncommon. Currently, there is broad emphasis in the literature on minimizing PLA product terms and the results are encouraging [RSV86, GR87]. One would expect future PLA synthesis tools to produce the smallest possible PLA; hence, only the area of a fully minimized PLA will be predicted.

Inputs to a PLA state machine are comprised of external inputs (to the PLA) and the $\lceil \log_2 \zeta \rceil$ next-state bits fed back from the output. These external inputs reflect the status of external hardware or events which impact either loop control or conditional branch execution. The number of loop status input lines, i_{loop}, is determined by both data path and external control path (loop counter) status. i_{cond}, which is the number of conditional status input lines, are affected by the type and number of conditional branches in the data path. With these values, the number of PLA inputs, i, is

$$i = \lceil \log_2 \zeta \rceil + i_{cond} + i_{loop} \tag{3.2.1}$$

Similarly, the PLA state machine outputs contain the next-state bits and external control lines. Some of these external control lines are associated with loop (counter) control, o_{loop}, and conditional branch selection and control, o_{cond}. The remainder are used for explicit control of registers, multiplexers and auxiliary hardware. The number of control lines associated with registers, R_c, depends upon the type of register control as described in the next section. Multiplexers may be explicitly controlled by the PLA or implicitly controlled via the state feedback bits. In the latter case, the number of multiplexer control lines, M, is set to zero. Finally, the number of control lines for operating ALUs and other multi- function hardware, A, depends upon the data path module selection. Hence, the number of output lines, o, is

$$o = \lceil \log_2 \zeta \rceil + R_c + M + A + o_{loop} + o_{cond} \tag{3.2.2}$$

The number of control states (ζ) is directly estimated from the number of stages (control steps) in the data path design (P) and adjusted by any conditional path or loop state contribution. In particular, if loops are unwound, ζ_{loop} *additional* control steps are needed as detailed in Section 3.3.1. The number of additional control states associated with conditionals, ζ_{cond}, can only be approximated as described later in Section 3.3.2. ζ_{cond} reflects separate control of each conditional path. For example, a 4-way conditional of 3 states each would have $\zeta_{cond} = 9$. The number of control states can be approximated as follows:

$$\zeta \approx P + \zeta_{loop} + \zeta_{cond} \tag{3.2.3}$$

3.2.2 Computing R_c and p

The major difficulty in formulating a PLA area estimator using data path parameters is computing the number of product terms, p. It is clear that the number of control states and number of output control lines directly affect the number of product terms; loops and conditionals may also have an impact. For a simple sequential controller with no loops or conditionals, we show in this section that only state count and output control determine p. The number of output control lines is determined by the control style.

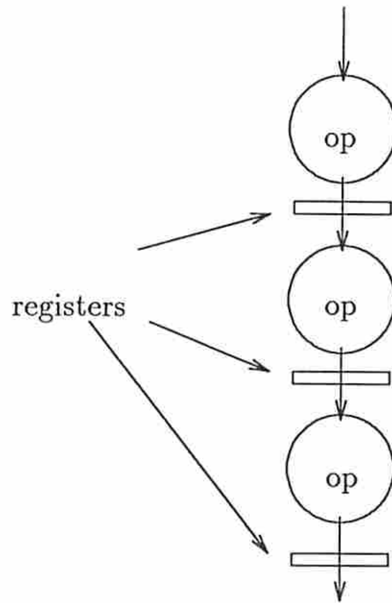
Two distinct PLA control styles whose choice only affects the number of register control lines, R_c, have been identified. One is for the controller to assert a unique output line at each stage or control step; this output line operates all of the registers active in the given stage and is thus labelled "stage control". (For a register used over several stages we OR the appropriate PLA outputs to produce its control.) The second type of PLA controls the registers with output lines directly. This method is labelled "register control". We will choose the approach which yields minimal control area for a given design.

To illustrate, Figure 3.3 contains two different partitioned dataflow graphs. In Figure 3.3a, a single control line may suffice if the same register is used at each stage and, hence, the "register control" method is superior. Conversely, the controller for the graph in Figure 3.3b would be smaller using the "stage control" method. In general, a small number of control states favor "stage control" whereas larger controllers favor "register control" as the minimal area PLA style.

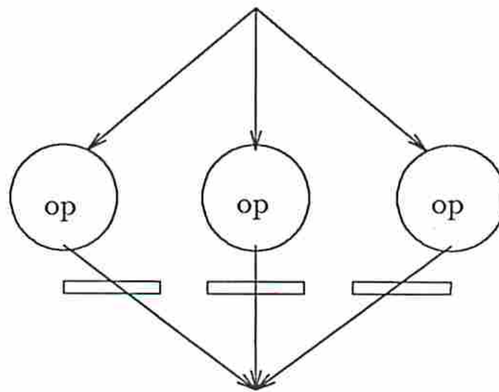
3.2.2.1 Stage Control Method

For the "stage control" method, a unique "stage" control line is asserted during each step of the ζ control stages, yielding an estimated register control line quantity of

$$R_c = \zeta \tag{3.2.4}$$



(a) Serial dataflow graph



(b) Parallel dataflow graph

Figure 3.3: Two Example Data Paths

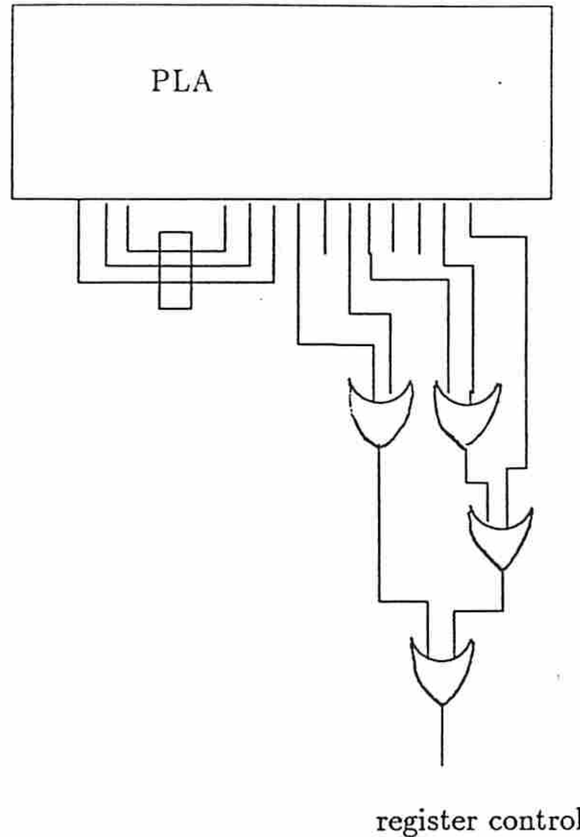


Figure 3.4: 8-state PLA with a register control line shown

The "stage control" method relies on each of the R *uniquely controlled* registers to be controlled by ORing together one or more of the R_c output lines. (If two registers have identical control values in all states, where "don't care" is an automatic match, then these registers can share the same control line.) A schematic view of an 8-stage PLA with one register control line asserted in 5 states is shown in Figure 3.4.

Since during PLA area estimation we do not know during which of the P data path stages a given register control line is asserted, the external control area associated with OR gates is formulated based upon empirical observations. Each of the R uniquely controlled registers is assumed to have its control formed using 2-input OR gates arranged into an $m : 1$ tree. The value for m depends upon the actual register control, which is unknown, and the number of data path stages P. If k is the ratio of stages where a given register is asserted versus the total number of stages, then $m = kP$ and the incremental area associated with

stage control random logic, $Area_{stage}$, is

$$Area_{stage} \approx R \times A_{or} \times (\lceil kP - 1 \rceil) \quad (3.2.5)$$

where A_{or} is the area of a 2-input OR gate. If each register is expected to be used in every state, $k = 1$. For small graphs, where "stage control" is superior to "register control", an empirical value of $k = 0.8$ has been observed. Varying this factor from 0.5 to 1 typically alters the total control area by less than a percent.

The number of product terms for a PLA is simple to determine in the "stage control" case. Since each stage control line is asserted in *one and only one* state as indicated in Figure 3.4, it must be formed by a single product term. There may also be additional product terms associated with loop termination tests, p_{loop}, and conditional branch tests, p_{cond}. Thus,

$$p = \zeta + p_{loop} + p_{cond} \quad (3.2.6)$$

Given the PLA architecture where each state has one product term active that is inactive in every other state, the equations for any next-state, multiplexer control, or auxiliary output such as ALU control can clearly be achieved. One simply ORs together one or more of these unique state product terms for multiplexers and auxiliary control. Next state generation may also rely upon the additional product terms associated with loops or conditionals.

3.2.2.2 Register Control Method

With the "register control" style, each unique register control line is specifically output by the PLA. Given an exact or estimated value of the number of uniquely controlled registers, R, then

$$R_c = R \quad (3.2.7)$$

Determining the number of product terms is more difficult in this case. By construction, one or more register control lines *must* be asserted during each stage; a given register control line may also be asserted during multiple stages. If each register is *only* asserted during a *single* stage, the product term count is identical to that of the "stage control" method. Asserting a register during

multiple stages is typical even for moderately sized designs which offers the possibility of reducing the number of product terms.

In a simple sequential state machine (no loops or conditionals), the next state only depends on the current state. As derived in Appendix C, a *minimum* area ζ-state counter can be constructed within a PLA using $\frac{n(n+1)}{2}$ product terms, where $n = \lceil \log_2 \zeta \rceil$. For designs where $\zeta < 8$, the actual number of product terms is into $\zeta - 1$, which is only one less than the "stage control" method. As the number of control states increases, this difference could increase, but does not. The example using a synthesized design with 19 states will demonstrate the problem.

The synthesized design of Figure 3.5 has 1 multiplier, 1 adder, and 6 registers which execute in 19 clock cycles or states. A simple counter for 19 clock cycles can be achieved using 14 product terms in the PLA. Except for states 1 and 17 through 19, the control states have multiple product terms active. (A single product term simplifies register control.) Of the 6 registers, only 2 can use existing product terms. It takes 11 additional product terms to satisfy the register control requirements, giving 25 product terms total. By comparison, a single product term per state offers the same control capability with only 19 product terms.

Other designs can be tried as well. Since register sharing tends to increase with the number of clock cycles, one would not expect the situation described in the example to change for more serial designs. More parallel designs do have an opportunity for savings as there is less sharing. However, unless *every* register control line can be formed by some ORed combination of state bits *and* controlling a register is possible in the last stage (where all state bits are zero), then ζ product terms are required.

As can be seen, potential savings in product terms by using the register method due to the register control requirements is insignificant. (A detailed explanation is given in Appendix D.) Thus, in comparing the two register control styles, one notes that the product terms and input count are unchanged, but that the number of output terms can change substantially (R versus ζ). Clearly, the method chosen should be the one which results in the minimum control area. The decision whether to select "stage control" or "register control" is influenced

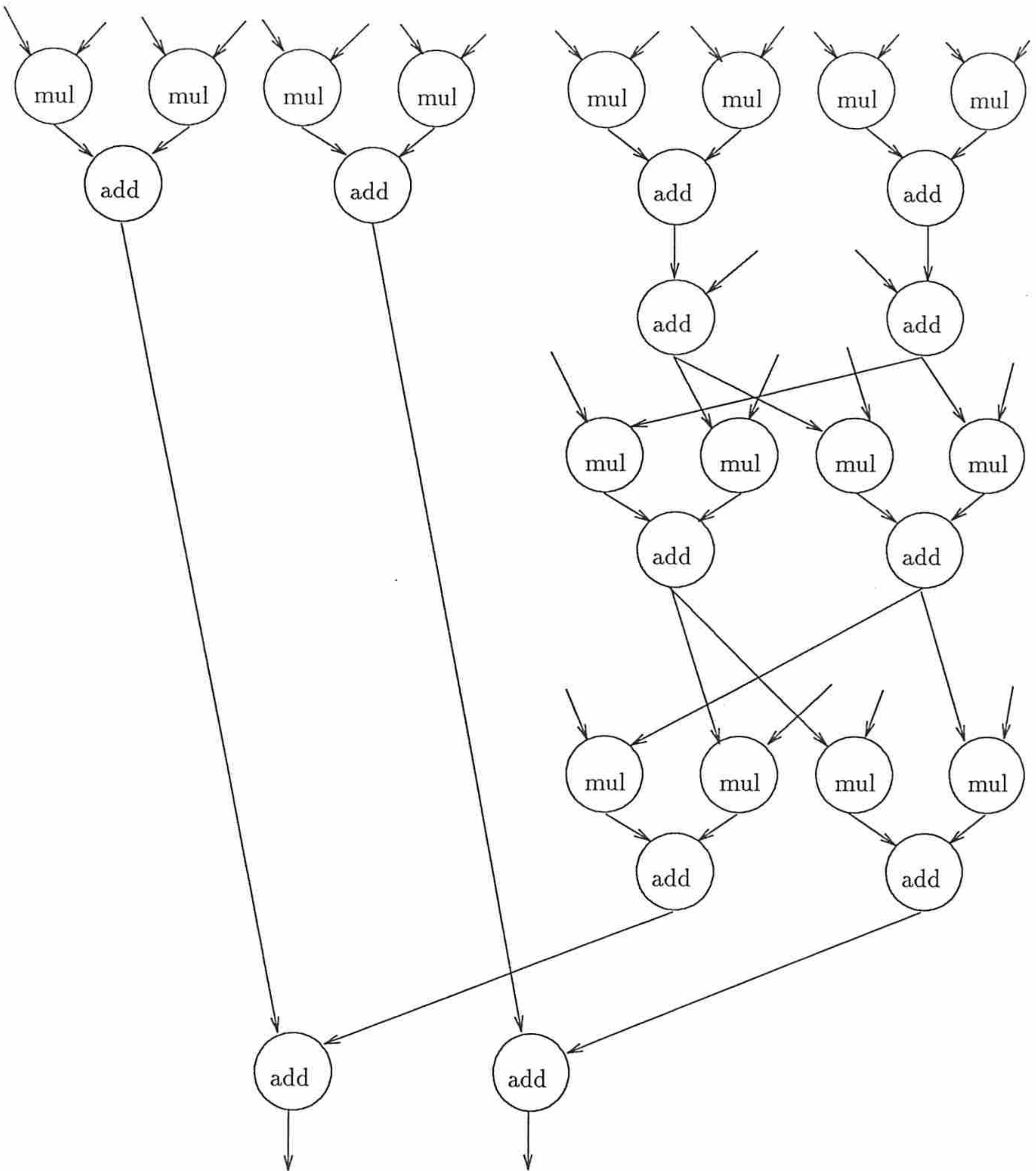


Figure 3.5: AR Lattice Filter

by the number of control states as well as the number of unique register control lines.

3.2.2.3 Multiplexer/Bus Control Lines

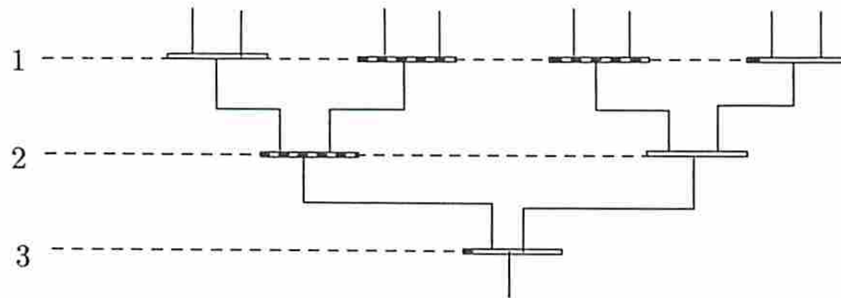
Multiplexers and buses are functionally identical hardware as viewed by the controller. Both select a specific value to be utilized in a given state; buses can be thought of as "global" multiplexers since they are, in general, widely distributed in a circuit as compared to multiplexers which are local to some hardware unit.

The hardware realization of value selection is depicted in Figure 3.6. Here, Figure 3.6a shows encoded control lines and Figure 3.6b shows unencoded control lines. For unoptimized multiplexers, during *one and only one state*, a given input line is selected through assertion of one or more multiplexer control lines. Note that if the same value is selected during multiple states, multiplexer area optimizers reduce the number of multiplexers, which eliminates this single-state uniqueness. Both optimized and unoptimized multiplexers will be considered.

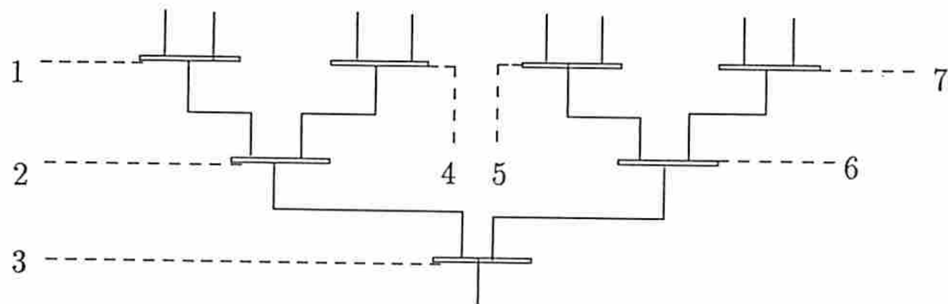
For VLSI chips using cells, $2:1$ multiplexer cells are typically used. An $n:1$ multiplexer tree comprised of $2:1$ multiplexer cells could have as many as $n-1$ instead of $\lceil \log_2 n \rceil$ control lines as shown in Figures 3.6a and 3.6b, respectively. This offers more control flexibility which can be exploited, provided it results in an overall smaller circuit area.

A global examination of the general multiplexer control problem suggests three styles depicted in Figure 3.7: directly using PLA next-state output lines, indirectly using PLA next-state output lines and random logic, or directly by using PLA multiplexer output lines. In the first case, specific PLA state output lines are connected directly to the multiplexer control lines. Clearly, since any PLA next-state output bit combination is *unique* for each state, these lines can be used to control unoptimized multiplexers. No other additional hardware is needed.

When multiplexer area is optimized (duplicate selection hardware eliminated), then additional random logic (AND/OR/INV) may be needed for multiplexer

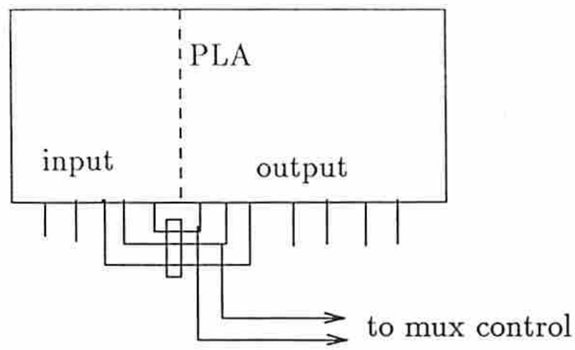


(a) encoded control lines

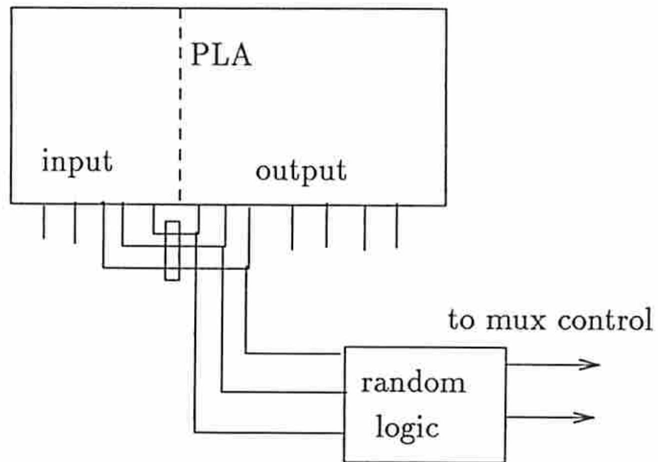


(b) unencoded control lines

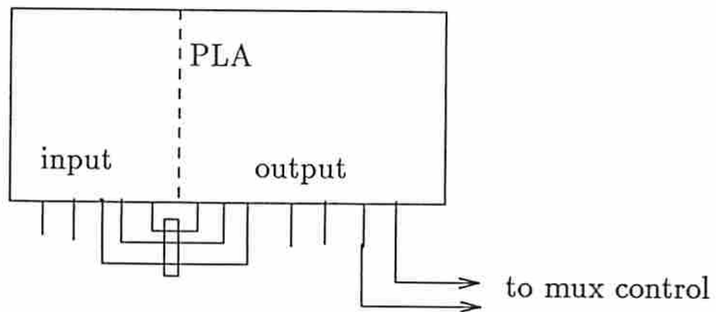
Figure 3.6: Multiplexer control methods



(a) direct mux control via PLA state bits



(b) indirect mux control via PLA state bits



(c) direct mux control via PLA output lines

Figure 3.7: Different PLA multiplexer control styles

Table 3.1: Multiplexer Control Analysis

Dataflow Graph	Stages	Ops/Modules Add	Ops/Modules Mul	module id	muxl/muxr[1]	S/A/O/I[2]
AR	9	12/2	16/2	add1	5/3	3/0/0/0
				add2	3/4	3/1/0/0
				mul1	6/8	3/0/0/0
				mul2	5/8	3/0/0/0
				regs (worst)	3	2/0/0/0
AR	19	12/1	16/1	add1	4/6	4/6/0/3
				mul1	10/16	4/1/0/0
				regs (worst)	2	1/0/0/0
Efilter	8	26/5	8/2	add1	3/2	2/0/0/0
				add2	4/3	2/0/0/0
				add3	5/3	4/2/0/1
				add4	7/3	3/0/0/0
				add5	3/3	4/2/2/3
				mul1	5/5	3/0/0/0
				mul2	2/2	1/0/0/0
				regs (worst)	4	2/2/0/0
Efilter	28	26/1	8/1	add1	7/9	5/8/6/5
				mul1	5/5	4/1/0/0
				regs (worst)	3	2/0/0/0

[1] *muxl* and *muxr* are the number of multiplexers used at the left and right inputs, respectively, for that particular module.
[2] S is the number of state bits used; A is the number of 2-input AND gates, O is the number of 2-input OR gates, and I is the number of inverters.

control as shown in Figure 3.7b. This hardware uses the PLA next-state output bits to indirectly operate multiplexers. The amount of additional hardware required determines the area efficiency of this approach.

As an example, multiplexers used in four synthesized designs based on two dataflow graphs will be examined. Using the minimized multiplexer assignment from **MABAL**, the control lines were analyzed to determine which state bits were used and the amount of additional hardware in the form of AND/OR/INV gates needed. Table 3.1 lists the statistics for these designs. With the exception of the 28-stage Efilter, the additional hardware required was small.

Comparing the approaches which utilize the PLA state bits, it is in the larger designs that random logic needed for indirect control might be significant. If this additional hardware exceeds the area reduction due to multiplexer minimization, then using *unoptimized* multiplexers results in a smaller design. (Multiple control signal patterns are used to select the same value.) For example, in the 28-stage Efilter, approximately 33 transistors of random logic are needed when multiplexers are minimized. The unoptimized configuration requires 10 additional multiplexers of two pass transistors each; thus, a savings of 30% in area is obtained for the direct PLA next-state bit control style.

Finally, the third approach is direct control of the multiplexers via PLA outputs as presented in Figure 3.7c; the number of multiplexer control lines is M. This approach is functionally identical to that of Figure 3.7b except that the decode logic is internal to the PLA. Despite the clean appearance of this method as compared to the indirect PLA control methods, the PLA area for implementing complete multiplexer control via PLA output lines is usually larger than the earlier approaches. To see why this is true, one needs to compare the area of each additional output line versus external random logic.

For NMOS technology, each n-bit AND or OR gate requires $n+1$ equivalent transistors; an INV requires two. As observed in the Berkeley NMOS PLA, approximately $\frac{3}{2}p+6$ equivalent transistors are used for each multiplexer control line *in a folded PLA*. For example, where the 9-stage AR filter uses a total of one two-input AND gate (3 transistors) for indirect control, 60 equivalent transistors are needed within the PLA for the 3 control lines (assuming only 3 lines are needed). If the PLA cannot have its output lines completely folded, then this difference increases. Clearly, none of the designs in Table 3.1 would benefit from direct PLA control. Although designs can be contrived where direct PLA control produces the lowest overall area, design examples (albeit small) encountered had smaller total control area using the indirect control methods. Hence, using existing PLA outputs rather than individual multiplexer control outputs results in an overall circuit area which is lower.

3.2.3 A Specific Model for Berkeley PLA Synthesis Tools

In control synthesis PLA layout generators use predefined macrocells. Given the values for i, o, and p from the earlier equations, the area of a PLA can be defined as

$$PLA_{area} = c_1(2i + o)p + c_2p + c_3(2i + o) + c_4 \tag{3.2.8}$$

where i is the number of inputs, o the number of outputs, and p the number of product terms.

The first term determines the total area of the cellpairs which is a block having an input (or inverse input or output) line and product term line passing through it; these interior blocks are labelled as c_1 in Figure 3.8. The second term computes the output register and interconnect area between the input and output sides as well as the product term pre-charge pair (if any) and ground cell area and are labelled as c_2; the third term describes the pre-charge pair, ground cell, and feedback register area for the input and output (c_3). Finally, the last term is any additional area for "closing the bounding box" and should only be of consequence for small state machines. The coefficients $c_1 \ldots c_4$ of Equation 3.2.8 can be readily determined by measuring the area of each type of macrocell specific to a control synthesis tool.

Initial calibration and verification of the model entailed use of the Berkeley PLA synthesis package [Ham83] and the MAGIC layout tool [OHM+84]. (The PLA package includes PEG to produce the state equations, EQNTOTT which transforms these into a PLA personality matrix, ESPRESSO, which minimizes the product terms of this matrix, and MKPLA, which generates a CIF description of the final controller including the state feedback register.) Initially, ten arbitrary PLAs (each with a randomly created personality matrix) were constructed using MKPLA and "CIF block" areas were analyzed using MAGIC to determine the coefficients. The Berkeley package defaults to 2 um NMOS; thus, the particular coefficients used for our experiments use this technology. Based upon the interior blocks measured, the coefficients which give the PLA area in square microns are

$$c_1 = 73.7717$$

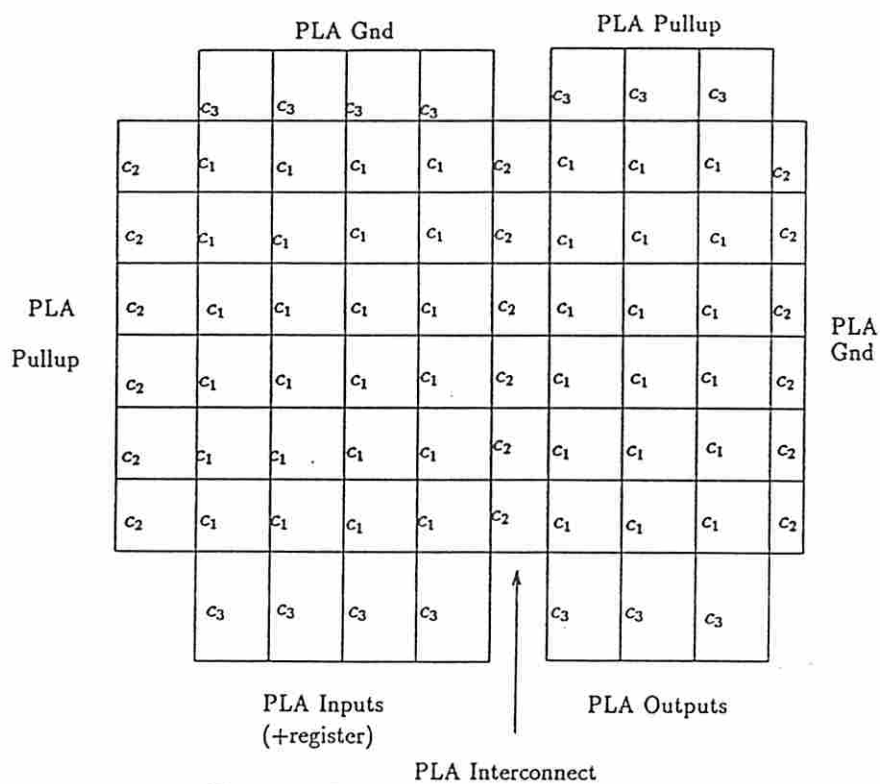


Figure 3.8: Internal construction of PLA

Table 3.2: Validation of PLA model

PLA			MAGIC			CIFAREA			Model
In	Out	Pterm	H	W	Area	H	W	Area	Area
3	6	5	124	131	25.2	121	131	24.6	24.5
4	11	10	156	196	47.4	153	195	46.2	46.8
5	25	20	236	325	118.9	233	323	116.7	120.6
6	12	40	407	231	145.7	401	227	141.1	138.4
6	56	50	492	612	466.7	483	603	451.4	446.2
7	40	70	660	488	499.2	651	483	487.4	490.2
7	87	80	761	908	1071.0	741	895	1028.0	1036.6

$$\begin{aligned} c_2 &= 19.603 \\ c_3 &= 611.698 \\ c_4 &= 3025.94 \end{aligned} \tag{3.2.9}$$

To avoid manually using MAGIC during later validation, a utility (CIFAREA) was written which analyzes the MKPLA CIF file and outputs the bounding box in λ and total area in square microns. Table 3.2 summarizes results of seven specific PLAs processed by MAGIC, CIFAREA, and the PLA model. **H** and **W** are the height and width of the PLA in microns; **area** is the total bounding box area in square mils.

Note that MAGIC is only capable of determining the area of a user-defined box. Thus, one was created which encapsulated the PLA to the best ability of a human user and used for comparison. It is therefore not surprising that the area given by MAGIC is always greater than CIFAREA; human perception of enclosure entails inclusion of some bounding area. Since CIFAREA is reasonably precise, all PLAs described later in this chapter have their area computed using this program; validation time is reduced and introduction of human errors minimized.

To verify Equations 3.2.8 and 3.2.9, over 100 PLA personality matrices were randomly generated and synthesized using the Berkeley tools. The size of the controller varied from a single input, output, and product term up to a controller with 22 inputs, 45 outputs, and 100 product terms. Comparison against the

```
while(input_1 is FALSE)
begin
    perform some functions
        (over one or more control steps)
end
```

Figure 3.9: Loop on Status Flag

results from CIFAREA resulted in a worst case area error of 3.3% for the smallest example with an average error of 0.73%.

One feature of using macrocells in the Berkeley toolset is that each cell handles a pair of lines. Since the number of internal input lines is always even, there is no influence on this parameter. However, both the output and product term values are rounded up by **PASTA** to the nearest even number to be consistent with this layout construction. Other PLA tools may not be restricted to paired lines for each primitive cell.

3.3 Loops and Conditionals

In the previous section, a detailed model for PLA area for controllers without loops was derived. In this section, the effects of loops and conditionals are derived and their impact on the PLA parameters assessed.

3.3.1 Effect of Loops on PLA Area

Loops are a common construct in control. For the most trivial case, a PLA state machine has one inherent loop where the last state connects to the first state. Cycles in the dataflow graph which become minor loops in the controller are of more interest. Each of the minor loops is labelled as α_i. Loop parameters associated with loop α_i consist of its length in control steps, $PL(\alpha_i)$, and the number of iterations associated with it, $N(\alpha_i)$. There are three types of loops: loop on status flag, variable count and fixed count; these are depicted in Figures 3.9 through 3.11.

```
for i:=1 to 6
begin
   perform some functions
      (over one or more control steps)
end
```

Figure 3.10: Fixed Count Loop

```
for i:=1 to variable₁
begin
   perform some functions
      (over one or more control steps)
end
```

Figure 3.11: Variable Count Loop

The simplest type of loop is where a status condition or flag (generated externally to the PLA) is checked to determine loop termination as shown in Figure 3.9. Since the loop operations are invariant, only a single test is performed to determine restart or exit from the loop. ($N(\alpha_i)$ is unknown in this case.)

By construction, each product term determines a specific next-state based upon the input lines. An external status flag consumes one input line which did not exist in a simple sequential state machine. As the outcome of the test chooses one of two distinct states, there must be two product terms rather than one. One next-state product term enables restart of the loop; the other terminates the loop. Although two product terms now select the same state (for loop start as well as loop restart), they cannot be merged due to the presence of the loop status variable in one which is absent in the other.

In the remaining cases (variable or fixed iteration loops), one can either "unroll" the loop or provide a count mechanism for tracking the loop as presented in Figures 3.10 and 3.11, respectively. In the first method, a fixed loop is unrolled into a number of PLA states so that no branching (to the top of the loop) is

necessary. (A variable loop could be implemented by unrolling the loop into the maximum number of loops expected; an "exit condition" would check for loop termination at the end of each unrolled loop and skip to just beyond the last loop state. If the exit condition were true, in this case, $N(\alpha_i)$, the number of iterations for loop α_i, is set to the maximum number of iterations.) As will be shown, unwinding may be useful for very small loops, but is impractical for either large or long loops because the number of control states increases dramatically.

3.3.1.1 Loop Unrolling

Each of the $N(\alpha_i)$ iterations of loop $alpha_i$ consists of $PL(\alpha_i)$ control steps; $PL(\alpha_i)$ states inherently exist as part of the acyclic control. Thus, the *additional* number of PLA states due to loop unrolling, ζ_{loop}, is

$$\zeta_{loop} \leq \sum PL(\alpha_i)[N(\alpha_i) - 1] \qquad (3.3.10)$$

The sum is taken over all α_i which are being unrolled. The inequality arises from the case where, given two or more loops,

- the loops are the same length (PL),
- the loops occur in the same timesteps (completely overlap in time),
- the loops have the same number of iterations (N),
- and the loops are in different parallel branches (not conditional branches).

Unrolling (unwinding) a loop is estimated to increase the number of control states by ϕ which is computed as

$$\phi = PL(\alpha_i)\,[N(\alpha_i) - 1] \qquad (3.3.11)$$

Register sharing between different iterations of the loop is considered. The degree of register independence in a loop, K_{reg}, which is a value between 0 and 1 inclusive, will affect the amount of additional hardware needed for loop operation. (Each value-storage operation during loops can be loop-dependent, which means different physical registers are used during each loop iteration, or

loop invariant which means the same register can be reused.) For example, a 16-bit add operation using a 4-bit adder would have loop dependent registers and $K_{reg} = 1$. A 4-bit multiply using shift-and-add would have a 16-bit register which is loop invariant and $K_{reg} = 0$.

Since registers in this case are controlled directly via the PLA, there are approximately $K_{reg}\phi$ additional product terms and output lines needed when the percentage of loop-invariant registers is low. These additional product terms also produce the next-state value. However, when K_{reg} is small, then the minimal number of product terms associated with a simple counter contributes to control area as only the normal next-state generation is needed. The effective counter size, ψ, is inversely related to K_{reg} and computed as

$$\psi = \begin{cases} (1 - K_{reg})\phi & \text{if } (1 - K_{reg})\phi > 1 \\ 1 & \text{otherwise} \end{cases} \tag{3.3.12}$$

The combined effect upon the incremental area associated with unrolling, $Area_{Unroll}$, becomes

$$Area_{Unroll} \approx c_1 K_{reg}^2 \phi^2 + (c_2 + c_3) K_{reg} \phi + (c_1 + c_2) \frac{\lfloor \log_2 \psi \rfloor (\lfloor \log_2 \psi \rfloor + 1)}{2} \tag{3.3.13}$$

In the unlikely situation where these four conditions are met, the same PLA states can be used to control both loops. However, since the inequality presumes this condition is detectable by some complex state minimization utility, the inequality of Equation 3.3.10 is removed in the PASTA implementation. **For unrolled loops, the values for i_{loop}, o_{loop}, and p_{loop} are zero.** The inputs, outputs, and product terms derived from ζ (which includes ζ_{loop}) will reflect the number of additional states incurred by unrolling the loop.

3.3.1.2 Implementing Loops using a Counter

Loop control hardware using a counter has a variety of implementations. Two primary schemes involve building a counter as part of the PLA or using an external counter plus some additional hardware. Four operational considerations are

1. initialization of the counter,
2. decrement of the counter,
3. testing for count completion,
4. and uniquely defining the state (for clocking the correct data path register).

A comparison between an external counter and a PLA counter was accomplished for 3 *um* NMOS technology with the Berkeley PLA synthesis tools [Ham83]. As compared to external hardware using the same technology, the incremental PLA area to construct an internal counter is greater than attaching an external counter under PLA control. One would expect this difference to hold for other technologies as well. Detailed analysis can be found in Appendix C.

A counter used by the controller for operating loops will be assumed by PASTA to be constructed externally to the PLA. One example architecture of a PLA containing an external counter (with external decoding for the loop) is shown in Figure 3.12. Here, a simple 5-state machine has a loop with three iterations during the middle two steps (for a total of nine states).

Use of an external counter impacts the PLA parameters for each loop. A given controller has N_c external counters. Each of these counters may have a different bitwidth; n_i is the bitwidth of counter i. It is assumed that individual counters do not share any control lines. Thus, from Appendix C, the increases in PLA inputs, outputs, and product terms due to loops are

$$i_{loop} = N_c \quad (3.3.14)$$

$$o_{loop} = \sum_i (n_i + 1) \quad (3.3.15)$$

$$p_{loop} = N_c \quad (3.3.16)$$

An area comparison between unrolling a loop and using an external counter determines which method to use for a given design. Although non-overlapping counters could be shared to reduce external area for an actual design, counter cost is not a dominating factor, so the analysis presented here assumes there is no sharing among counters.

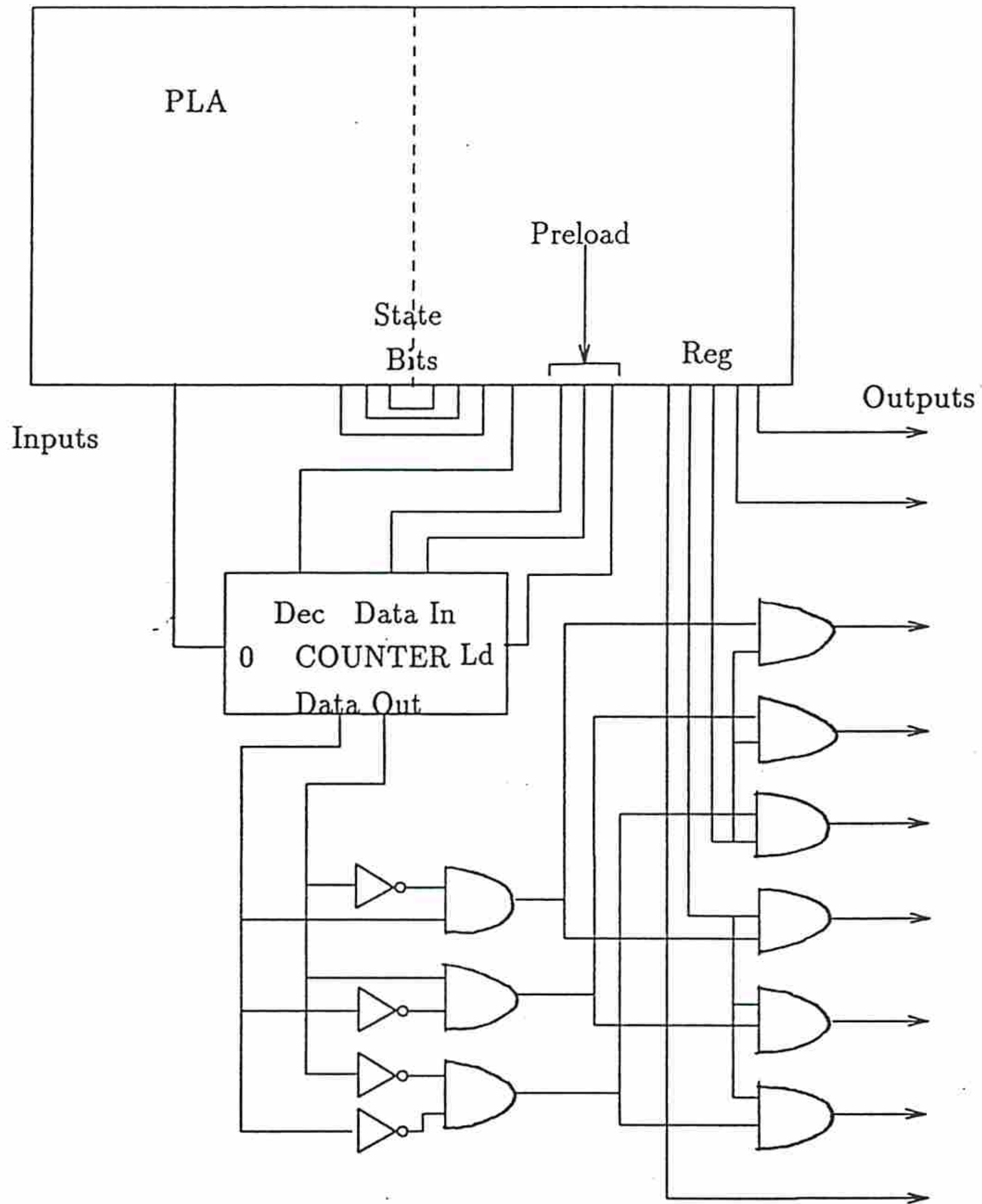


Figure 3.12: PLA with External Counter and Register Decoding for Loop

As shown in Figure 3.12, there are several contributions to circuit area for each external counter. The n-bit resettable counter itself has an area of nA_{cntr}, where A_{cntr} is the area of a single-bit counter with carry. A decoder and AND gates are also used to load the proper registers during a given loop iteration. The maximum amount of hardware is needed if registers are completely loop dependent ($K_{reg} = 1$); no extra hardware is needed if registers are loop invariant. Decode hardware distinguishes a loop state; its area is thus approximated by $K_{reg}PL(\alpha_i)nA_{decod}$, where A_{decod} is the area of a $1:2$ decoder. Similarly, AND gates determine which loop-dependent register to select during each loop state contributing $K_{reg}PL(\alpha_i)N(\alpha_i)A_{and}$ to the area, with A_{and} the area of a 2-bit AND gate. The incremental area increase of a PLA controller having an external counter, $Area_{Ext\ Cntr}$, becomes

$$Area_{Ext\ Cntr} \approx (c_1+c_3)(n+3)+c_2+nA_{cntr}+K_{reg}PL(\alpha_i)N(\alpha_i)A_{and}+K_{reg}PL(\alpha_i)nA_{decod} \quad (3.3.17)$$

as derived in Appendix C. (Note that $K_{reg}PL(\alpha_i)$ must be an integer.) The first two major terms in Equation 3.3.17 reflect the PLA area increase to operate the counter; remaining terms are for external hardware.

Examining Equations 3.3.17 and 3.3.13, one can determine the correct implementation before synthesis of the PLA. Table 3.3 lists the "breakpoint" between unrolling a loop versus an external counter. The value listed at each position is the number of loop invariant register control lines. If the number of lines for the target controller meets or exceeds this value, then the loop should be implemented using an external counter. Other symbols shown are $\star$ to signify that the unrolled loop has the lowest area regardless of the number of loop invariant register control lines, and $\diamond$ when the external counter always has the lowest area. Given the number of iterations and loop length, the number of loop-invariant registers (and, hence, number of control lines per loop) solely determines which approach is superior. As expected, except for very short loops and small iterations, the smallest area is realized using an external counter.

Table 3.3: Unrolling versus External Counter for Loop Implementation

	Loop Length ($PL(\alpha_i)$)							
Iter ($N(\alpha_i)$)	1	2	4	8	16	32	64	128
2	⋆	⋆	4	3	3	3	2	2
3	⋆	2	2	2	2	2	1	1
4	⋆	2	2	1	1	1	1	1
5	⋆	2	1	1	1	1	1	◇
6	◇	1	1	1	1	1	1	◇
8	◇	1	1	1	1	1	1	◇
16	◇	1	1	1	1	1	1	◇
64	◇	1	1	1	1	1	1	◇
128	◇	1	1	1	1	1	◇	◇

⋆ ◇ See text for explanation.

3.3.2 Effect of Conditional Branches on PLA Area

Another important extension for a PLA model is handling of conditional branches such as the *if-then-else* or *switch-case* statements in programming. Here, a number of mutually exclusive data paths could be executed dependent upon some condition external to the PLA. Conditional branches can be implemented in two different ways: different state sequences for each conditional path or "sharing" the same states among different branches.

Definition 3.1 *A* **conditional subgraph** *is a directed subgraph of a dataflow graph which has a* **distribute** *operation as its source and the matching* **join** *operation as its sink.*

Definition 3.2 *A* **conditional path**, *β, is any path in a conditional subgraph from its source to its sink.*

In a dataflow graph, each path β starts at a **distribute** operation and ends at the associated **join** operation taking any arbitrary branch including nested conditionals that are encountered along the path. Complete overlap of common operations is not allowed; thus a restriction for any two paths, β_i and β_j, for some operation o_{beta_i}, $o_{\beta_i} \in \beta_i$ and $o_{\beta_i} \notin \beta_j$ or the reverse, exclusive of the **distribute** and **join** operations. A given conditional path may contain other conditional paths within it that are associated with nested conditionals.

Definition 3.3 *A* **maximal conditional subgraph** *or* **mcs** *is a conditional subgraph that is* **not** *contained in any other conditional subgraph. The set of "outermost" conditional subgraphs in a dataflow graph is composed solely of maximal conditional subgraphs.*

Two different paths within the same **mcs** may be mutually exclusive if they lie along different different branches of the same conditional. However, if these paths lie in two different **mcs**, they are not mutually exclusive. This is important for determining the number of additional control states needed in the PLA.

One method for extending the PLA model for conditionals is to assume each conditional path has a set of unique states. The location of the conditional is important in determining the number of control states which are added. In particular, where m-way conditionals which occur along the primary (critical) path, only $m-1$ of these paths contribute *additional* control states. (The critical path is the longest path in the graph.) As demonstrated in Figure 3.13a, only two of the three conditional branches contribute *additional* control states. In the absence of the conditional, the control states along one path would still be counted.

Each conditional path β_i has $PC(\beta_i)$ control steps. $\mathcal{C}$ is a binary function whose value is 1 if the β_i conditional is not on the critical path and 0 otherwise; if a conditional path is part of the critical path, the states associated with it have already been counted. For a PLA controlling conditional j, where all β_i paths are different mutually exclusive branches of j and D_j is the total number of conditional paths of j, the number of PLA states is increased from the unconditional case to

$$\zeta_{cond}(j) = \sum_{i=1}^{D_j} \mathcal{C}(\beta_i) \times PC(\beta_i) \tag{3.3.18}$$

over all conditional paths β_i.

A more complicated case is where two (or more) conditionals are contained in different **mcs**es. For each control state where *both* conditionals are active, the number of control states is the product of the number of conditional paths for each. This complex example is illustrated in Figure 3.14. j_t is the number of

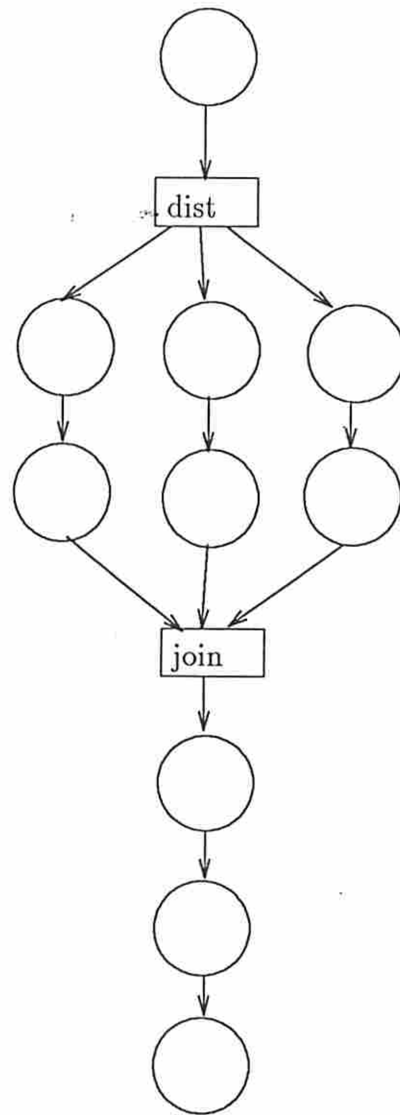


Figure 3.13: Conditional Path Example

conditional paths simulataneously active in control step t. A_i is the number of conditional branches of conditional i which could be active during control step t. In the example, $j_t = 2$ and $A_1 = A_2 = 2$ for 3 control steps (one per operation).

Within **PASTA**, estimating the number of control steps is performed on a state-by-state basis dependent upon the input description. Since exact overlap of operations between paths is not known, the additional number of states, ζ_{cond}, is estimated as

$$\zeta_{cond} \approx \sum_{t=1}^{\zeta} \left(\prod_{j_t} A_1 A_2 \ldots A_{j_t} - 1 \right) \tag{3.3.19}$$

where t is the control state index. Control of the graph shown in Figure 3.14 would entail adding 9 control steps.

Unlike sequential data paths where register sharing is not obvious, conditional branches provide a clear opportunity for sharing registers. In the "stage control" method, one only need consider the longest path in any maximal conditional subgraph to determine the number of control lines necessary. Within each of the maximal conditional subgraphs, the same register control lines can be used at each control step independent of which conditional branch is taken. However, different **mcs** need different control lines and the sum of each maximal path is taken. This value is offset by the number of timesteps associated with a conditional path which lies on the critical path, PC'_{mcs}. Thus, the number of additional output lines needed, o_{cond}, is

$$o_{cond} = \sum_{mcs} \max_i \{PC(\beta_i)\} - \sum PC'_{mcs} \tag{3.3.20}$$

o_{cond} is summed over all maximally conditional subgraphs.

The number of product terms is also adjusted for conditionals. Using the same reasoning as that given for the loop termination condition, a U_k-way conditional requires $U_k - 1$ product terms to choose the correct conditional branch. Thus, the number of additional product terms for conditionals, p_{cond}, is

$$p_{cond} = \sum_k (U_k - 1) \tag{3.3.21}$$

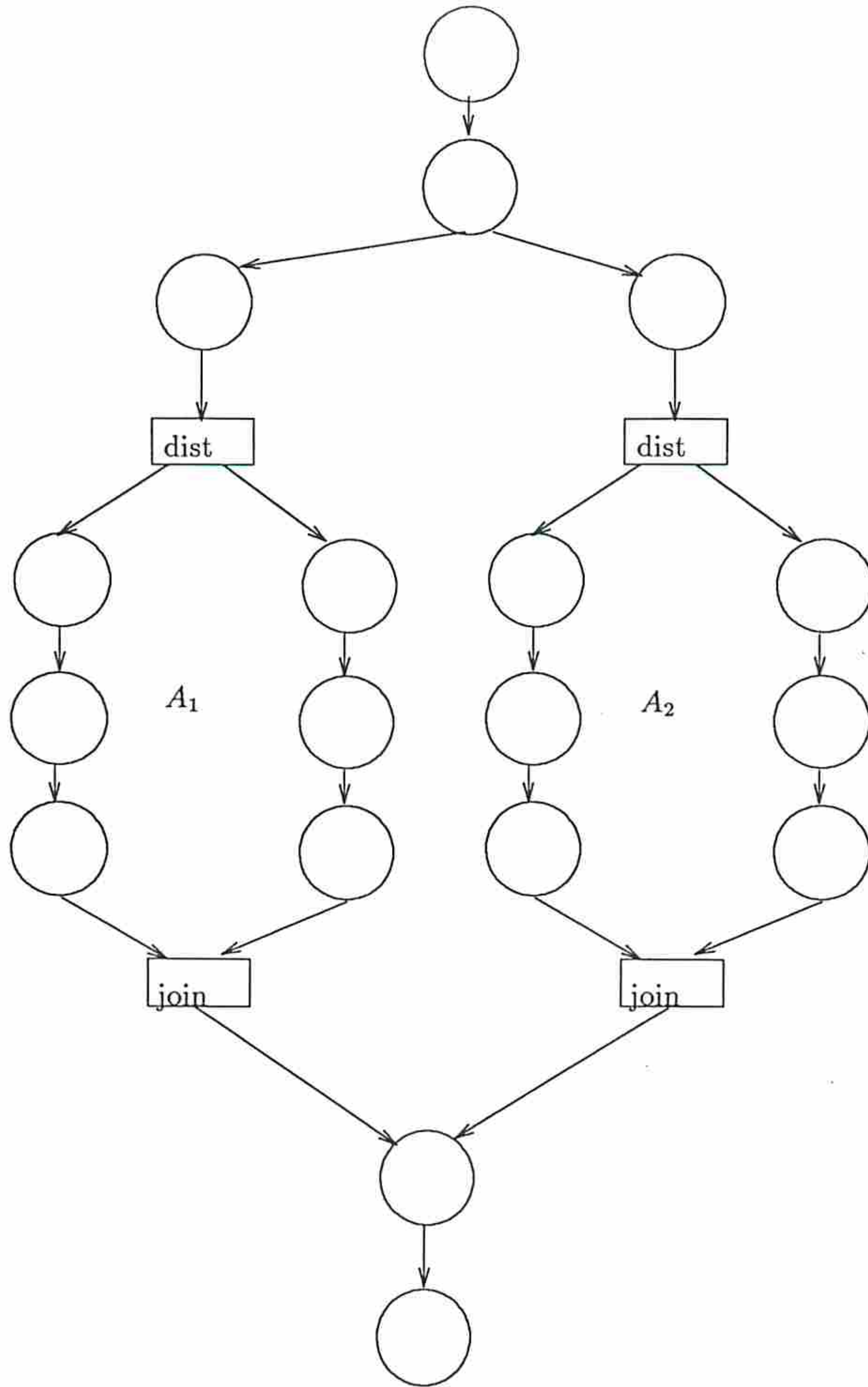


Figure 3.14: Complex Conditional Path Example

and summed over all conditionals. In the "register control" method, each register already has a control line which will be coupled to the additional product terms to reflect conditional usage. Thus, $o_{cond} = 0$, since no additional register control lines are required.

The use of status bits offers another method for a PLA to operate conditionals. A status bit (or bits) determines which of D conditional paths to operate and requires $\lceil \log_2 D \rceil$ PLA inputs. In order for one path to be distinguishable from another, a separate product term is required for each unique "state". Similarly, the output lines must control the proper hardware and may have to determine the next state based upon the inputs. The number of outputs and product terms remain unchanged since the status bit is used to distinguish between several states which share the same product term. Likewise, the number of registers controlled does not change. However, since each conditional path is likely to need a unique status bit (and is a requirement for overlapping conditional paths in parallel branches), the status-bit method input line count can exceed the input count for the state expansion method. As a result, if status bits are used, they may yield a larger PLA.

3.4 Extending the PLA Model

There are two natural extensions to the PLA model described earlier. First, only a minimized PLA was considered. In some designs which have more placement freedom and where space is at a premium, folding of the PLA controller is performed. Estimating the size of a folded PLA is useful for such designs. In addition, the PLA model is specific to control of a non-pipelined data path design. By adjusting the model, control of pipelined designs can be accommodated. Both of these extensions are discussed in the following sections.

3.4.1 Estimation of Folded PLA Area

Folding of PLAs is a common practice for reducing their size. Numerous heuristic techniques are available, but few models predicting folding possibilities have appeared until recently. A unique approach which uses statistical modeling to

quantize the amount of PLA folding has been published [MT86]. Using as a basis the number of rows, number of columns, and cross-point density of a PLA, Makarenko analytically derived and verified experimentally a probability density function (PDF) for the expected number of folds.

This model has two limitations. One is the assumption that cross-point density is uniform. In most PLAs, cross-point density varies. The second limitation is that accurate results could only be obtained by repeated trials of a "random selection heuristic" after tuning to a particular PLA generation algorithm.

To overcome these limitations, the folding model as described in the equations of Theorem 3.16 in [MT86] was only applied to the output lines of the PLA; folding of input columns presents a special problem. The authors of this paper [MT86] suggest that inverters be provided at the inputs as needed. However, if a and $\bar{a}$ appear on opposite sides of the PLA as inputs, a must still be routed to the other side of the PLA to produce its inverse. This would likely result in an area increase. In addition, the cross-point density on the input side for a state machine is generally close to 0.5 - a value where folding is no longer possible according to the probabilistic model.

Conversely, on the output side, regardless of which of the basic methods described here is used to predict the state-machine parameters, the cross-point density is more closely distributed. Furthermore, except for very small PLAs, the density is less than 0.50 making this portion amenable to folding.

With this adjustment to the folding model, multiple sampling is not necessary to account for a particular folding algorithm; the PDF value is calculated once for each folding quantity. This is possible as folding methods published (e.g. [BB87] and [RSV86]) yield near-optimal results for the output columns as predicted in other publications ([LVSV82], [LA83]).

Inputs to Makarenko's folding model are the number of rows (r), number of columns (c) and cross-point density (d). The output rows and columns are trivial to compute:

$$r = p \tag{3.4.22}$$

$$c = o \tag{3.4.23}$$

Cross-point density here is the number of physical connections between product terms and output lines. (An example PLA is shown in Figure 3.1 with cross-points marked.) Since PASTA assumes a simple state assignment where sequential states are represented as sequential binary numbers, an exact value can be determined for connections on the output state bits by summing the binary '1's in each state bit representation over all states, giving S_b. A lower bound can be computed for the remaining output lines. Regardless of whether "stage control" or "register control" is used, each product term drives at least one register/stage output line. In addition, multiplexer/bus and auxiliary outputs (if any) must be driven by at least one product term. Thus, a lower estimate of crosspoint density is

$$d = \frac{S_b + p + A + M}{o \times p} \tag{3.4.24}$$

Since A and M (auxiliary and multiplexer) output lines are generally a small quantity, the density error will be small. Otherwise, the lower crosspoint density may result in higher folding predictions than is actually possible.

The PDF described in [MT86] is taken over all folding numbers from zero to $c/2$ using the crosspoint density described here; the fold number with the highest probability is taken as the estimate on the number of folds. Experimental results are described in Section 3.5.

3.4.2 Extensions for Pipelined Controllers

Pipelined designs have additional control considerations since multiple portions of the data path are being independently and simultaneously operated. Problems include tracking valid data in the pipe (fill/flush) as well as handling conditional paths. Note that inner loops are not allowed within the pipe itself, so they will not be considered here.

One method for tracking valid data in the pipe is for the controller to track and update the pipe status during filling, full operation, and flushing of the pipe contents. In practice, this is rarely done. Rather, a status bit is maintained at each stage which indicates data validity. (In some high-speed applications such as video processing, where occasional invalid results are still acceptable, data validity checking is not even performed.) Essentially, the valid bit is introduced

with the data and is reset during any stage where the data becomes invalid. This bit propagates to the end of the pipe stage-by-stage; it is also used to disable any detrimental operations that might be performed during a given stage. For example, if a memory write should only contain valid data, this bit is used to inhibit the write.

The control problem focuses upon handling unconditional and conditional pipelines, which is an extension of the non-pipelined case previously described. Two new terms are introduced: initiation interval (l), which is the number of clock cycles between successive initiations of input data, and microcycles (u), the number of groups of clock cycles which overlap in execution.

$$u = \left\lceil \frac{P}{l} \right\rceil \tag{3.4.25}$$

P is the number of clock cycles before a given dataset is completely processed and appears at the output.

Since data is introduced every l clock cycles, l is the basic number of states in the PLA provided there is no explicit fill/flush mechanism other than the use of "valid" bit registers.

$$\zeta = l \tag{3.4.26}$$

The PLA area estimation equations accept l instead of P to determine the PLA parameters for unconditional pipelined designs.

Conditional branches pose a more difficult problem. If a conditional branch appears across two or more microcycles, a different part of the path may be active in each. One must consider all possible distinct paths in determining the number of PLA states. Thus, the state contribution for conditionals, ζ_{cond}, becomes

$$\zeta_{cond} = \sum_{i=1}^{l} \zeta_i^l \tag{3.4.27}$$

where

$$\zeta_i^l = \prod_{k=1}^{u} DC'(\beta_i^k) \tag{3.4.28}$$

For each relative clock cycle of the l in a microcycle, the number of distinct paths in every microcycle is multiplied together. $DC'(\beta_i^k)$ is the number of distinct

conditional paths in microcycle k and relative clock cycle number of i; it is one if there are no conditional paths. The number of PLA states is determined by the summing the value for each clock cycle over the initiation interval.

From Equation 3.4.28, it can be seen that the controller size grows as the number and length of the conditional branches increase. Eventually, it would be less costly to use one controller in each microcycle operating the l stages. This tradeoff is a topic of ongoing research in control area estimation which will be examined in Chapters 5 and 6.

3.5 Validation of the PLA Model

In the previous sections, models for PLA state machine parameters were developed for sequential dataflow graphs as well as for loops and conditional branches. Extensions for PLA folding and use in pipelined designs were also discussed. Experimental results are presented in this section.

3.5.1 Validation Process

To validate the PLAs, a number of scheduled behavioral graphs had their controllers generated using the Berkeley toolset as well as predicted by **PASTA**. Since both the Berkeley toolset and **PASTA** require a description file as their input, it is essential that the two descriptions address control in the same fashion. Thus, both files should

- address the same graph,
- start with the same data path schedule,
- be configured for the same type of control (register versus stage control), and
- (for the Berkeley tools) not be a redundant or poor description.

The first two are obvious; the third requires a decision regarding which control method to adopt. For validation, a variety of designs cover one or the other control method, depending upon which is smaller in area. Finally, the fourth

point highlights a problem encountered during validation: poor descriptions. This issue will be examined more closely.

3.5.1.1 Basic Model

For simple descriptions with no loops or conditionals, comparison between the predicted and actual PLA is reduced to the type of control method employed. Small PLAs tend to favor stage-control as this limits the size of the PLA. As the controller grows, a transition to control of registers is used to maintain the lowest possible area.

Earlier in this chapter, the basic model parameters (inputs, outputs, and product terms) were calibrated against the Berkeley tools. Now the complete control model as implemented in PASTA will be used to validate the basic PLA equations presented earlier. For "stage control", verification is easily accomplished. Since no control of registers is being considered, each PEG description takes on the following form:

```
--
--    Generic state machine (state outputs)
--

OUTPUTS : s1 s2 ... sn;

Start   :  ASSERT s1;
        :  ASSERT s2;
   -- insert states as needed (each colon represents a state)
   -- last state MUST loop back to top of PLA
        :  ASSERT sn;
           GOTO Start;
```

Some additional results and examples are shown in Table 3.4. The first portion of Table 3.4, where type is *Sta*, are the stage-control PLAs. Note that in all cases, the inputs, outputs, and product terms match. The error introduced is strictly due to the PLA area model coefficients.

Table 3.4: PLA Area Estimation: Basic Model

States	Type	Estimate		Actual		Error (%)
		i/o/p	Area (mil^2)	i/o/p	Area (mil^2)	
2	Sta	1/3/2	11.4	1/3/2	11.8	3.5
4	Sta	2/6/4	18.7	2/6/4	18.9	0.1
8	Sta	3/11/8	38.0	3/11/8	38.5	1.2
16	Sta	4/20/16	80.7	4/20/16	82.3	2.0
32	Sta	5/37/32	225.1	5/37/32	226.8	0.1
64	Sta	6/70/64	674.2	6/70/64	684.5	1.5
96	Sta	7/103/96	1402.4	7/103/96	1414.8	0.1
128	Sta	7/135/128	2335.8	7/135/128	2346.3	0.0
4	R=2	2/4/4	16.1	2/4/4	16.1	0.0
8	R=4	3/7/8	31.2	3/7/8	31.0	0.0
16	R=8	4/12/16	60.8	4/12/16	60.7	0.0
32	R=16	5/21/33	159.2	5/21/32	153.1	3.8
64	R=32	6/38/64	421.5	6/38/64	420.0	0.0
96	R=32	7/39/96	656.0	7/39/96	651.6	0.1
128	R=40	7/47/128	978.1	7/47/128	974.8	0.0

The second half of the table addresses register control. A number of PLAs were constructed with an arbitrary number of registers which were randomly asserted (but at least one register asserted per state). The form of the PEG file is as follows:

```
--
--     Generic state machine (register outputs)
--

OUTPUTS : r1 r2 ... rn;

Start  :  ASSERT r? r? ....;
       :  ASSERT r? r? ....;
   -- insert states as needed (each colon represents a state)
   -- last state MUST loop back to top of PLA
       :  ASSERT r? r? ....;
```

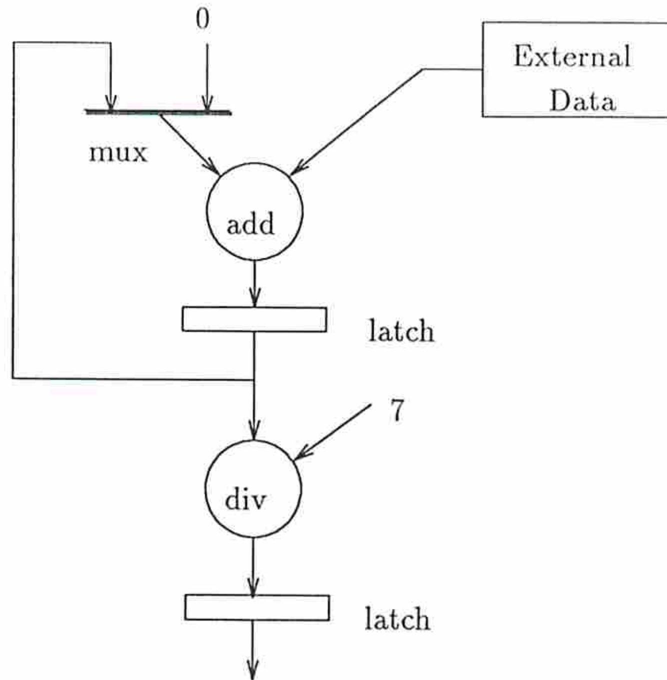


Figure 3.15: Data Path of Averaging Operation

```
GOTO Start;
```

Results for the register control method are even more accurate. However, one anomaly did occur for the 32 state design. In this case, the PLA minimization tools could not reduce the number of product terms to the theoretical estimate; one additional product term was needed in the actual design.

3.5.2 Loops and Conditionals

As part of the PLA model validation for loops and conditional paths, four example behaviors were synthesized manually and compared against the estimates. In addition, a comparison was made against both loop unrolling and an external counter for the loop examples. Two of these behavioral graphs are specific to loops as shown in Figures 3.15 and 3.16 and the others specific to conditionals, with one conditional graph shown in Figure 3.17. Table 3.5 lists the results.

For each of the loop examples, which represent the extremes of register sharing, construction of the PEG control file was slightly different. Note that the

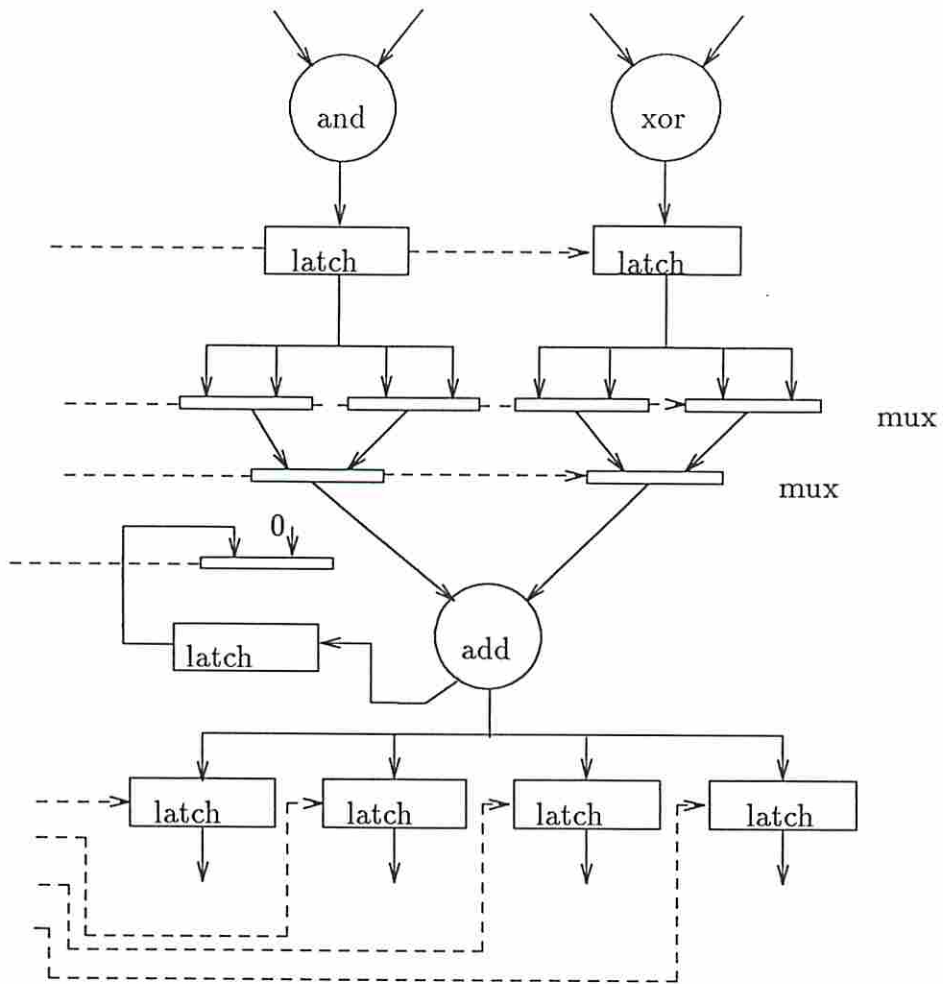


Figure 3.16: Serialized ADD operation

Table 3.5: PLA Area Estimation: Conditionals and Loops

Type	Figure	Impl.	Physical K_{reg}	Control Area (mil^2) Estimate	Actual	Error (%)
Loop	3.15	Unroll	0.0	27.8	27.3	1.8
		Ext. Cntr.	0.0	83.1	85.1	2.4
Loop	3.16	Unroll	1.0	27.8	27.6	0.1
		Ext. Cntr.	1.0	31.4	29.9	5.0
Cond.	not shown	Expand	–	34.8	34.6	0.1
Cond.	3.17	Expand	–	130.1	132.7	2.0

multiplexer control was included for the loop unrolling example, but is not required as a single 3-input AND gate would have sufficed. The PEG descriptions for the larger loop shown in Figure 3.16 are similar.

```
--
--  Loop1 example for unrolling (Figure 3.15)
--

OUTPUTS :  mux1 reg1 reg2;

Start   :  ASSERT mux1 reg1;
        :  ASSERT reg1;
        :  ASSERT reg1;
        :  ASSERT reg1;
        :  ASSERT reg1;
        :  ASSERT reg1;
        :  ASSERT reg1;
        :  ASSERT reg2;
           GOTO Start;

--
--  Loop1 example for hardware cntr (Figure 3.15)
--

INPUTS  :  instat;
OUTPUTS :  dec ld reg1 reg2 val1 val2 val3;

Start   :  ASSERT ld reg1 val1 val2 val3;
Back    :  ASSERT dec reg1;
           IF instat THEN Back;
        :  ASSERT reg2;
           GOTO Start;
```

The **PASTA** input files for the examples shown in Figures 3.15 and 3.16 are fairly simple. (The PASTA notation is described in Appendix E.) Each description starts with an equal sign "=" which indicates the data path states, registers, and optional multiplexer lines for the controller. In the second description, a unique register is controlled during each of the 7 iterations of the control-step loop. (The colon signifies that K_{reg} follows (scaled by a factor of 100); by default, $K_{reg} = 1$.) The last description has a 4-iteration loop with the same registers operated during each loop. Thus, $K_{reg} = 1$ and the value after the loop count is set to 100.

```
#
#  PASTA input for examples
#
# Figure 3.15 unrolled
= 8 2 1
# Figure 3.15 counter
= 2 2
(7:0 * 1) + 1
# Figure 3.16 unrolled
= 6 2 2
# Figure 3.16 counter
= 2 2
(4:100 * 1)
```

Despite the small examples used for designs with $K_{reg} = 0$, errors are within acceptable tolerances. For both designs, the same number of inputs, outputs, and product terms was predicted and measured. Conversely, the error for designs with $K_{reg} = 1$ is higher. In this case, there is additional decode hardware besides the external counter. This area can only be estimated by **PASTA**, whereas a human designer can optimize this portion of the control logic. In a typical design, one would expect that a large dataflow graph would have some loops as well as some loop-free portions. Given the high accuracy of the non-looping portion of the controller model, large examples should improve upon the error shown here.

The designs of Table 3.5 containing conditional branches include a simple 8-node graph with one conditional branch (not shown) and a complex conditional branch shown in Figure 3.17. The PEG descriptions become complex due to the quantity of conditional branches. In these designs, multiplexers were specifically included since that is the way they were built by the circuit engineer.

```
--
-- Simple conditional
--
INPUTS  :  s1;

OUTPUTS :  r1 r2 r3 r4 m1;

Start   :  ASSERT r1;
           IF s1 THEN Lpr;
        :  ASSERT m1 r2;
        :  ASSERT m1 r3;
           GOTO Lpend;
Lpr     :  ASSERT r2;
        :  ASSERT r3;
Lpend   :  ASSERT r4;
           GOTO Start;

--
-- Park conditional by hand
--

INPUTS  :  d1 d2 d3 d4 d5;

OUTPUTS :  r1 r2 r3 r4 r5 r6 r7 m1 m2 m3 m4 m5 m6;

Start   :  ASSERT r1;
           CASE(d1 d3)
             0 ? => d1m;
```

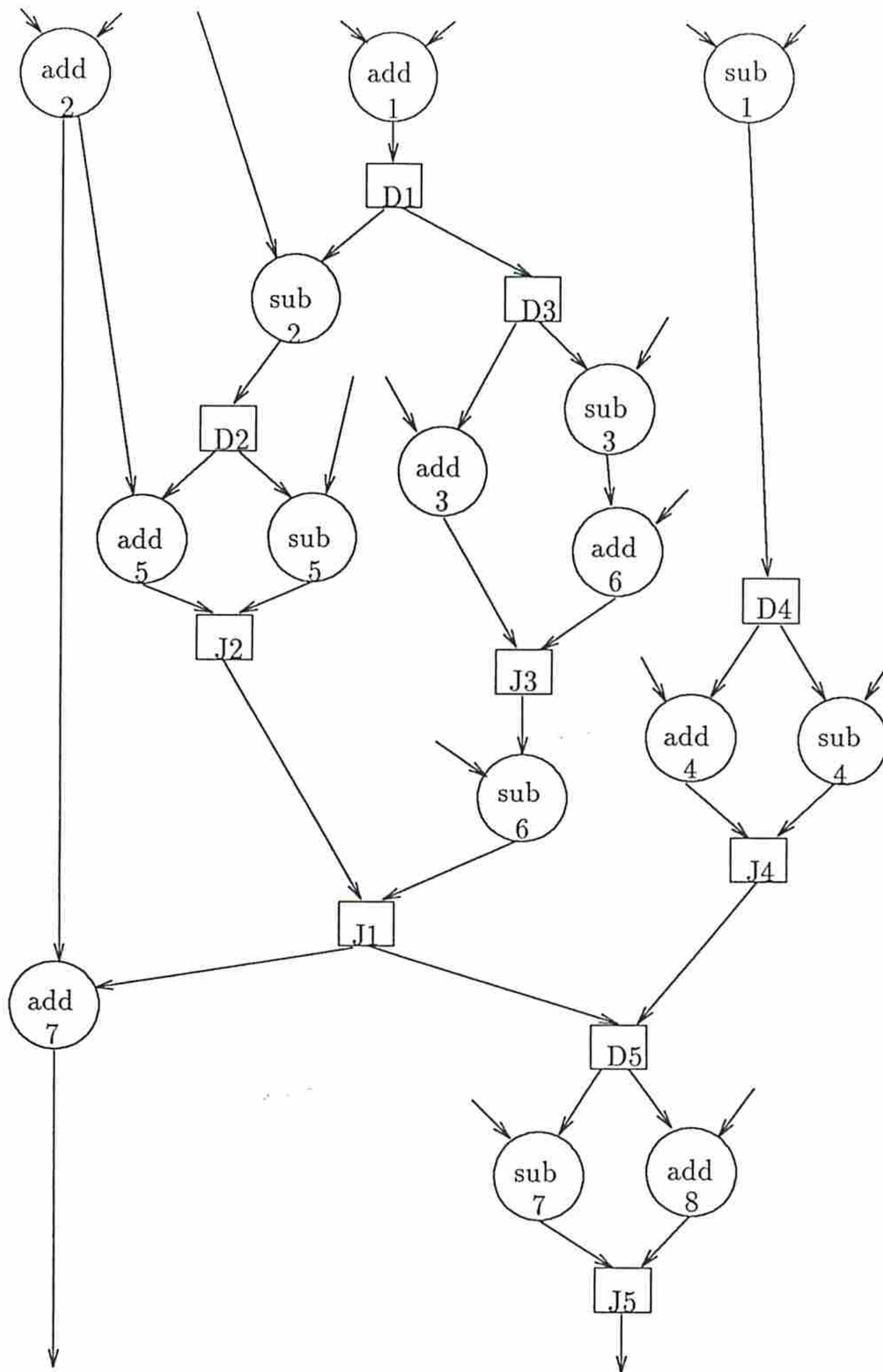


Figure 3.17: Complex Conditional Graph ©1986 N. Park

```
                1 0 => d3m;
                1 1 => d3p;
              ENDCASE => d3p;
d3m       :   ASSERT r2 m1;
          :   ASSERT r3 m3;
              GOTO j3;

d1m       :   ASSERT r2 m1;
          :   ASSERT r3 m6;
              IF d2 THEN d2p;
          :   ASSERT r4 m2 m1;
              CASE (d4)
                1 => d4p;
                0 => j1;
              ENDCASE;

d2p       :   ASSERT r4 m5;
              CASE (d4)
                1 => d4p;
                0 => j1;
              ENDCASE;

d3p       :   ASSERT r2 m1 m4;
          :   ASSERT r3 m2;
j3        :   ASSERT r4 m4 m5;
              IF d4 THEN d4p;
j1        :   ASSERT r5 m3 m1;
              CASE (d4)
                1 => d5p;
                0 => d5m;
              ENDCASE;

d4p       :   ASSERT r5 m6 m4;
```

```
          IF d5 THEN d5p;
d5m     : ASSERT r6 m6 m5;
          GOTO j5;

d5p     : ASSERT r6 m3 m2;
j5      : ASSERT r7 m3 m2 m1;
          GOTO Start;
```

In contrast to the PEG input, the PASTA input is much smaller, being an estimator. The number of states with which an inner loop is controlled is obtained by applying the data path estimator (**PSAD-NP**) to the inner branches after the resources are fixed by **PSAD-NP** for the entire design.

```
#
# Park stuff
#
= 4 4 1
1 + s|{2,2} + 1
= 5 7 6
(a|{(1+b|{1,1}),(c|{1,2})} & (1 + d|{1,1})) + (1 & e|{1,1})
```

There is no error for the small conditional; the inputs, outputs, and product terms match exactly. However, the large conditional has an error associated with it. Although the inputs and outputs matched (9 and 17, respectively), the number of product terms did not (22 versus 23 actual). An examination of outputs showed that the estimated design generated a PLA with 14 states; the real design took 15 states. The extra state is due to the actual schedule which entailed one more state within an inner conditional than predicted.

During the first portion of the validation, details of the individual estimated and actual PLAs has been given. For the remainder of this section, detailed descriptions will be excluded and only the summarized results are presented.

Table 3.6: Summary of Results for Pipelined PLA

Dataflow Graph	Parts	Init. Interval	Inputs		Outputs		P terms	
			Pre.	Act.	Pre.	Act.	Pre.	Act.
Multiplier	4	2	2	2	5	5	3	3
	7	5	3	3	7	7	6	6
AR Lattice Filter	7	3	2	2	5	5	3	3
	16	6	3	3	9	9	6	6
Large Conditional	6	2	11	11	8	8	38	40
	6	3	10	10	7	7	29	30

3.5.3 Pipelined Designs

A comparison of experimental results and predictions for pipelined designs are shown in Table 3.6. The same behavioral descriptions that have been previously used were pipelined using **Sehwa** and a controller manually constructed to operate each design [PP86]. Note that behavior containing inner loops is not allowed in pipelined design. Thus, only sequential and conditional behavior is demonstrated.

3.5.4 PLA Column Folding

The validity of the PLA folding model can be seen by the results contained in Table 3.7. Several of the designs described previously in Table 3.5 were manually folded using the algorithm described in [BB87]. The density adjustment factor determined in [MT86] (0.82) is used for the estimation model. Folding over 30 designs resulted in an average error of ± 1 columns.

3.5.5 Use of PASTA

Although PLA area estimation can be used as a post data path synthesis tool, its primary purpose is that of estimating control area prior to synthesis. In order to accomplish this task, tools which predict the number of states, hardware resources and register requirements are necessary. The number and length of conditional paths, as well as the length of each loop, can be estimated from

Table 3.7: Comparison of Folded PLA Area

Description	States	Columns	Folded Columns	
			Actual	Estimate
AR Lattice Filter	7	9	4	5
	19	11	5	4
Multiplier	4	6	3	3
	6	7	3	4
Large Conditional	21	11	5	4
Average Loop (Unrolled)	8	5	2	2
Serial ADD (Counter)	3	7	1	2

the resources and dataflow graph; loop counts are not known. For practical designs, this is rarely a problem as most loops either have a fixed count or a loop termination test based upon some condition external to the PLA.

When using estimated data path results, the actual scheduling and allocation of each operation are not known. However, using the predictor described here, the control area can be estimated. Two examples illustrate the use of this technique.

For the first example, an elliptical wave filter (a dataflow graph with 36 nodes and 57 edges) was processed first through **SLIMOS** to obtain the best module set [JPP88]. The dataflow graph and module set are examined by the pipeline estimation tool to produce a set of predicted designs [JPP87]. One design having a initiation interval of $l = 7$ has resources of 4 adders and 2 multipliers and a clock of 2950 nS predicted. Finally, a register prediction tool estimated that 14 registers would be needed. Since the number of registers is larger than the number of states, the stage control method is used. The predicted and actual results for synthesized controller and data path are shown in Table 3.8.

For the second example, the large conditional (Park) is revisited. Estimated resources and clock cycle time were generated for an initiation interval of five [JMP88]. Based upon this prediction, conditional path lengths were estimated as their length in operations; these paths were separated into control steps (17). Since only 3 registers are predicted as being needed, register control was used for this method. The results are also included in Table 3.8.

Table 3.8: PLA Area Estimation using Data Path Prediction Tools

Dataflow Graph	Stages		Inputs		Outputs		P terms		Area	
	Pre.	Act	Pre.	Act	Pre.	Act	Pre.	Act	Pre.	Act
Ellip. Fltr	18	14	3	3	10	10	7	7	34.7	33.4
Large Cond.	14	17	10	9	8	7	22	23	102.3	97.9

3.5.6 Shortcomings of Using the Berkeley Toolset

The Berkeley toolset offers a complete and robust solution to the synthesis of PLA controllers. Unfortunately, there are several shortcomings to this package found in PEG, EQNTOTT, and ESPRESSO. PEG is the front-end tool which accepts the user-specified control description and generates the state equations. EQNTOTT accepts the state equations produced by PEG, outputting an unoptimized PLA personality matrix. ESPRESSO reads this personality matrix and minimizes the number of product terms.

One shortcoming (or feature) of PEG is that the controller must be completely specified by the user, but the user has considerable freedom in expressing equivalent behavior. PEG will merrily generate useless descriptions as easily as excellent ones; the primary driver is the input control file. Unfortunately, this could result in a large discrepancy between PEG and **PASTA** control area estimates. For example, the following is a description in PEG:

```
OUTPUTS:
    register_1_WRITE
    register_2_WRITE
    register_3_WRITE
    register_4_WRITE
    register_5_WRITE
    register1_1_WRITE;

Start  : ASSERT register_1_WRITE register_2_WRITE register_3_WRITE;
State1 : ASSERT register_2_WRITE register_4_WRITE register_5_WRITE;
```

```
State2 : ASSERT register_4_WRITE register_5_WRITE register1_1_WRITE;
State3 : ASSERT register_1_WRITE register_4_WRITE register_5_WRITE;
State4 : GOTO Start;
```

The minimized PLA resulting from this description (after executing PEG, EQNTOTT, and ESPRESSO) has 3 inputs, 9 outputs, and 4 product terms for a total area of 25.3 mil^2. The comparable **PASTA** input file specifies 5 states and 6 registers and gives 3 inputs, 8 outputs, and 5 product terms for a total area of 27.7; an error of 10%. Although the area is close, one would expect the product-term and output counts to match.

The difference in output lines is due to the method by which the PLA is controlled. The PEG description is controlling six registers in five control steps. Conversely, **PASTA** uses the stage control method thereby saving one output line. (In this example, three external 2-input OR gates will be required for the registers; however, the total control area would still be larger using register control.)

Besides a difference in output lines, the product-term counts are dissimilar. This is caused by the State4 description where nothing is happening. Although this could be an idle state, it is likely the user meant to loop the description at the end of *State3*. The PLA controller is now reduced by one state giving 2 inputs, 8 outputs, and 4 product terms. An adjustment in the **PASTA** input file and forcing register control yields the same PLA parameters. In this case, **PASTA** gave an area of 21.6 mil^2 versus the 21.3 mil^2 actual, an error of only 1.4%.

Finally, in this example, there is a pair of redundant register control lines: *Register_4_WRITE* and *Register_5_WRITE*. (This is why the earlier OR gate estimate was three instead of four.) Unfortunately, since the Berkeley tools have PLA interior modules which use pairs of lines, reducing the output line count from 8 to 7 does not have any effect in this case. However, the net effect of all changes has reduced the PLA area by 19% which would be even higher for single-line PLA blocks. It is clear that careful attention must be paid to the

control description; a poor one will result in a more costly PLA to carry out the same functions.

Another unfortunate consequence of the PEG description language is its inability to accept "don't care" output signals. All output lines produced by PEG are positively controlled. An output line that is not asserted is deasserted. A "don't care" state occurs with some frequency for both registers and multiplexers. These "don't cares" could be exploited to reduce the PLA density and, possibly, the number of product terms. Lower PLA density might allow increased folding and a further reduction in PLA area.

The last feature lacking in the toolset is the ability to recognize redundant output lines and combine them. In particular, it is easy to produce PLAs with identically controlled output. A useful extension would be for EQNTOTT or ESPRESSO to combine these outputs. This would be even more critical for reducing the PLA size when output lines have many "don't care" states. Many examples have been encountered which exhibit redundant output control lines.

3.6 Summary

In this chapter, a method for estimating the area of a PLA controller was presented. The model encompasses common controller design considerations such as loop control, conditional branches, and design style selection (non-pipelined versus pipelined architecture). In addition, the model was extended for estimating the area of folded PLAs. Inputs to the model are values obtained during data path synthesis such as operator allocation and scheduling.

Another more important application of PASTA - that of high-level area estimation - was also demonstrated. The results suggest that this control area predictor, using the results of data path estimation tools, is capable of providing the designer with accurate chip design area which includes both the control path and data path. Future efforts involve integrating the PLA area estimator with these high-level data path estimators of scheduling and allocation, as well as register and multiplexer models, into a large system tradeoff tool. Estimation techniques could then be used to accurately describe the shape of the

design space in area and time with minimal execution time. Synthesis would be reserved for local exploration to find the best design in the chosen region.

Chapter 4

Predicting Register and Multiplexer Requirements

4.1 Introduction

An RTL design consists of hardware modules connected via registers, multiplexers, and wires. Buses could also be used in place of multiplexers. The design space is characterized by the area and time parameters of a design; parameters such as power consumption and temperature could also be used. Currently, prediction techniques exist for area/time (AT) characterization using operator area/delay only [JPP87] [JMP88]. These estimates do not include register and multiplexer/bus area which have been shown to have a measurable impact on the AT curve [GP82].

Prediction of register and multiplexer usage requires knowledge about the data path. The inputs available to such prediction models consist of

- the behavioral dataflow graph,
- the module set including the number of each type of operator,
- the number of partitions into which the dataflow graph is divided, and
- the initiation interval (for pipelined designs only).

Estimating bus requirements is a more difficult problem. The number of bus drivers can be determined similar to multiplexer estimation. However, this value

is also influenced by the number of available buses and tradeoff between buses and multiplexers. Bus and driver area estimation is not addressed in this thesis.

In this chapter, methods for accurately predicting register and estimating multiplexer area are described. Upper and lower register bounds are derived for non-pipelined and pipelined designs; these bounds are dependent upon the dataflow graph and, for pipelined designs, the partitions and initiation interval. An accurate model for predicting register use from these bounds is described. The model derived for multiplexer area also requires operator and register usage in addition to the register model parameters. Finally, these models are validated against synthesis tools.

4.1.1 System to be Modeled

The following is a description of the system to be modeled.

- Behavior is described in the form of an *acyclic dataflow graph.* Nodes represent operations and edges represent values. A given dataflow graph contains one source that has no incoming edges (henceforth called *root*) and one sink which has no outgoing edges (*outport*). An example dataflow graph is shown in Figure 4.1; all incoming edges not tied specifically to another node are implicitly connected to *root*. All outgoing edges not specifically tied to another node are implicitly connected to *outport*.

- An *edge* begins at one node (called the *source*) and ends at a different node (*sink*).

- Between any two nodes, there is a maximum of one edge. Multiple edges between two nodes are merged into a single edge which combines the separate values along the original edges into a new value.

- *Root* can only be a source node; *outport* can only be a sink node.

- Conditional branches are allowed within the dataflow graph. They may be nested to any depth, and have no restriction on fanout. Special nodes, *dist* and *join*, respectively describe the beginning and end of a multi-way

conditional branch. All paths from a given *dist* must pass through the matching *join*.

- A module library consists of components that can implement the operations in the dataflow graph. Components have area, propagation delay, and other physical attributes associated with them. A module set is a subset of the module library used for a specific design. The module set is complete in that every operation in the dataflow graph has one (and only one) operator type taken from the module library which is used for implementation.

Other terms used throughout this chapter are described here.

- A *path* is a sequence of **one or more** edges $(e_1, e_2, \ldots, e_n)$ such that the sink of e_i is the source of e_{i+1}.
- A *conditional path* is one which starts at a *dist* node and ends at the matching *join* node.
- A *complete path* is one which starts at *root* and terminates at *outport*.

4.2 Register Area in Non-pipelined Designs

To estimate the number of registers, knowledge of both the theoretical upper and lower register bounds are needed. Theoretical register limits can be determined for non-pipelined designs strictly by examining the dataflow graph. More correctly, one must examine all possible partitions of this dataflow graph. A *partition* is any edge cutset that separates *root* and *outport*; an example partition is shown in Figure 4.1. A scheduled dataflow graph will have one or more partitions which, when the partition lines are drawn in order on the dataflow graph (from the first on up), may partially overlap but will never fully intersect since values on edges only propagate forward in the graph. Physically, a partition distinguishes different clock cycles of the scheduled graph.

Each partition in a dataflow graph is comprised of a set of edges and each of these edges has a value associated with it. A register is required for each *unique* value that must be retained past the clock cycle in which it is generated. Thus,

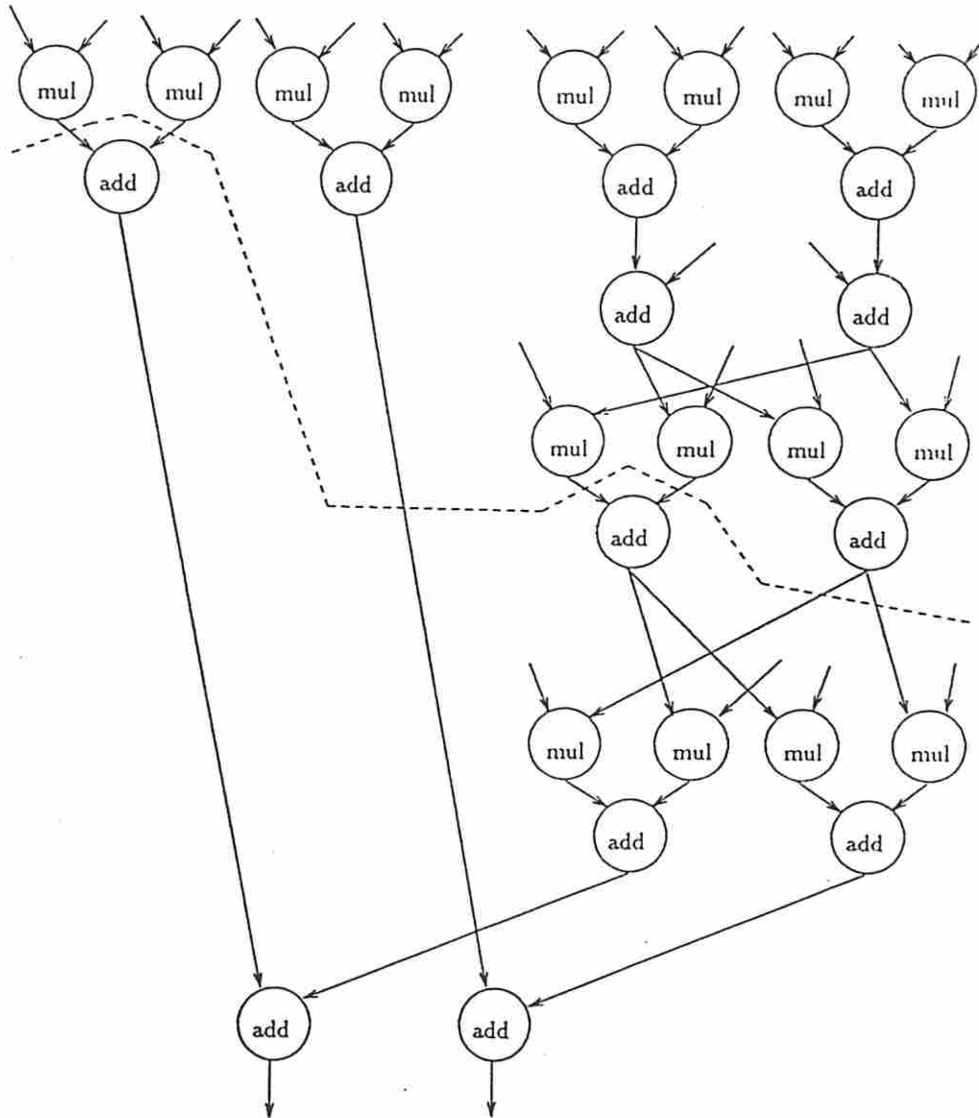


Figure 4.1: AR lattice filter showing cutset

each edge within a given partition cutset signifies potential register use. The actual number of registers needed for a given clock cycle is determined by four effects:

- For non-pipelined designs, **it is assumed that no registers are needed on any incoming edges from *root* as these values are stored externally.** Edges which have a source of *root* are therefore deliberately excluded when considering register use for non-pipelined designs.
- Conversely, it is also assumed that registers are required on all unique outgoing edges. (These two constraints reflect register assignment used by the validating synthesis tools.)
- A unique value generated by an operation may be present on one or more edges. Clearly, the value only need be stored once regardless of how many edges hold the value in a partition cutset.
- If two values in the cutset occur on different conditional branches, only one of these values is actually active during that clock cycle. These mutually exclusive values are thus able to share the same register.

The assumptions made regarding which of the graph incoming or outgoing edges are eligible for register assignment reflect operation of the current USC synthesis tools. Where incoming and/or outgoing values are used (V_{in} and V_{out}), the equations which will be given are readily modified to reflect assignment of registers to V_{in} and/or V_{out}.

By examining all possible partitions, theoretical bounds can be established upon the minimum (R_{min}) and maximum (R_{max}) number of registers necessary for non-pipelined designs without performing synthesis. Essentially, the partition with the maximum number of edges is the cutset which defines the maximum number of registers needed. Conversely, the partition with the minimum number of edges is the smallest cutset which defines the minimum number of registers. (The actual size of a given cutset is not that simple, but can be determined using the assumptions just presented.) Estimating the number of registers for non-pipelined designs (the quantity Rnp_{est}) from these limits is complicated by the fact that only a subset of these partition cutsets is observed in actual designs.

4.2.1 Register Bounds in Non-Pipelined Designs

Analysis of a dataflow graph to determine minimum and maximum register limits can be treated as a network flow problem. An algorithm which derives the upper bound on the number of registers using network flow has been published by Kurdahi [Kur87].

In network flow analysis, each edge of the dataflow graph is first assigned a non-negative value which defines the minimum (or maximum) flow (akin to oil pipeline flow or electrical current) allowed through that edge. A *legal network flow* is one where a non-negative integer flow value is assigned to each directed edge such that

1. the flow on each edge is at least at its minimum (or at most at its maximum),
2. the total flow into each node (excepting *root* and *outport*) equals the total flow out of that node, and
3. the total outgoing flow from *root* is equal to the total incoming flow to *outport*.

When flow through an edge exceeds its minimum, it is said to have *excess flow*. (Conversely, when flow is less than its maximum, it has *excess capacity*.)

In Kurdahi's algorithm, a minimum flow constraint of one is assigned to every edge. Then, the graph is initially seeded by adding flow along different *complete paths* until a legal network flow is achieved. Since this seeding usually results in *excess flow* along one or more edges, an attempt is made to reduce flow along complete paths until the minimum flow is reached.

Reducing total network flow to the smallest legal network flow possible can be accomplished using a *min flow, max cut* algorithm. Algorithms for determining *min-flow, max-cut* of a dataflow graph are detailed in Even [Eve79]; Dinic's method was chosen by Kurdahi. After an initial flow is seeded, the Dinic method finds the minimum flow by first forming a *layered network* to determine if any *complete path* (from *root* to *outport*) can have its flow reduced, and then reducing it if one exists.

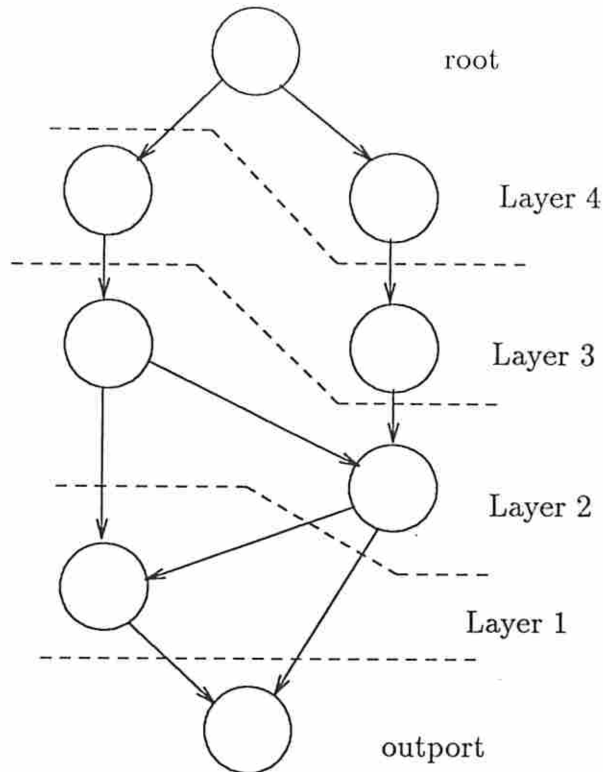


Figure 4.2: Example layered network

Formation of the *layered network* entails finding a complete path along edges with *excess flow*. By reducing flow only along edges with excess flow, it is ensured that a legal network flow will be retained. For construction of the first layer, a set of all nodes is constructed which have flow to *outport* along an edge with excess flow as shown in Figure 4.2. (All edges are assumed to have excess flow in this example.) Each successive layer consists of nodes having edges with excess flow to any of the previous layers.

At some point, no further layers can be formed. If the last layer reached contains *root*, then one or more complete paths have been found where flow can be reduced. In this case, the flow is reduced along one path and another attempt to reduce flow is made, starting with formation of the layered network. If *root* was not reached, there is no such path indicating that minimum flow has been achieved. The total network flow at this point gives the *Kurdahi predicted upper bound on the number of registers*, R_{kur}.

Kurdahi's approach gives an exact solution when all edges in the dataflow graph represent a unique value. If the same value resides on multiple edges leaving the same node, or conditionals exist in the graph, this approach instead yields a result which is greater than the exact value. A revised method will be presented which accommodates these more general dataflow graphs to provide an improved solution.

In Kurdahi's algorithm, the lower bound on flow through the dataflow graph is one unit for every edge. However, if a given node has more than one outgoing edge where each edge has the same value, only the *total* flow among these edges need be one (as only one register is required); the flow along a single edge might be zero. This problem is resolved by locally transforming the node to reflect the proper flow constraints and applying *eligibility* rules to the edges.

Eligibility determines which edges emanating from a node are suitable for register assignment. Essentially, if two edges have the same value (which implies they are from the same node) and one edge has a lifetime that *always* meets or exceeds the other, then the edge with the shorter lifetime is *ineligible* for register use. The longer lifetime eligible edge is said to *dominate* the ineligible edge lifetime. Ineligible edges emanating from a node cannot be used to satisfy the *minimum flow* through that node.

Definition 4.1 Eligible edges: *Given a node n in a dataflow graph with outgoing edges $E(n) = \{e_1, e_2, \ldots\}$, $e_i \in E(n)$ is* ineligible *for meeting the minimum flow through n if*

1. *there exists an edge $e_j \in E(n)$ containing the same value as e_i (which means that e_i and e_j have the same source, but must have different sinks), and*

2. *there is a path from the sink (or destination) of e_i to the sink of e_j.*

Any edge which is not marked ineligible is eligible *to meet the flow requirements through n. Clearly, any edge which is the sole outgoing edge of a node is automatically* eligible. *(See Figure 4.3).*

As an example, a dataflow graph with eligible edges highlighted is included in Figure 4.3. Assume that both edges from $s1$ hold the same value; thus, either

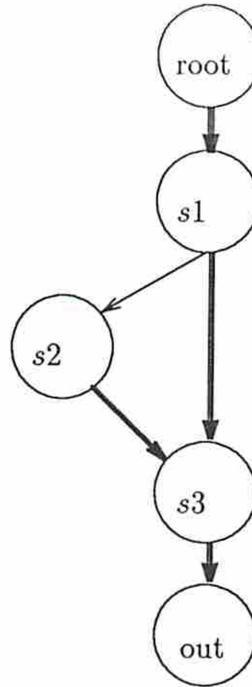


Figure 4.3: Example figure highlighting eligible edges

edge might be assigned a register. However, assigning a register to edge $s1 - s2$ is not helpful since the value from $s1$ must be active until $s3$ completes. Edge $s1 - s3$ dominates value lifetime making $s1 - s2$ is ineligible to store the value produced at $s1$. As a result, there is no minimum flow constraint on $s1 - s2$.

At most nodes, two or more edges with the same value may be eligible. Only one of these eligible edges need have a flow (register assignment) to satisfy the legal network flow. Which edge is assigned a register in an actual design is not known.

To reflect the eligibility constraints, each node which outputs a value along multiple edges is transformed into a subgraph. The algorithm which assigns flow lower bound and transforms nodes with multiple edges is shown in Figure 4.4; it replaces the unity flow lower bound constraint of Kurdahi. Node transformation is performed as needed so that the *min flow, max cut* Dinic method described in the original method can be used. This new procedure produces the modified maximum number of registers, $\tilde{R}_{max}$.

```
procedure low − bound − assign
begin
  Mark all nodes in G(N, E) as "not visited"
  For each node n ∈ G(N, E) marked as "not visited"
  begin
    For each outgoing value v from node n
    begin
      If v is assigned to multiple edges
      begin
        Add no-op node n'_v into dataflow graph below n
        Add edge between n and n'_v; assign flow lower bound of one
        Mark node n'_v as "visited"
        For each edge e having value v
        begin
          If edge e is eligible
          begin
            Move source of e to n'_v
          end
          Assign flow lower bound of zero to e
        end
      end
      Else assign flow lower bound of one to the single edge
    end
    Mark node n as "visited"
  end
end
```

Figure 4.4: Algorithm for DFG transformation/flow lower bound assignment

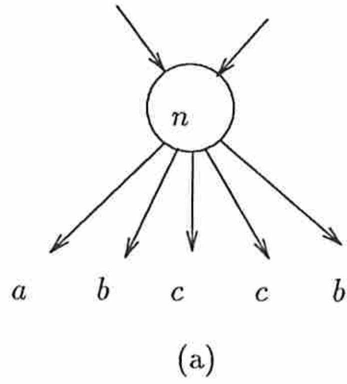
The effects of this procedure upon the dataflow graph are shown in Figure 4.5(a) and (b). In Figure 4.5a, node n has generated three values among five edges. Excepting the middle c valued edge, all edges are eligible.

After applying the transformation, the graph shown in Figure 4.5b is produced. For each value v which originally resided on multiple edges in Figure 4.5, a new no-op node n'_v is introduced which has an edge from n (flow constraint of one) and the original *eligible* multiple edges with value v have their sources tied to this new node. Ineligible edges are not moved, as shown by the one edge with a value of c connected to n. The flow constraint along all constructed edges which have not been assigned are set to zero. This transformation ensures that at least one edge for each value has a flow, while excluding ineligible edges from having a flow requirement.

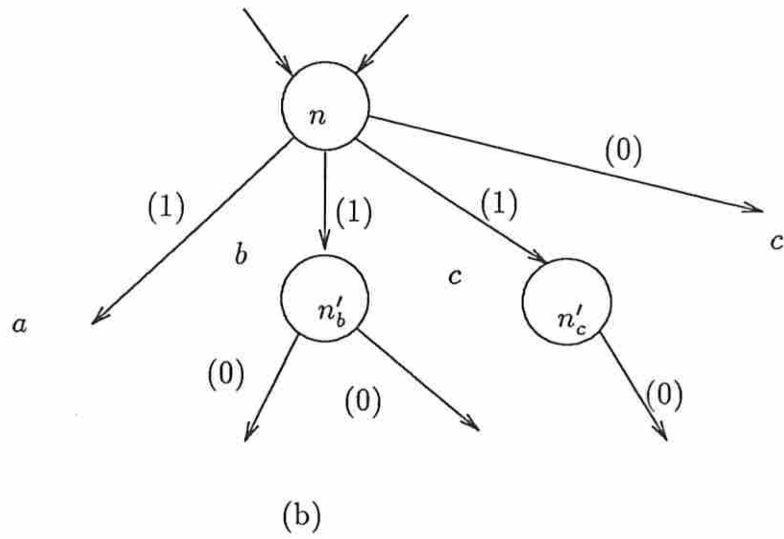
Determining register requirements for a general dataflow graph with conditional branches entails building outward from the innermost conditional. When a dataflow graph contains conditionals, *mutually exclusive subgraphs* off the same conditional branch are compared and the subgraph exhibiting the largest number of registers is retained while the others are removed. (A *mutually exclusive subgraph* is a connected graph with a starting source node of *dist* and terminating sink node of the matching *join* along one of the conditional branches of *dist*.) Repeatedly applying this transformation starting with the innermost conditional alters the dataflow graph into one without conditionals while retaining the register requirements. A conditional dataflow graph and its unconditional transformed view are displayed in Figure 4.6.

Collectively, conditional removal and dataflow transformation/lower bound assignment produce a graph that can have flow seeded and Dinic's algorithm applied as Kurdahi described in his register algorithm. Then, the total minimum network flow realized is the maximum number of registers needed, $\tilde{R}_{max}$, if every value in the graph requires a storage register and assuming this register is optimally shared. Hence, the estimated maximum number of registers needed for non-pipelined designs, Rnp_{max}, is

$$Rnp_{max} = \tilde{R}_{max} \tag{4.2.1}$$



(a)



(b)

Figure 4.5: Transformation of dataflow showing lower bound on flow

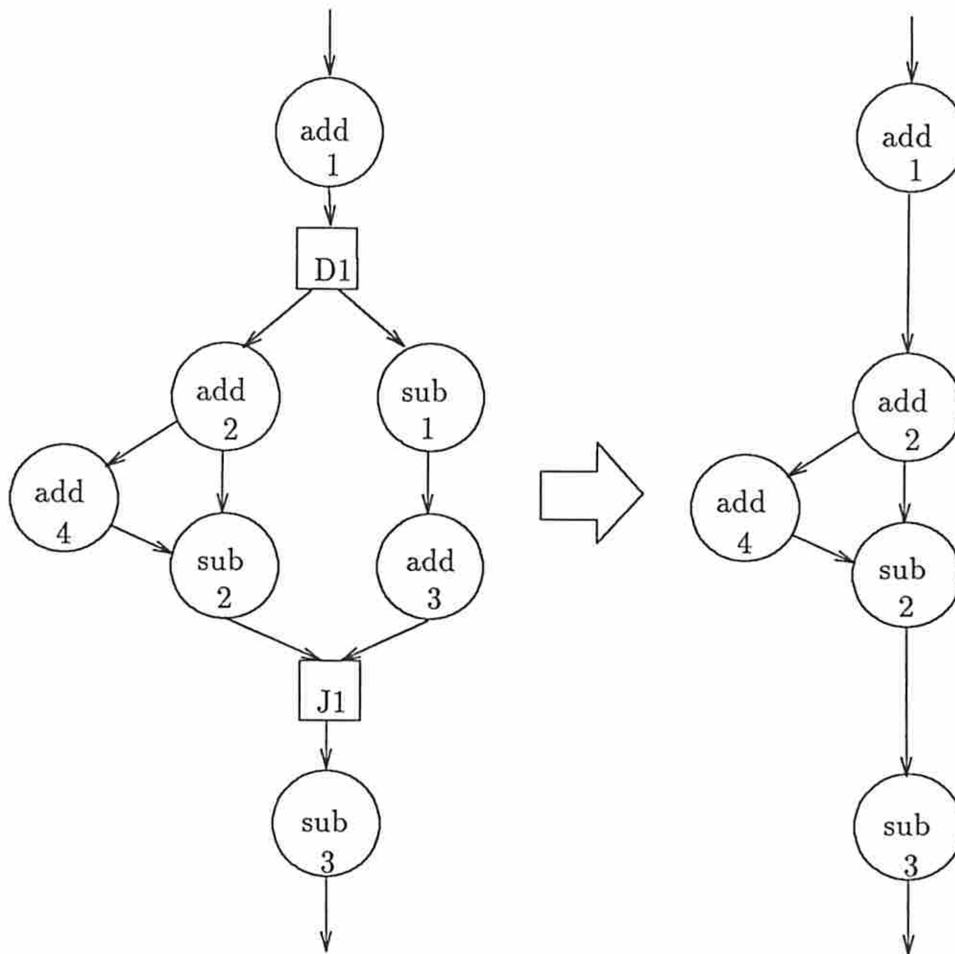


Figure 4.6: Transforming conditional graphs for register computation

Dataflow Graph	Nodes	Edges	Computed maximum		Actual Maximum
			Rnp_{max}	Kurdahi [Kur87]	
AR filter	30	52	8	12	8
FIR filter	25	47	8	8	8
Elliptical filter	36	57	8	15	12
HAL example [PK87]	12	23	5	8	5
Random	28	57	11	13	11
Multiplier	11	23	5	8	5
Criss-cross	8	13	2	4	3

Note: Kurdahi's algorithm was adjusted to ignore register cutsets along incoming edges.

Table 4.1: Comparison of register requirements: non-pipelined designs

The procedure for finding $\tilde{R}_{max}$ does not distinguish between edges having the same value when both are eligible. This simplification can lead to underestimating the maximum number of registers when "dependent register assignment" occurs. Depicted in Figure 4.7, this condition arises when two or more nodes, each having two or more outgoing eligible edges containing the same value, have identical outgoing paths. Dependent upon the graph scheduling, an additional register may be required.

In the example, both $a2$ and $a3$ have edges to $a4$ and $a5$. Using the algorithm, this criss-cross dataflow is transformed into that shown in Figure 4.8; numbers given are the minimum flow bound. All edges are eligible giving a register value (minimum flow) of two instead of the actual three. Assuming a flow of one from $a2$ to $a4$ and from $a3$ to $a5$, an additional register is needed for the *register dependent* edge from $a3$ to $a4$ (or $a2$ to $a5$) to accommodate the worst case cutsets of $\{(a2, a4), (a3, a4), (a5, a6)\}$ or $\{(a2, a5), (a4, a6), (a3, a5)\}$. In the dataflow examples encountered, few exhibit dependent register assignment as the results in Table 4.1 indicate. Hence, resolving dependent flow was not considered.

Determining the minimum number of registers required for non-pipelined designs, Rnp_{min}, is trivial; one simply counts the number of edges with unique values having a sink of *outport*. Hence,

$$Rnp_{min} = V_{out} \tag{4.2.2}$$

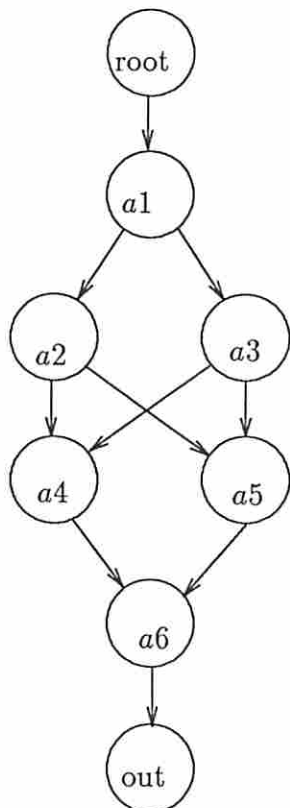


Figure 4.7: Criss-cross

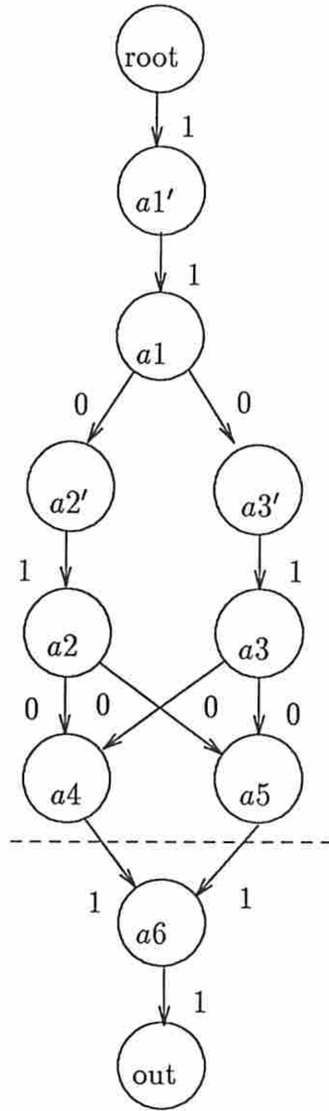


Figure 4.8: Modified criss-cross marked with lower bound on flow

Equation 4.2.2 yields the minimum number of registers for non-pipelined designs since by our definition:

- registers are *always* required after the last operation(s) in the dataflow graph, and
- any smaller register cutsets in the dataflow graph are dominated by the required registers at the *outport*.

4.2.2 Estimating Register Use in Non-Pipelined Designs

Although register bounds are readily determined for non-pipelined design, the count for a specific design could fall anywhere between. A method for predicting register usage for any given design is important, particularly for larger designs where these bounds may be widely separated. The approach presented is based upon empirical observations on the number of registers needed in any non-pipelined design. The estimate of the number of registers in non-pipelined designs, R_{stab}, is dependent upon the register bounds described earlier as well the topology of the dataflow graph.

Terminology which will be used throughout the remainder of this chapter reflect the dataflow graph and resources assigned to it. $\grave{n}_i$ is the number of operations of type i in the dataflow graph. n_i is the *effective* number of operations of type i as defined by Jain [JPP87]. Essentially, n_i is the maximum number of resources of type i needed for the most parallel design. Dataflow graphs with no conditionals have $n_i = \grave{n}_i$; otherwise, n_i may be less than $\grave{n}_i$.

The dataflow graph "shape", as measured by the total number of nodes versus its length from *root* to *outport* L (in nodes), gives the *effective* number of expected registers per partition as

$$R_{eff} = \frac{1}{L} \sum_i n_i \tag{4.2.3}$$

Actually, one must also consider operations which generate more than one unique outgoing value. However, this clouds the derivation without any benefit. The computer program which implements the algorithm determines the actual number of outgoing values. **Note that L never includes nodes having**

no delay or area such as *dist*, *join*, *root*, and *outport*. *Dist* and *join* only propagate values; they do not generate any new ones. Per the register assignment assumption, *root* values are not considered and *outport* does not produce any values.

If R_{eff} is small compared to Rnp_{max}, the graph is relatively long (measured from *root* to *outport* in nodes) and narrow (fewer nodes in parallel). Hence, the registers needed for a given design should be closer to Rnp_{min} as only a small number of edge cutsets or partitions are likely to have a large cardinality. Conversely, as R_{eff} approaches Rnp_{max}, the graph becomes much more squat and wide; an increasing number of cutsets would exhibit the larger cardinality and Rnp_{max} registers would be needed. A linear approximation between these limits gives

$$R_{stab} = \frac{\sum_i n_i}{L \times Rnp_{max}}(Rnp_{max} - Rnp_{min}) + Rnp_{min} \tag{4.2.4}$$

The estimated value for the number of registers in non-pipelined designs, Rnp_{est}, is determined by R_{stab} except for the case where the number of partitions p is one. Here, the value is known to be Rnp_{min}.

$$Rnp_{est} = \begin{cases} Rnp_{min} & \text{if } p = 1 \\ R_{stab} & \text{otherwise} \end{cases} \tag{4.2.5}$$

In the program **REGEST**, which implements the register prediction algorithm, non-pipelined design register count and area are computed for the specified graph. Both Rnp_{min} and R_{stab} are printed. In addition, **REGEST** computes the average bitwidth at the node outputs in the graph and estimates the cost in register-bits.

The AR filter serialization versus register usage graph in Figure 4.9 is constructed from Table 4.2. Model results are compared against the register allocation of REAL [KP87] (which uses the non-pipelined designs synthesized by **MAHA** [PPM86]). Additional examples are analyzed later in this chapter.

Partitions	Registers	
	Actual	Estimate
1	2	2
3	4	5
4	5	5
7	6	5
8	6	5
9	6	5
14	5	5
17	7	5
18	7	5
19	6	5

Table 4.2: Comparison of AR Filter estimated and actual register use

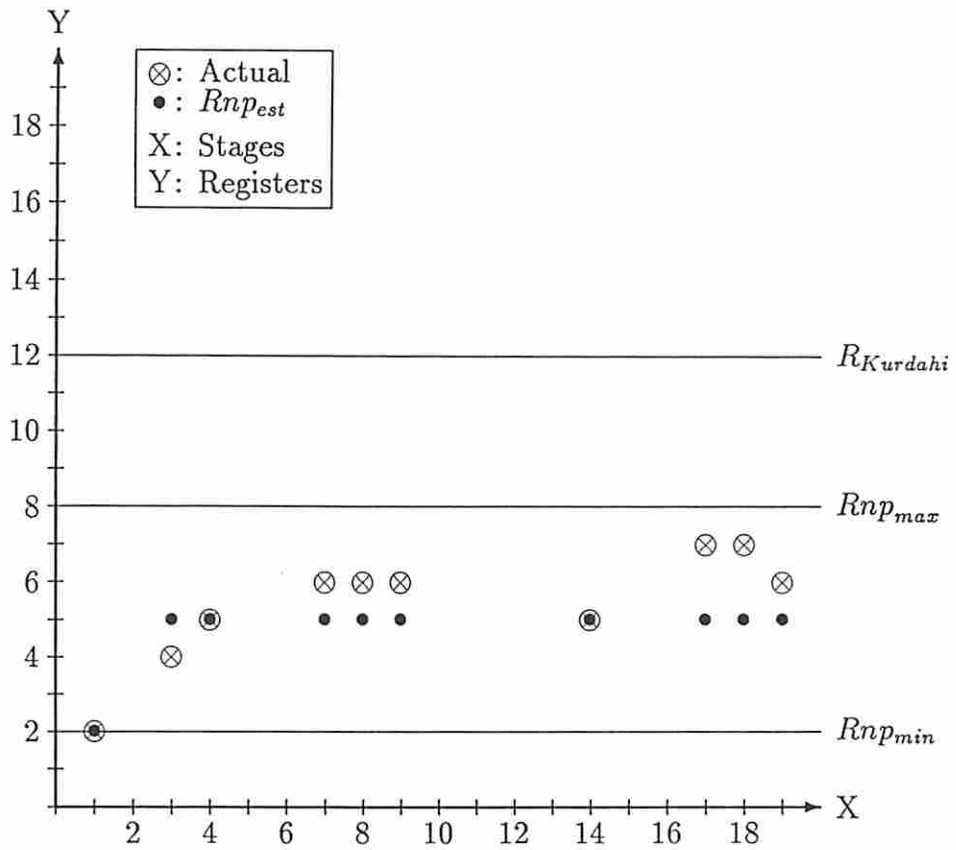


Figure 4.9: Register values for AR lattice filter

4.3 Register Area in Pipelined Designs

Computation of register requirements for pipelined designs is an extension of the non-pipelined results. In non-pipelined designs, the value assigned to a given edge is fixed until the output values are latched at the end of the p-th (last) clock cycle. However, in pipelined designs, the value stored along an edge is only stable for at most the initiation interval, l, where $l < p$, before it is either transferred to another register or discarded. (Non-pipelined design can be considered a special case of pipelined design where $p = l$.) Whereas a single edge could never have more than one register in a non-pipelined design, edges in pipelined designs may have more if the value crosses microcycle boundaries (described shortly).

Another major difference between pipelined and non-pipelined designs is that the dataflow graph input values are latched instead of the output values (per the USC synthesis tools used for validation). This alters the restriction described for non-pipelined designs which not only demonstrates the flexibility of the model, but allows comparison against the pipeline synthesis tools which assign registers to the dataflow graph inputs.

4.3.1 Register Bounds in Pipelined Designs

A pipelined design consists of two or more *microcycles*, u. Each *microcycle* is comprised of l consecutive partitions in the dataflow graph, excepting the last microcycle which may have less than l. What distinguishes pipelined design from non-pipelined design is that these microcycles all operate synchronously in parallel. In other words, the first clock cycle or partition of all microcycles execute simultaneously followed by the second clock cycle of all microcycles, and so forth, until the l-th clock cycle is completed, at which time the procedure repeats.

The tremendous impact of pipelined design upon register count can be realized by examining Figure 4.10. (All edges with no source shown are assumed to be connected to root with all the left-hand-side incoming graph edges of this type taking one value and the right-hand-side taking another value.) Here, a

graph has been divided into two microcycles (indicated by the triple line) with an initiation interval of three for a total of six partitions or clock cycles. In the first clock cycle, *add2*, *add3*, *add4*, and *add5* generate their values at the same time. In the second, all the multipliers operate. In the last clock cycle of each microcycle, the remaining *add* operations occur, two values are saved across the microcycle boundary, and an output is generated. At this point, the procedure repeats with *add2* and *add4* using a new set of data and *add4* and *add5* using the last output of *adda* and *addb*.

A non-pipelined design would only require two registers. However, for this pipelined design, eight registers are required:

- Two registers are needed for the incoming root values along the left and right edge.
- Two registers are needed within the first microcycle; these registers are shared over the three clock cycles.
- Two registers are needed in the second microcycle. Since this microcycle operates in parallel with the first, it cannot share the registers used the first microcycle.
- Similarly, two registers are needed to retain the old incoming *root* values during the second microcycle as the first microcycle *root* registers now have a new value assigned.

Although the total number of clock cycles or partitions, p, is simply $u \times l$ in this example, the last microcycle could have less then l clock cycles. The parameters which are provided are p and l. The number of microcycles is computed as

$$u = \left\lceil \frac{p}{l} \right\rceil \tag{4.3.6}$$

One feature of pipelined designs is that registers are needed on all incoming values, V_{in}, of the dataflow graph. Another feature is that one edge may have more than one register dependent upon whether it crosses a microcycle boundary. For the first microcycle, the number of registers needed is the maximum

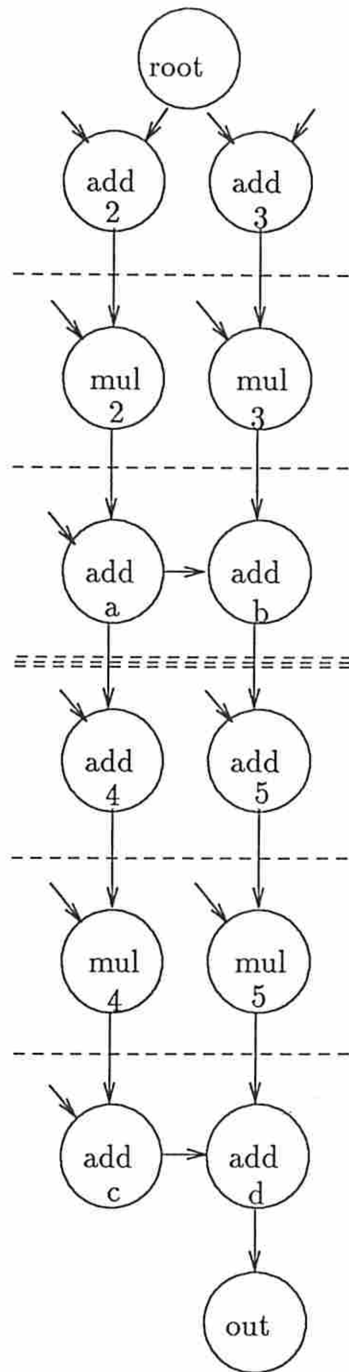


Figure 4.10: Graph depicting pipelined scheduling

of V_{in} and Rnp_{max}. Remaining microcycles would each need the register estimate derived for non-pipelined designs giving the estimated maximum number of registers for pipelined designs, Rp_{max}, as

$$Rp_{max} = \max(V_{in}, Rnp_{max}) + (u-1)Rnp_{max} \tag{4.3.7}$$

Determining the lower bound on estimated number of registers for pipelined designs, Rp_{min}, is similar to the method used to compute Rp_{max}.

$$Rp_{min} = \max(V_{in}, Rnp_{min}) + (u-1)\tilde{R}_{min} \tag{4.3.8}$$

$\tilde{R}_{min}$ is computed using a dual of the Dinic algorithm for $\tilde{R}_{max}$ described in the previous section. There are several differences to the heuristic:

- The flow constraint assigned to each edge is the maximum rather than the minimum allowed.
- Any edge with a zero constraint for finding $\tilde{R}_{max}$ is replaced by the value for $\tilde{R}_{max}$ or some higher arbitrary number. These edges do not constrain flow.
- The network is not seeded, or rather, the initial network flow is zero.
- The layered network is built *forward* starting with *root* and workings towards *outport* through edges with *excess capacity*.
- If a complete path exists through the layered network, flow is added along this path and the process repeats with a new layered network.

When the maximum flow is reached, the total network flow is the minimum number of registers needed in the graph or $\tilde{R}_{min}$.

4.3.2 Estimating Register Use in Pipelined Designs

Estimating register area for pipelined designs is complicated by the partitioning of the design into microcycles. Each microcycle can be treated, in isolation, as a non-pipelined subgraph. And, as will be shown, only the microcycle count is

significant to estimate pipelined design register usage; individual partition and initiation interval differences are secondary.

One feature of *non-inferior* pipelined designs is that the initiation interval and the number of stages are closely related; as the number of stages in a non-inferior design space increases, so does the initiation interval as shown in Figure 4.11. (*Non-inferior* designs are the subset of all synthesized designs that are superior or equal to comparable designs; quality is measured in area and time.) As in the non-pipelined case, increasing the number of stages boosts the register quantity needed towards some limit. Thus, one would expect a dataflow graph partitioned into a large number of stages to exhibit register usage related to this limit (times the number of microcycles).

Conversely, as initiation interval increases, the expected register count declines. When initiation interval is low, there are few stages where register sharing can occur and the register limit is again reached. As initiation interval increases, the ability to reuse registers rises. (Generally, register sharing only occurs during the same microcycle.)

Given the inverse effects of initiation interval and partition increase upon the number of registers, the prediction model assumes that only the number of microcycles has an effect on the register count. The actual values for initiation interval and stage count induce secondary effects that tend to cancel each other out. Hence, as pipelined design register count is related to the sum of individual microcycles, a simple average between the two register limits is taken for Rp_{est}, the estimated number of registers for pipelined designs.

$$Rp_{est} = \frac{1}{2}(Rp_{max} + Rp_{min}) \qquad (4.3.9)$$

For the unique case where a pipelined design has a single microcycle, then registers are used at only the input edges giving

$$Rp_{est}(1) = V_{in} \qquad (4.3.10)$$

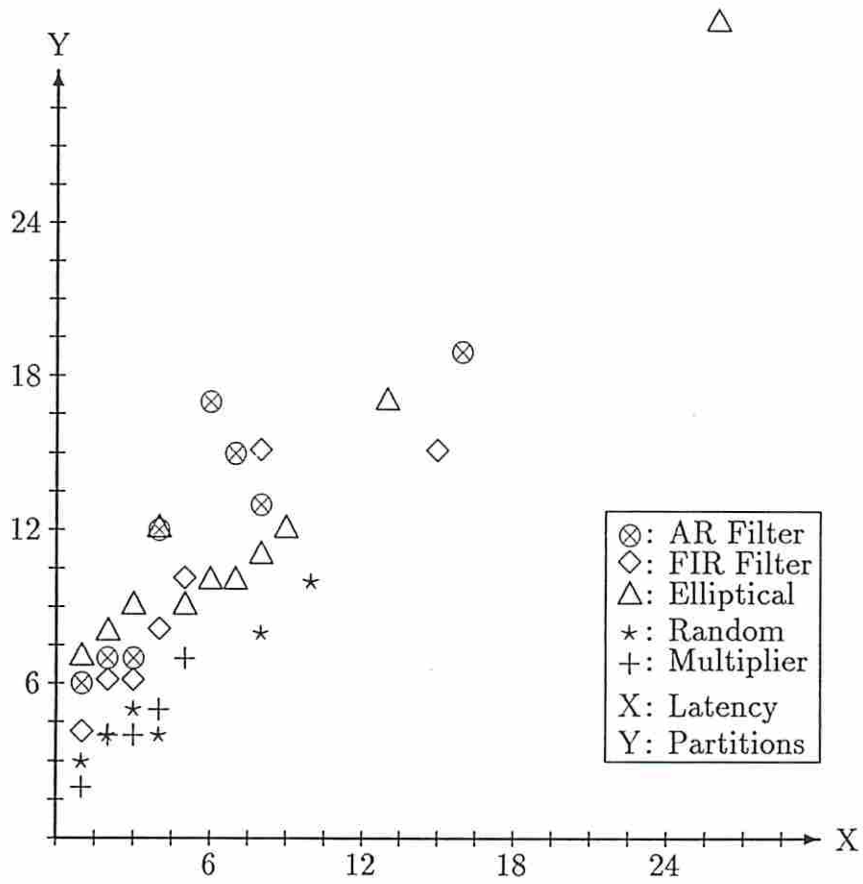


Figure 4.11: Relation between partitions and latency

Init. Interval	Partitions	Registers	
		Actual	Estimate
1	6	74	68
2	7	46	48
3	7	32	38
4	12	36	38
6	17	28	38
8	13	24	28
12	16	21	28
16	19	23	28

Table 4.3: Comparison of pipelined AR Filter estimated and actual register use

The program **REGEST** prints out the number of registers for each specified initiation interval and partition count of pipelined designs. As in the non-pipelined case, the register-bit count is also given. Pipelined results for the AR filter are contained in Table 4.3 and Figure 4.12 with additional results described in Section 4.6.

4.4 Predicting the Number of Microcycles

Predicting the number of microcycles is not important for operator area estimation in pipelined design, but is critical to the register and multiplexer predictors and, to some extent, the controller. The number of microcycles (u) is given by

$$u = \left\lceil \frac{p}{l} \right\rceil \tag{4.4.11}$$

where p is the total number of clock cycles and l is the pipeline initiation interval or latency. (Physically, the number of microcycles is the maximum number of data sets which can be resident in the pipeline at any time.) Prediction of the area-time data path for pipelined designs (**PSADP**) only generates the module counts and the initiation interval; neither the number of clock cycles (p) in a given design nor the number of microcycles are computed. Given initiation interval, efforts here focused on deriving a model for predicting the number of clock cycles which would ultimately allow computation of microcycles.

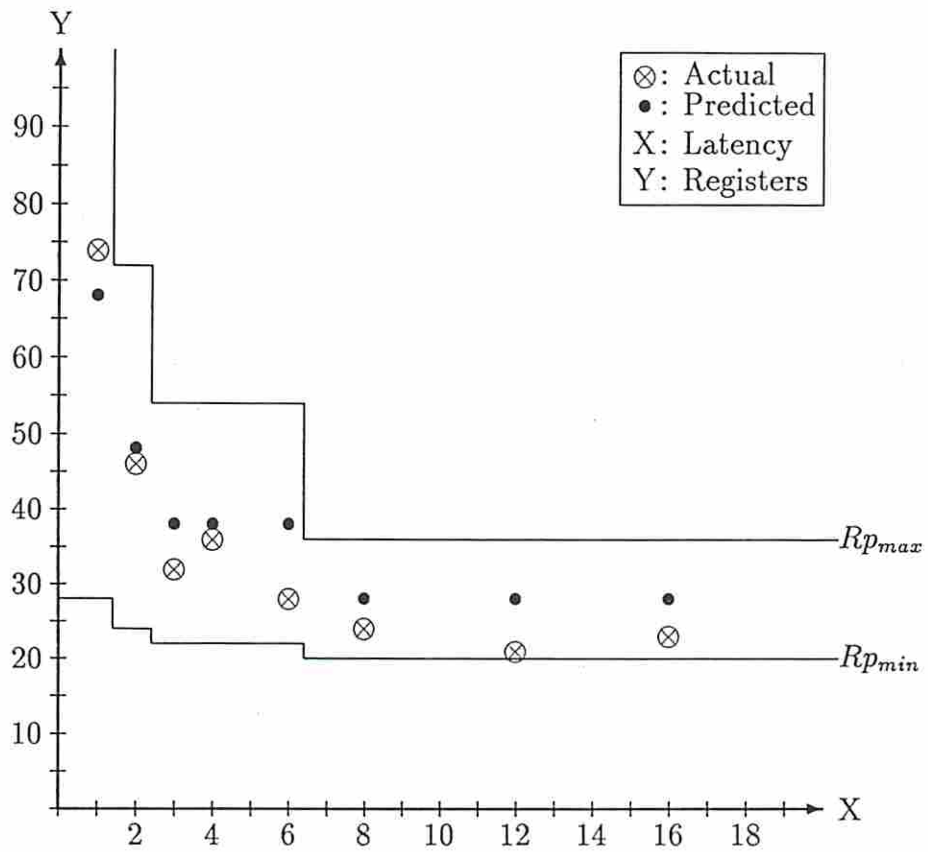


Figure 4.12: Register values for pipelined AR lattice filter

There is a good reason for the lack of a model for pipelined clock cycle count. The number of clock cycles is heavily influenced by the individual data dependencies and ordering of the operations in the dataflow graph besides the direct influence of initiation interval (l). With graph topology having a large effect, clock cycle count is also likely to be affected by the type of pipelining and could even be synthesis tool specific. Since pursuit of such a generalized model is beyond the scope of this thesis, an alternative solution was found in the form of an empirical model.

A brief discussion of empirical observations of initiation interval (l) and clock cycles (p) was given earlier. It was noted that p increased at a somewhat lower rate than l. Additional examination of **Sehwa** (pipelined synthesis) output revealed that p had a constant offset and a somewhat fixed shape versus initiation interval over several dataflow graphs; u steadily decreased as the design was serialized. A simple linear model was formulated to reflect the observed behavior of non-inferior pipelined designs. In particular, the lower-bound pipelined area-delay predictor provided a direct input to this model. The estimate for the number of clock cycles, p, in pipelined design is

$$p = \left\lfloor L' + n_{offcp} \times \frac{l-1}{l_{max}-1} \right\rfloor \tag{4.4.12}$$

L' is the minimum relative length of the *critical path* used to establish the "floor" for the clock cycle count and is given by

$$L' = \left\lceil \frac{T_{cp}}{c_{min}} \right\rceil \tag{4.4.13}$$

where T_{cp} is the delay of the *critical path* and c_{min} is the minimum clock cycle time (the maximum delay of any given module). For the fastest design, where $l = 1$, experiments indicated that clock cycle time was as short as possible, which justifies the "floor" on clock cycle count. (Resynchronization introduces another dimension to the model which entails some knowledge of the graph topology; its effects were not analyzed.)

The fraction in the second term of Equation 4.4.12 scales the maximum off-critical path operation count which could be assigned to separate clock cycles

Table 4.4: Pipelined Predicted Data Values

Design	c_{min}	L'	max$\{n_i\}$	n_{offcp}
Elliptical Filter	375	14	26	20
AR Lattice Filter	2950	8	16	20
FIR Filter	375	4	15	14

and reduce overall resource requirements. Thus, when $l = 1$, the maximum number of resources are used and $p = L'$. According to the AT theory, the graph can be serialized up to

$$l_{max} = max\{n_i\} \tag{4.4.14}$$

where n_i is the effective number of operations of type i.

For the minimum area design, there is only one expected resource of each type i. Experiments revealed that in the general worst case, each off-critical-path operation would require an additional clock cycle. n_{offcp} is the relative number of operations *not* in the critical path or

$$n_{offcp} = \sum_i n_{offcp_i} = \sum_i n_i - N_{cp} \tag{4.4.15}$$

where N_{cp} is the number of operations in the critical path.

Clearly, the minimum number of additional cycles needed is

$$n_{offcp}(min) = max_i\,(n_i - Ncp_i) \tag{4.4.16}$$

Although this value is intuitively more appealing than n_{offcp}, it did not fit the pipelined data as well for the graph sizes used in validation (< 60 operations).

To verify the model, values from several different dataflow graphs were entered into the model and compared against **Sehwa** output. Tables 4.4 and 4.5 contain the results of these comparisons. Note that since the goal is to predict the number of microcycles given by Equation 4.4.11, verification of this model did not compare p. The results indicate a close correlation between the predicted and actual number of microcycles.

Table 4.5: Comparison of Predicted and Actual Microcycles

Design	Init. Intrvl	Actual		Predicted		Error
		Clocks	Microcycles	Clocks	Microcycles	
Efilter	1	14	14	14	14	0
	2	14	7	14	7	0
	3	15	5	15	5	0
	4	17	5	16	4	1
	5	14	3	17	4	1
	6	17	3	18	3	0
	7	19	3	18	3	0
	8	21	3	19	3	0
	9	24	3	20	3	0
	13	19	2	23	2	0
	26	32	2	34	2	0
AR	1	6	6	8	8	2
	2	7	4	9	5	1
	3	7	3	10	4	1
	4	12	3	11	3	0
	6	16	3	14	3	0
	8	12	2	16	2	0
	12	13	2	21	2	0
	16	19	2	26	2	0
FIR	1	4	4	4	4	0
	2	6	3	4	2	1
	3	6	2	5	2	0
	4	8	2	6	2	0
	5	10	2	7	2	0
	8	15	2	10	2	0
	15	15	1	17	2	1

For one example, the AR filter, a deviation between predicted and actual microcycles occurred when initiation interval was small. Due to the dataflow graph topology, whereby the top consists solely of multipliers and the bottom consists of adders, there are two clock cycles in which there is no resource conflict. This advantage was exploited by **Sehwa**. As initiation interval increases, this resource polarization is weakened since a given microcycle is now more likely to contain both module types. Since the partition estimation model does not consider the location of modules, polarized resources will result in a smaller partition count than predicted.

4.5 Multiplexer Area

Another consideration in chip area is the cost (area) of routing data to shared hardware. Two common methods which provide sharing are attaching the operator or register inputs to buses or multiplexer trees. Bus structures are generally more useful when a number of values are transported relatively large distances in the physical circuit whereas multiplexers are more aptly suited to localized communication. Given only the number of operators of each type and the scheduling, a tradeoff between bus and multiplexer allocation is not predictable at this time due to the lack of adequate models for bus estimation. Consequently, for the estimation technique described here, buses are not considered.

In the following sections, the bounds on multiplexer area and a technique for estimating the cost are presented. The equations are simplified in that the bitwidths of individual operators and values are normalized to one. Expansion to include bitwidth attributes is readily accomplished, but needlessly complicates the discussion. Note that **MUXEST**, the program which implements this multiplexer estimation, does take into account operator bitwidth for computing multiplexer costs.

4.5.1 Theoretical Bounds on Multiplexer Area

Bounds on multiplexer area can be determined by finding the separate bounds on multiplexers for operators and registers and summing them. A given dataflow

graph contains $\grave{n}_i$ operations of type i (with n_i or effective number of operations being equal to the worst-case number of operators needed). A specific design implementation has o_i resources of type i ($o_i \leq n_i \leq \grave{n}_i$) and R registers.

The number of multiplexers needed depends upon the design parameters (n_i, $\grave{n}_i$, o_i, R) and the shape of the dataflow graph. Clearly, in the best case where incoming values to an operator or register always arrive from the same source, no multiplexers are needed. In the worst case, all incoming values arrive from unique sources. These bounds were derived earlier by Kurdahi [Kur87].

$$0 \leq M_i \leq \grave{n}_i - o_i \qquad (4.5.17)$$

$$0 \leq M_r \leq V_{reg} - R \qquad (4.5.18)$$

M_i is the number of $2:1$ multiplexers needed for all operators of type i and M_r is the total number of $2:1$ multiplexers needed for registers which lie within the bounds given above.

V_{reg} is the number of values in a dataflow graph which are assigned to registers and is dependent upon the shape of the graph, the design style (pipelined or non-pipelined) and the scheduling. Each design style is separately addressed.

4.5.2 Estimating Multiplexers in Non-Pipelined Designs

Multiplexer area varies with the degree of operator sharing in non-pipelined designs. When the number of stages is low, there is little sharing of resources except along conditional paths and, hence, multiplexing is minimal. Increasing the number of stages increases sharing and raises the multiplexer count up to some limit defined by the dataflow graph and hardware allocation for a particular design. Beyond this point, the number of multiplexers may remain constant or decrease. Despite the increased sharing as more serialized designs are produced, fewer multiplexers may be needed since the number of distinct locations where values are produced is decreasing.

The number of values routed through multiplexers is some fraction of the total values in the dataflow graph, V. V is defined as the number of operations, all n_i, plus the number of unique incoming values from *root*, V_{in}. (Again, if a

given operation type produces more than one value, then the equations must be adjusted to reflect these additional values.)

$$V = \sum_i n_i + V_{in} \tag{4.5.19}$$

V will ultimately be used to determine the number of unique values presented at a given operator or register input. Using the effective number of operations (n_i) more accurately reflects implementation than the total number of operations ($\grave{n}_i$) as will be shown.

For dataflow graphs with conditionals, $\grave{n}_i > n_i$ for one or more i. However, it is not necessary to consider all the conditional values generated, just one path from *root* to *outport*, as values primarily leave a conditional through the nearest *join*. Operations outside of the conditional use values generated in conditionals through the associated *join*; essentially, this gives outside operators access to values in only one branch of a conditional. Operations in mutually exclusive branches can never use value(s) generated operations in another branch of that conditional. Because of these characteristics, the effective number of operations (n_i) best approximates the number of values produced within the graph which are available for input by other operators or registers.

The number of values which are assigned to registers, V_{reg}, is schedule dependent. Given that R registers are used for a specific design, the dataflow graph has at least $V_{reg} \geq R$ values to be stored. Given that a design has p clock cycles, a *minimum* of one stored value is generated per clock cycle giving a different lower bound: $V_{reg} \geq p$. The maximum of these two quantities provide a greatest lower bound on V_{reg}.

One upper bound on V_{reg} is simply all values that could be assigned a register, which is all edges in the graph which do not have *root* as their source. However, the maximum value of V_{reg} may not exceed the assignment of R registers for each of the p stages. Summarizing,

$$\max(R, p) \leq V_{reg} \leq \min(V - \mid E_{in} \mid, p \times R) \tag{4.5.20}$$

Since V_{reg} cannot be explicitly determined without knowing the exact scheduling in a dataflow graph, V_{reg} is arbitrarily set to an average of these bounds or

$$V_{reg} = \frac{\max(R, p) + \min(V - \mid E_{in} \mid, p \times R)}{2} \qquad (4.5.21)$$

Note that a special case occurs for $p = 1$ giving $V_{reg} = R$ as registers are only assigned at the output and no multiplexers are needed for registers.

The multiplexer estimation models can be divided into two categories: multiplexers attached to operator inputs and multiplexers attached to registers inputs. This separation is due to the input connectivity of these elements. Operator inputs are either connected to operator or register outputs or *root*; register inputs are only connected to operator outputs (for non-pipelined designs). The models derived are based on the probable number of unique values expected at each operator or register input.

4.5.2.1 Multiplexer Area with Operators in Non-Pipelined Design

A model for predicting multiplexer usage with operators in non-pipelined designs is detailed in this section. First, the number of values from other operators and registers which could be found on the inputs of all operators of a given type is defined. Next, the number of *distinct* values expected at an input is derived based upon the number of values and number of selections drawn from this pool. Using these results, the multiplexer count is readily computed.

In a given design, the total number of distinct values arriving at the input of an operator of type i is dependent upon the number of occurrences of this operation type in the graph ($\grave{n}_i$) and the number of operators actually implementing the function (o_i). Other factors include the number of distinct physical inputs (I_i) and outputs (O_i) associated with this operation type. For example, an adder would have two inputs (three if carry-in), and one output (two for carry-out); I_i and O_i do not inherently reflect the bitwidth of a particular connection and are assumed to be one. (However, the **MUXEST** estimation program does use the actual bitwidth.)

Values found on inputs of a specific operator may be from any operator or register, excepting those values which are solely directed to *outport* (V_{out}) and

any values output by the node itself (O_i). The values eligible for input to an operator of type i are thus

$$V^i = V - V_{out} - O_i \tag{4.5.22}$$

Some of these V^i eligible values are assigned to registers (V^i_{reg}), while the remainder are only associated with operators ($V^i - V^i_{reg}$). The number of values which are assigned registers, V^i_{reg}, is modified to reflect the exclusion of values directed solely towards *outport*.

$$V^i_{reg} = \max(V_{reg} - V_{out} - O_i, 0) \tag{4.5.23}$$

The probabilistic model used to estimate the number of multiplexers needed at each of the I_i inputs makes the following assumptions:

- Every incoming value either arrives from a register or an operator.
- The V^i eligible values have a *uniform probability* of being found on a given input of each operator of type i.
- Each selection occurrence is independent of any other. Thus, the selection of a specific eligible value does not affect the probability of it being selected again.
- The multiplexer count needed at each input of operator i is directly related to the number of statistically *distinct* values connected to it; the maximum number, when all connected values are distinct, is $\grave{n}_i - o_i$.

An output value V_a is *distinct* from another output V_b if they are physically output from different registers or operators. As sharing of operations increases, the number of distinct values in the graph decreases as there are fewer physical places where values are generated.

An operator of type i has multiplexers at each of the I_i inputs determined by the expected number of *distinct* values originating in operators, $V_o(i)$, and the expected number of *distinct* values originating in registers, $V_r(i)$. $V_r(i)$ is weighted by the fraction of values assigned to registers; $V_o(i)$ is weighted by the

fraction of values *not* assigned to registers. The number of multiplexers, M_i, required at the inputs of all operators of type i is

$$M_i = \left\lceil \left(1 - \frac{V^i_{reg}}{V^i}\right) V_o(i) + \frac{V^i_{reg}}{V^i} V_r(i) \right\rceil - 1 \quad (4.5.24)$$

$V_o(i)$ and $V_r(i)$ are found by computing the expected number of distinct members chosen from a pool of values. For $V_o(i)$, the pool consists of all values (V^i); for $V_r(i)$, the number of values is equal to the number of registers (R). V^i is not reduced by V'_{reg} - the number of values assigned to registers - since a given value may be found both on a register input and an operator input. The number of selections drawn from each pool is dependent upon the number of operations ($\grave{n}_i$) and the number of operators used to implement this operation (o_i).

$$V_o(i) = \mathcal{V}(\grave{n}_i - o_i + 1, V^i) \quad (4.5.25)$$

$$V_r(i) = \mathcal{V}(\grave{n}_i - o_i + 1, R) \quad (4.5.26)$$

The total number of operations ($\grave{n}_i$) is taken instead of the effective number of operations (n_i). Operations of the same type along different mutually exclusive paths may share the same hardware, but may not have the same connectivity; multiplexers might be necessary.

$\mathcal{V}(k, n)$ is the *average* number of *distinct* members chosen after k selections from a pool of n distinct members, with replacement. Drawing upon a pool of n members k times gives a total of n^k possible ordered k-tuples. Q is the quantity of these k-tuples which have exactly q distinct members. The weighted average of Q taken over all possible values of q gives the *expected (or mean) number of distinct members*, $\mathcal{V}$. Clearly, q is bounded by the lesser of k selections or n members.

$$\mathcal{V}(k, n) = \frac{1}{n^k} \sum_{q=1}^{\min(n,k)} q Q(n, k, q) \quad (4.5.27)$$

As an example, when $n = 5$ and $k = 3$, q could range from 1 for tuples (a, a, a), (b, b, b), $\ldots$, (e, e, e) to 3 for tuples (a, b, c), (a, b, d), ..., (c,d,e). The number of 3-tuples where q = 1, 2, or 3 is $Q(5, 3, 1) = 5$, $Q(5, 3, 2) = 60$, and

$Q(5,3,3) = 60$ giving

$$\mathcal{V} = \frac{1 \times 5 + 2 \times 60 + 3 \times 60}{5^3} \tag{4.5.28}$$

or 2.44 as the expected number of distinct values.

Finally, $Q(n,k,q)$ is mathematically described by ordered Stirling numbers of the second kind [Liu68] giving

$$Q(n,k,q) = \frac{1}{q!} P \begin{pmatrix} n \\ q \end{pmatrix} \sum_{j=0}^{q} (-1)^j C \begin{pmatrix} q \\ j \end{pmatrix} (q-j)^k \tag{4.5.29}$$

where $P()$ is the permutation operator and $C()$ is the combination operator.

4.5.2.2 Multiplexer Area with Registers in Non-Pipelined Design

The number of multiplexers with registers is dependent upon the number of values assigned to registers (V'_{reg}) and the number of registers available (R). For non-pipelined design, we assume that the only values found on a register input are produced by operators. Many designers and synthesis programs make the same assumption. Exceptions usually occur in CPU design, where synthesis is less applicable. Thus, the pool of values is the sum of all o_i. These values have a uniform probability of being found on a register input. Thus, the number of multiplexers required for all registers in non-pipelined design, M_r, is

$$M_r = \left\lceil \mathcal{V}(V'_{reg} - R + 1, \sum_i o_i) \right\rceil - 1 \tag{4.5.30}$$

4.5.2.3 Example for Non-Pipelined Design

To illustrate the model, a small example was chosen as shown in Figure 4.13. A single design produced by **MAHA**, a non-pipelined synthesis program, provided the scheduling and resource allocation for the adders and multipliers with **REAL** provided the register assignment. Multiplexers were inserted manually and verified using **MABAL**, a module and bus allocator (which can determine register and multiplexer requirements).

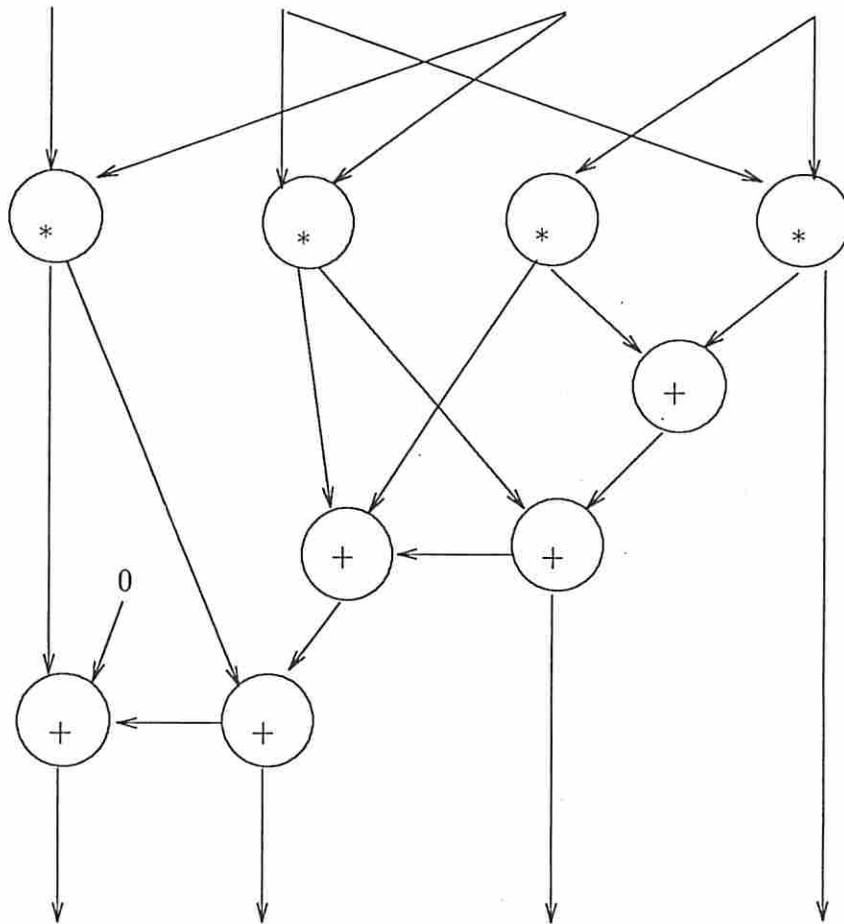


Figure 4.13: Example dataflow graph showing scheduling

For this design, three adders and two multipliers are used in four stages with five registers. (Edge cutsets indicating the schedule produced by **MAHA** are marked in Figure 4.13.) All multiplier inputs are attached to one of four distinct graph inputs. By swapping inputs on the multiplier, the total multiplier multiplexer count can be reduced from three to two. Values found on all adder inputs consist of multiplier outputs in two cases, register outputs in six cases, and adder output and constant value in one case each. All carry-in pins are either attached to a constant (zero) or to one of two registers. Rearranging inputs reduces the number of adder multiplexers to four on a single adder: one for one input, one for the other input, and two for the carry-in bit. (The other adders do not need multiplexers.) Finally, two registers require three multiplexers amongst them giving a total of 9 multiplexers for the manual design. The final architecture, validated using **MABAL**, is shown in Figure 4.14.

For the probabilistic model, a more general examination of the graph is made. There are five inputs to the graph (one is a constant), five register outputs, two adder outputs, and two multiplier outputs which might be found at each adder input. (The carry line is folded into one of the adder inputs.) The resultant multiplexer estimator predicts that four multiplexers are needed for all adders and four for all multipliers. (The distribution of these eight total multiplexers among the two adders and three multipliers is unknown.) The predicted multiplexer requirements for registers is one. The net result is that the probabilistic model matches the highly minimized design. If the maximum number of multiplexers were taken as an estimate, the result would have been 12 multiplexers. A broader assessment of this model compared to actual synthesis results is given later.

4.5.3 Estimating Multiplexers in Pipelined Designs

As in the pipelined register estimation, multiplexer estimation for pipelined designs shifts to computation of an average for a single microcycle which is repeated over all microcycles. The model described here does not consider sharing of multiplexers between different microcycles.

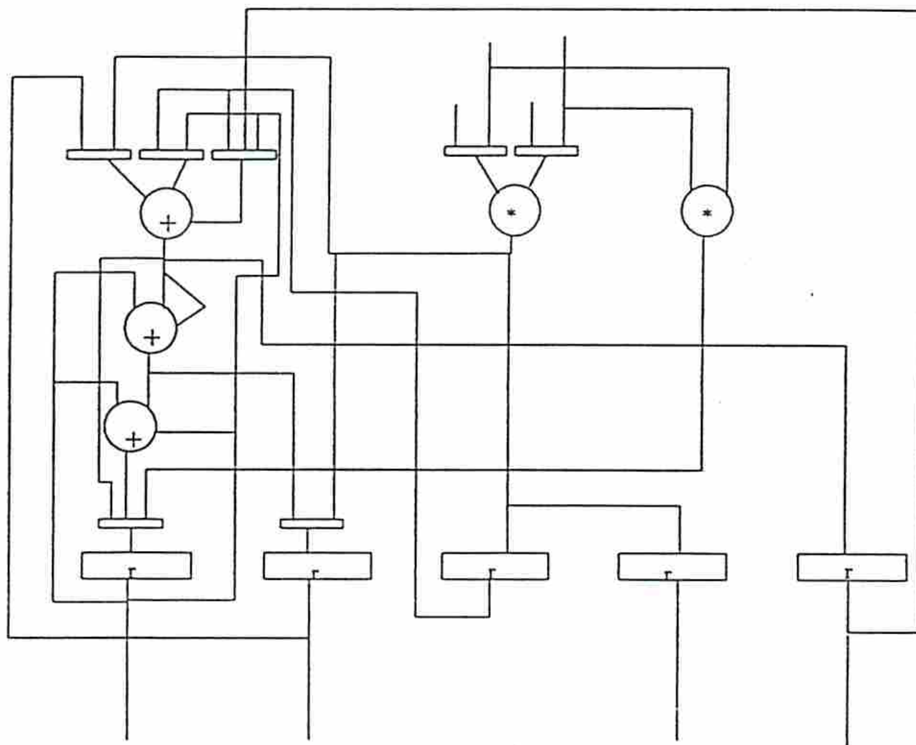


Figure 4.14: Example dataflow graph showing RTL architecture

Similar to non-pipelined design, but confined to a single microcycle, estimates are derived for values (V^u), registers (R^u), and values assigned to registers (V^u_{reg}). The average number of registers used in a given microcycle, R^u, is

$$R^u = \left\lceil \frac{Rp - V_{out}}{u} \right\rceil \tag{4.5.31}$$

Graph output edges (V_{out}) are excluded from R^u since, in the model used by the synthesis tools, they are not assigned to any register. Values found on an operator input within a given microcycle are produced by registers or operators within that microcycle. The one exception occurs in the first microcycle where values from *root* may be found on an operator or register input. The average number of values per microcycle, V^u, can be computed as

$$V^u = \left\lceil \frac{R^u(u-1) + V_{in} + \sum_i n_i}{u} \right\rceil \tag{4.5.32}$$

with the fraction of these values assigned to registers bounded by the number of stages within a given microcycle (the initiation interval). The estimated number of values assigned to registers per microcycle for pipelined design, V^u_{reg}, is a slight modification of Equation 4.5.20. V^u and R^u substitute for $V - V_{in}$ and R; initiation interval l replaces p.

$$\max(R^u, l) \leq V^u_{reg} \leq \min(V^u, lR^u) \tag{4.5.33}$$

Similar to non-pipelined estimation, the limits are averaged to compute V^u_{reg}.

$$V^u_{reg} = \frac{\max(R^u, l) + \min(V^u, lR^u)}{2} \tag{4.5.34}$$

4.5.3.1 Multiplexer Area with Operators in Pipelined Design

Equation 4.5.24 is modified to compute the number of multiplexers used in a given microcycle. For a specific microcycle z, the number of multiplexers needed for all operators of type i, $M(1)^z_i$, is

$$M(1)^z_i = \left\lceil \frac{V^u - V^u_{reg}}{V^u} V^z_o(i) + \frac{V^u_{reg}}{V^u} V^z_r(i) \right\rceil - 1 \tag{4.5.35}$$

```
procedure compute − M_i^u − sum
begin
    Set y_i = ⌈(ṅ_i − o_i)/u⌉.
    Set T = ⌈(ṅ_i − o_i)/y_i⌉.
    Set M_i^u = T * M_i^1(1).
    Set new y_i = ṅ_i − o_i − T * y_i.
    If y_i > 0
    begin
        Set M_i^u = M_i^u + M(1)_i^z.
    end
end
```

Figure 4.15: Computation of y_i and M_i^u

Again $V_o^z(i)$ and V_r^z are found by computing the number of unique objects chosen from the pool of values *but* restricted to a single microcycle.

$$V_o^z(i) = \mathcal{V}(y_i + 1, V^u - 1) \tag{4.5.36}$$

$$V_r^z(i) = \mathcal{V}(y_i + 1, R^u) \tag{4.5.37}$$

y_i is a variable which describes the excess selection set size of operation type i during a given microcycle. Initially, it is set to

$$y_i\ (initial) = \left\lceil \frac{\grave{n}_i - o_i}{u} \right\rceil \tag{4.5.38}$$

This value for y_i is used for up to T microcycles, which is the microcycle count where the sum of all y_i matches the total excess $(\grave{n}_i - o_i)$. Clearly, $T \leq u$.

$$T = \left\lceil \frac{\grave{n}_i - o_i}{y_i} \right\rceil \tag{4.5.39}$$

The sum of the multiplexers used in each of the microcycles $M(1)_i^z$, where $y_i > 0$, gives the number of multiplexers for all operators of type i, M_i^u. An algorithm for computing y_i and M_i^u is detailed in Figure 4.15.

4.5.3.2 Multiplexer Area with Registers in Pipelined Design

Multiplexer area in pipelined designs is similar to the non-pipelined computation of Equation 4.5.30. In each microcycle, the number of values which can be assigned to registers is

$$V_r^u = \left\lceil \frac{R^u(u-1) + \sum_i o_i}{u} \right\rceil \tag{4.5.40}$$

This quantity of values is fixed for all microcycles. When applied over all microcycles, the total number of multiplexers with registers, M_r^u, is determined.

$$M_r^u = \left\lceil u\mathcal{V}(V_{reg}^u - R^u + 1, V_r^u) \right\rceil - 1 \tag{4.5.41}$$

4.6 Experiments and Validation

To validate the models, a number of tools were used to generate designs with registers and multiplexers for comparison. For non-pipelined designs, a method has been published whereby register allocation is minimal (**REAL**) [KP87]. By coupling the data path synthesis results from **MAHA** with **REAL**, accurate register values can be obtained. Predicted register results were produced by **REGEST**, a program which implements the register analysis algorithm presented in this chapter. (The use of **REGEST** is described in Appendix F.) Tables 4.6 and 4.7 summarize the register experiments on non-pipelined designs. For these examples, a variety of algorithms with dataflow graph representations under 60 nodes in size were used.

Each non-inferior design as produced by **MAHA** and **REAL** is listed in Tables 4.6 and 4.7 along with the predicted register count from **REGEST**. The error is also given.

The last example in the Table 4.7, *Conditional*, is a conditional dataflow graph described in Park [PP86]. It should be noted that **REAL** does not guarantee finding the optimal number of registers for conditionals; the optimal number is actually 3 instead of 5. Hence, the estimates are low in comparison to **REAL**, but correct.

Dataflow Graph	Partitions	Registers		Error
		Actual	Predicted	
AR filter	1	2	2	0
	3	4	5	1
	4	5	5	0
	7	6	5	1
	8	6	5	1
	9	6	5	1
	14	5	5	0
	17	7	5	2
	18	7	5	2
	19	6	5	1
FIR filter	1	1	1	0
	2	8	4	4
	3	7	4	3
	4	7	4	3
	5	5	4	1
	8	2	4	2
	11	5	4	1
	15	5	4	1
Elliptical filter	1	8	8	0
	3	8	8	0
	4	8	8	0
	5	8	8	0
	7	9	8	1
	8	9	8	1
	9	9	8	1
	13	11	8	3
	14	9	8	1
	15	9	8	1
	28	11	8	3

Table 4.6: Register prediction: Non-pipelined designs

Dataflow Graph	Partitions	Registers		Error
		Actual	Predicted	
HAL example	1	3	3	0
	2	4	4	0
	3	3	4	1
	7	4	4	0
Random	1	2	2	0
	2	6	7	1
	3	8	7	1
	4	7	7	0
	5	7	7	0
	7	9	7	2
	10	8	7	1
	14	8	7	1
Multiplier	1	4	4	0
	3	4	4	0
	4	5	4	1
	5	4	4	0
	6	4	4	0
Conditional*	1	2	2	0
	2	3	3	0
	5	5	3	2
	7	5	3	2

* **REAL** did not find correct values for actual designs; see text.

Table 4.7: Register prediction: Non-pipelined designs

The results of register experiments on pipelined designs are depicted in Tables 4.8 and 4.9. Examples were processed through **Sehwa** and results from **REAL** collected; it is not known whether optimal results are generated for pipelined designs. The register estimate errors are clearly higher than that for non-pipelined designs. Given that register prediction in a pipelined design is treated as two or more concatenated non-pipelined designs, the prediction error in a given stage is essentially multiplied by the number of microcycles. (An error measure which is the total register error divided by the number of microcycles gives errors comparable to that for non-pipelined estimates.) Unfortunately, a more precise estimate is likely to require some knowledge of the number of values crossing each microcycle boundary; specific scheduling information is not available during prediction.

Problems were also experienced using **REAL** with a conditional graph when the number of microcycles was greater than one. Therefore, manual register assignment was accomplished as shown by a fast conditional pipelined design in Figure 4.16. In this figure, the number of registers assigned by a human designer is shown. (This is divided into the number of registers required on that stage "+" the number of registers required for incoming values from *root*.) Since no registers can be shared, the human result of 25 registers (27 if output latches used) is optimal and compares favorably to the estimate.

For both pipelined and non-pipelined designs, the register prediction model indicates a good match. Where errors occurred in non-pipelined designs, there appeared to be no pattern as to their magnitude. Conversely, for pipelined designs, the location of operations within the dataflow graph could be correlated to error. In particular, graphs which had acute "necking", where edge cutsets in one region of the graph have large cardinality whereas those in another have small cardinality, often produced estimates higher than measured. (Both the AR and Elliptical filters exhibit necking.) The register prediction error in pipelined designs magnifies any register variation between microcycles or register sharing across microcycle boundaries.

To validate the multiplexer results produced by **MUXEST**, a tool recently added to the **ADAM** system was used. Known as **MABAL** for module and bus allocator, it accepts a schedule and a dataflow graph and performs the tasks of

Dataflow Graph	Init. Interval	Partitions	Registers		Error
			Actual	Predicted	
AR filter	1	6	74	68	6
	2	7	46	48	2
	3	7	32	38	6
	4	12	36	38	2
	6	17	28	38	10
	7	15	24	38	14
	8	13	21	28	7
	16	19	23	28	5
FIR filter	1	4	44	42	2
	2	6	32	33	1
	3	6	25	25	0
	4	8	24	25	1
	5	10	20	25	5
	8	15	20	25	5
	15	15	17	17	0
Elliptical filter	1	7	73	53	20
	2	8	43	32	11
	3	9	35	25	10
	4	12	33	25	8
	5	9	18	18	0
	6	10	18	18	0
	7	10	17	18	1
	8	11	17	18	1
	9	12	18	18	0
	13	17	16	18	2
	26	32	16	18	2

Table 4.8: Register prediction: Pipelined designs

Dataflow Graph	Init. Interval	Partitions	Registers		Error
			Actual	Predicted	
Random	1	3	30	26	4
	2	4	21	19	2
	3	5	15	19	4
	4	4	14	14	0
	5	5	14	14	0
	6	6	14	14	0
	8	8	14	14	0
	10	10	14	14	0
Multiplier	1	2	8	8	0
	2	4	9	8	1
	3	4	8	8	0
	4	5	8	8	0
	5	7	8	8	0
Conditional*	1	5	55	25	30
	2	6	30	19	11
	3	6	19	16	3
	5	5	13	13	0
	6	6	13	13	0

* **REAL** did not find correct values for actual designs; see text.

Table 4.9: Register prediction: Pipelined designs

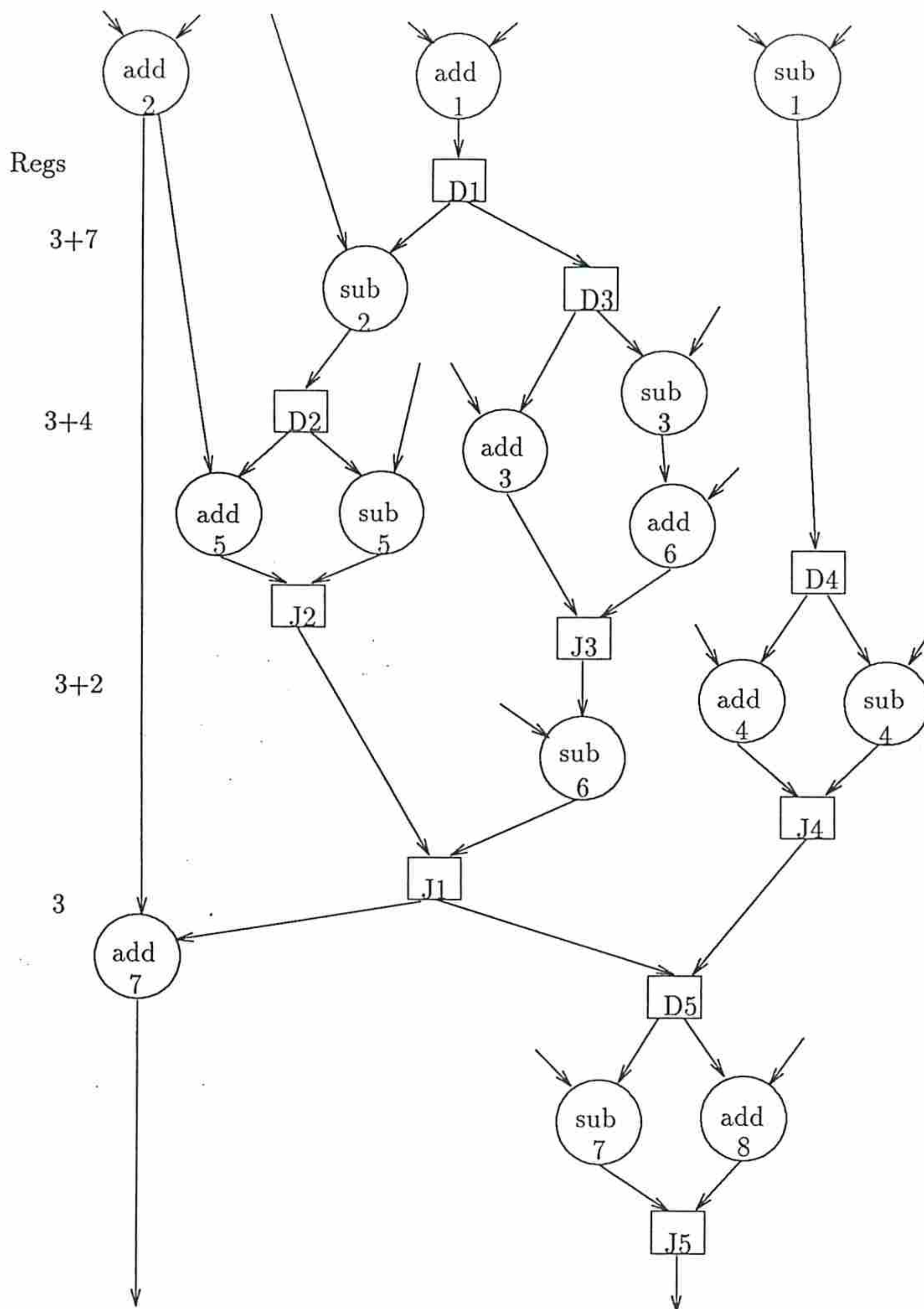


Figure 4.16: Pipelined conditional graph

operator allocation, register allocation, bus and/or multiplexer allocation, and module binding to produce a complete netlist at the RT-level [KP90]. Register assignment is based upon the algorithms used in **REAL**; bus and multiplexer assignment is driven by cost-based heuristics. **MABAL** does a high degree of minimization; thus, for very regular graphs, one would expect it to achieve better than the expected results.

A comparison between the multiplexer estimation and **MABAL** results are presented in Tables 4.10 through 4.12 for non-pipelined and pipelined designs, respectively. (The use of **MUXEST** is described in Appendix G.) Designs with a single stage are excluded, with the exception of the conditional, as no multiplexers are needed. As can be seen, the estimaters compare favorably against actual results with a couple of exceptions. The average error of several multiplexers appears to be independent of the graph size.

4.7 Summary

In this chapter, models for estimating register and multiplexer requirements have been presented. Both non-pipelined and pipelined designs, as well as behavior which includes conditional paths, are encompassed. The simplicity of these models eases their insertion into current synthesis packages and provides useful results.

One of the disadvantages is the computation complexity of theoretical register bounds. Although this analysis need only be performed once per dataflow graph, the computation time is on the order of $O(n^2)$ where n is the number of nodes in the dataflow graph. Extensions which include bus structures would also be useful in trading off multiplexer and bus usage for a given design prior to synthesis. However, the current model is adequate for integration with other data path operator and control path area prediction models.

Dataflow Graph	Partitions	Multiplexers				Total Error
		Actual		Predicted		
		Op	Reg	Op	Reg	
AR filter	3	24	3	30	3	6
	4	30	5	30	5	0
	7	34	3	34	5	2
	8	33	7	34	4	2
	9	34	7	36	3	2
	14	33	5	34	2	2
	17	34	5	34	3	2
	18	34	5	34	2	3
	19	32	4	34	1	1
FIR filter	2	16	1	14	3	0
	3	25	3	24	5	1
	4	27	4	26	5	0
	5	29	6	28	4	3
	8	24	2	28	3	5
	10	28	2	30	2	2
	15	27	1	30	1	3
Elliptical filter	3	26	8	30	6	2
	4	27	7	32	7	5
	5	31	8	34	7	2
	7	30	7	36	6	5
	8	29	10	36	6	3
	9	28	9	36	5	4
	13	31	10	38	5	2
	14	30	11	38	4	1
	15	35	7	40	2	0
	28	26	3	38	1	10

Table 4.10: Multiplexer estimation: Non-pipelined designs

Dataflow Graph	Partitions	Multiplexers				Total Error
		Actual		Predicted		
		Op	Reg	Op	Reg	
HAL example	2	6	2	8	1	1
	3	10	3	12	2	1
	7	10	2	12	2	2
Random	2	23	1	24	3	3
	3	29	3	30	5	3
	4	33	4	32	6	1
	5	35	5	34	6	0
	7	37	5	36	5	1
	10	35	4	36	4	1
	14	35	4	36	3	0
Multiplier	3	4	3	4	2	1
	4	8	3	8	1	2
	5	10	3	12	2	1
	6	10	2	12	1	1
Conditional	1	8	0	8	0	0
	2	18	3	16	3	2
	5	19	4	18	4	1
	7	22	6	18	5	5

Table 4.11: Multiplexer estimation: Non-pipelined designs

Dataflow Graph	Init. Interval	Partitions	Multiplexers				Total Error
			Actual		Predicted		
			Op	Reg	Op	Reg	
AR filter	2	7	28	12	26	11	3
	3	7	32	14	40	8	2
	4	12	34	13	40	8	1
	6	16	32	12	38	10	4
	7	12	33	11	40	8	4
	8	13	32	14	34	8	4
	16	19	33	10	38	9	4
FIR filter	2	6	18	8	20	6	0
	3	6	27	11	28	5	5
	4	8	28	10	32	5	1
	5	10	27	10	32	7	2
	8	15	28	10	36	6	4
Elliptical filter	2	8	22	10	30	11	9
	3	9	26	10	38	10	12
	4	12	29	12	42	10	9
	5	9	27	18	40	10	5
	6	10	29	19	40	9	1
	8	11	30	14	38	9	3
	9	12	29	13	40	9	7
	13	17	28	11	34	9	4
	26	32	21	9	28	8	6
Random	2	4	24	8	26	8	2
	3	5	25	13	32	9	3
	4	4	30	9	34	5	0
	5	5	32	11	36	5	2
	6	6	34	14	36	5	7
	8	8	32	13	38	4	3
	10	10	30	11	40	4	3
Conditional	1	5	8	0	4	0	4
	2	6	19	4	14	4	5
	3	6	22	4	18	6	2
	5	5	20	4	16	6	2
	6	6	20	6	18	6	2

Table 4.12: Multiplexer estimation: Pipelined designs

Chapter 5

Analysis of Control Path/Data Path Tradeoffs

5.1 Introduction

In order to migrate functionality between the control path and data path, explicit tradeoffs can be applied to the behavior. In this chapter, the focus is upon control path/data path tradeoff templates. The types of tradeoffs and example usage will be presented in the first part.

In the second part, one the tradeoff types - bitwidth tradeoffs - is further examined. Using the control path area model of Chapter 3, an analytical model for assessing bitwidth tradeoffs is developed. Finally, the model is applied to several examples and limitations of this model described.

5.2 Types of Control Path/Data Path Tradeoffs

Control path/data path tradeoffs encompass a large range of design activities. Interactions within the data path and control path as well as between data path and control path hardware collectively comprise these tradeoffs. Tradeoffs having primary impact on the data path which, in turn, affect the control path are presented in this section. (Controller specific tradeoffs such as use of the PLAs versus microcode or multilevel logic are not addressed here.)

Tradeoffs which affect the area of the data path and the controller are useful for exploring the migration of functionality as certain decisions are made. In the simplest case, a designer may desire a very small design and would thus have the tendency to reduce hardware to a minimal quantity. If the impact of

data path serialization upon the control path and routing/storage hardware is ignored, an inferior design may be realized. Even if considered, sharing of data path hardware may not be adequate to meet a tight area design goal, and other types of tradeoffs must be examined. The use of bit-serial operators, for example, will expand the design space into a region which may satisfy the objectives.

There are two types of control path/data path tradeoffs: *explicit* and *implicit*. Explicit tradeoffs are those which specifically affect *both* the control path and data path area and/or time. These tradeoffs involve *implementation of control path functions in data path hardware*. For example, the implementation of a loop counter within a PLA versus an external counter comprises an explicit tradeoff.

There are also *implicit* tradeoffs where changes to the data (control) path indirectly affect the control (data) path. Three types of implicit tradeoffs which have been identified are *composition*, *decomposition*, and *sequential/combinational* tradeoffs. These are explored in this chapter.

5.2.1 Composition of Operators

Operator composition or complex operator substitution is one class of hardware/firmware analysis problem. Choosing an adder/subtractor or ALU over a simple adder is not obvious. Clearly, there is an immediate negative impact on both the data path and control path area. However, a parallel design may perform faster or, more likely, a serialized design may achieve a previously unreachable area objective.

Use of an ALU versus dedicated hardware (such as a multiplier) is only partly a module selection issue. This substitution is not a 1:1 mapping since the ALU could also perform other functions. Additional control is also required for operation and sharing of this resource which affects the overall circuit size.

Another form of composition is transformation from a subgraph of interconnected operations into a single operation. One such example is shown in the AR lattice filter of Figure 5.1. There are eight instances of the multiply/adder pair (highlighted in the graph). A costly (but fast) module implementation, as compared to separate multipliers and adder, might result in a finished design where *both* area and time are lower than implementations of the original graph. This is

true when the data path area reduction dominates the overall area change and the increased circuit speed is not hampered by a slower controller.

5.2.2 Operator Decomposition

Decomposition entails expansion of some operation into a collection of more primitive operations which achieve an identical outcome. Operator decomposition is useful

- to reduce the area of hardware implementation (possibly at the expense of controller size),
- when no library module is capable of providing the function, and
- to explicitly define operations at a lower level (such as a ripple-carry versus carry-lookahead adder).

A multiplier serves as an example of operator decomposition. A candidate implementation for an 8-bit multiply is a 16-bit adder with a shift register. The controller area increases to accommodate looping 8 times to perform the multiply. Although such an approach reduces data path hardware, data path time rises and controller complexity may increase significantly.

Another transformation involves strength reduction. For example, a multiply or divide operation by a power-of-2 term would be transformed into the appropriate set of shift registers. Since transformations of this type introduce operations that may exist elsewhere in the dataflow graph, further area savings may be realized due to resource sharing. Other reductions include transforming addition/subtraction by a small constant into a presettable counter, and multiplication by a small constant implemented via a series of additions.

5.2.3 Sequential/Combinational Tradeoffs

Sequential/combinational or bitwidth tradeoffs are a class of problems where an operation implemented as a single combinational circuit is transformed into a smaller bitwidth set of operators performing in a sequential fashion. Consider a single operation which implements an 8-bit multiply. Expansion into sequential

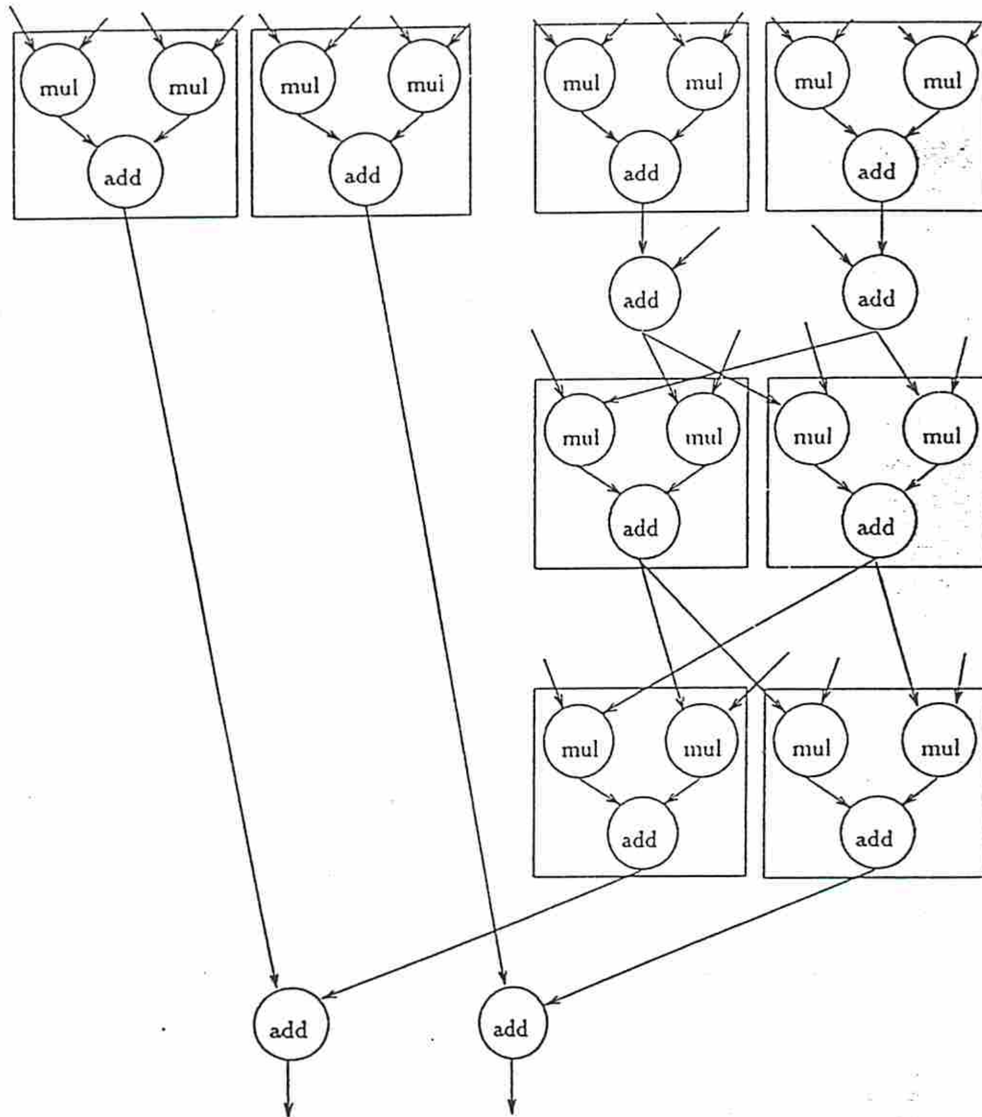


Figure 5.1: AR Filter showing Operation Groupings

4-bit multiplies yields the dataflow graph of Figure 5.2. This simple change has considerable impact on both circuit area and time.

First of all, the multiply is an indivisible structure as far as any synthesis engine is concerned. Consequently, it tends to dominate the dataflow graph since it is either slow or expensive as compared to adders, comparators, and other more prevalent operators. By transforming the multiply, hardware area can decrease at the expense of time - *in general*. If the 8-bit multiply uses CSAs (Carry Save Adders) and the 4-bit multiply is a flash (very high speed) ROM, the second implementation might be faster and/or larger. This "recursive" expansion could continue resulting in additional design possibilities.

Only a global analysis can determine whether such a substitution has pushed the design in the right direction. If the operation occurs frequently in the dataflow graph, the data path area savings may be overwhelmed by the added control loops. Alternatively, a subroutine in the controller could be used. Although extra procedural hardware and stack registers would be needed, there might be other operators which could take advantage of a controller subroutine capability, thereby lowering the overall area.

In the second example dataflow graph of Figure 5.3, a multiplication is located both on and off the critical path. Depending upon the constraints, either none, *mul1*, or *mul1* and *mul2* are candidates for transformation. (*Mul2* is not a candidate by itself since it is on the critical path; if a transformation to a slower design is acceptable *on* the critical path, it is clearly acceptable *off* the critical path for *this* graph. The transformation must consider such effects.) Furthermore, each data path transformation candidate is accompanied by the choice of a sequential, looping, or subroutine control path, giving 13 total implementations. If *sub7* was also a multiplier, this amount would increase to 44 possibilities. An exponential increase in complexity demonstrates why global brute force evaluation is impractical for transformation evaluation.

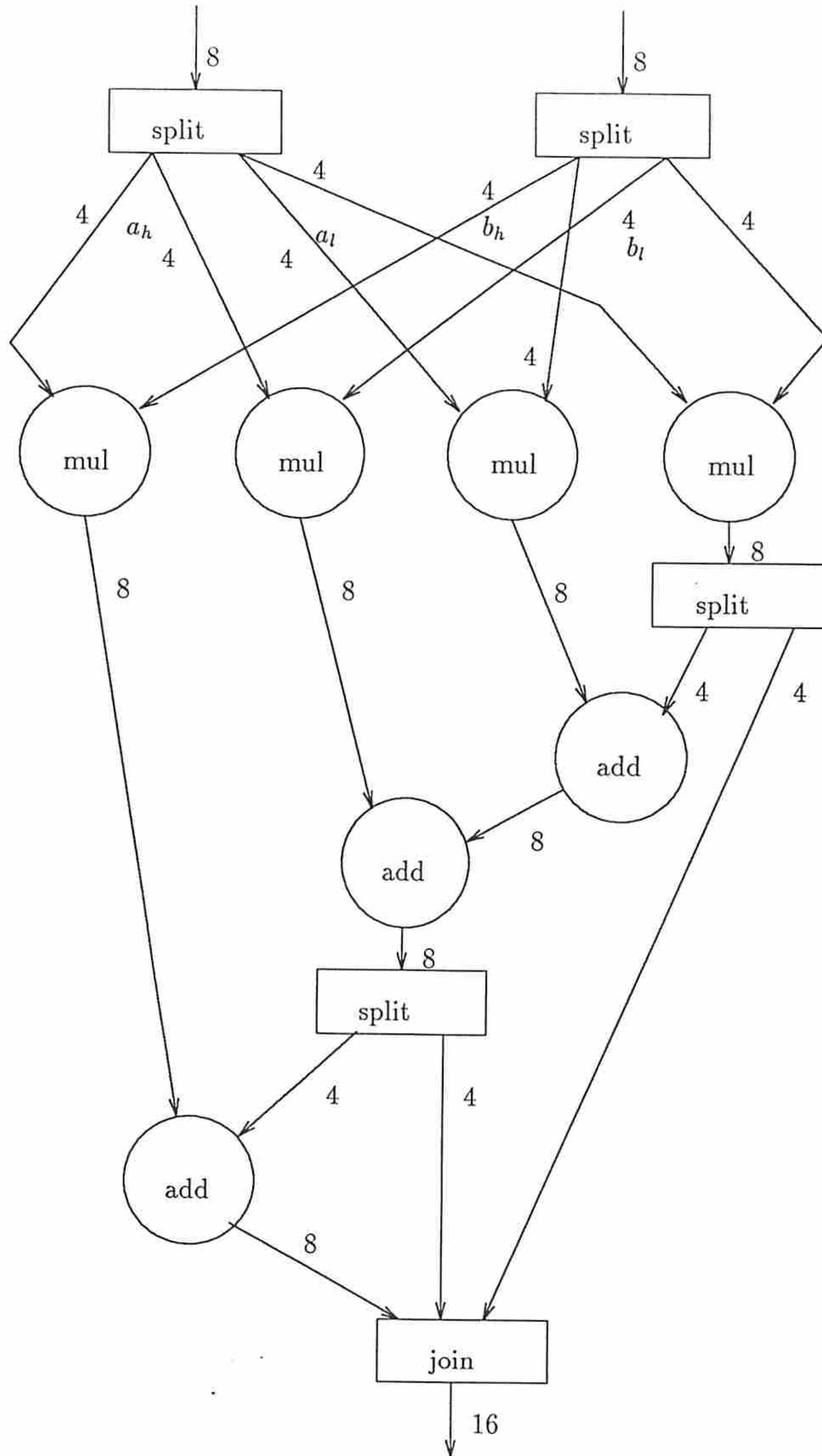


Figure 5.2: 8-bit multiplier implemented using 4-bit multipliers

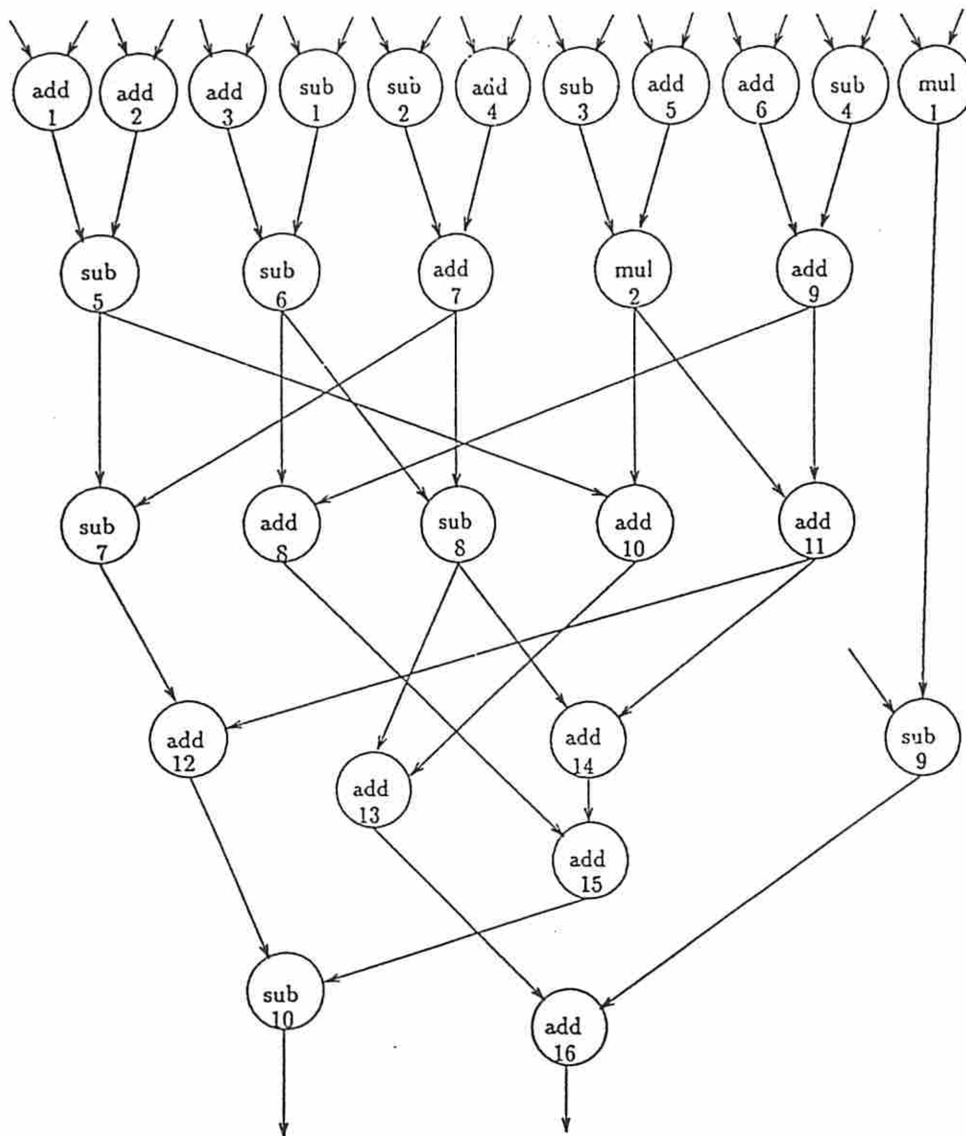


Figure 5.3: Example Dataflow Graph

5.3 Control Path/Data Path Bitwidth Tradeoff Model

Given the types of control path/data path tradeoffs, an analytical model is derived for the most useful type: bitwidth tradeoffs. The results obtained by this model are dependent upon the control path area prediction derived in Chapter 3; thus, it is only the control area model developed earlier which allows a bitwidth model to be developed.

A common tradeoff to apply to any behavior is bitwidth modification of the operators. The usefulness of bit-serial operators has been illustrated; a commercial package even implements a user-defined degree of serialization [HC88]. Consequently, a model for evaluating bitwidth effects on the total area of a given dataflow graph will be derived. Terminology used throughout the remainder of this section is described below.

- A_{dp} is the area of the data path.
- A_{cp} is the area of the PLA controller.
- $c_1 \ldots c_4$ are area constants reflecting the area of a PLA controller from Equation 3.2.8 in Chapter 3 where

$$A_{cp} \approx c_1(2i+o)p + c_2p + c_3(2i+o) + c_4 \tag{5.3.1}$$

  and i, o, and p are the number of inputs, outputs, and product-terms respectively
- ζ is the number of states in the PLA.
- A_{mux} is the area of a 1-bit 2:1 multiplexer. The area of an n-bit m:1 multiplexer is approximated by

$$A_{mux}^{m,n} \approx n \times (m-1) \times A_{mux} \tag{5.3.2}$$

- A_{reg} is the area of a 1-bit register. The area of an arbitrary n-bit register is

$$A^n_{reg} \approx nA_{reg} \tag{5.3.3}$$

- A^n_{cntr} is the area of an n-bit binary counter with reset capability. Assuming the use of a ripple-carry counter, then

$$A^n_{cntr} \approx nA^1_{cntr} \tag{5.3.4}$$

A dataflow graph consisting of a single operation (of the right type) can be implemented to perform its operation fast with parallel logic or cheap with serial logic as depicted in Figure 5.4. Here, a b-bit operation of type *op* can be implemented using $\frac{b}{4}$-bit operations upon each subset of the data. Multiplexers select which subset to operate upon and registers hold the intermediate results.

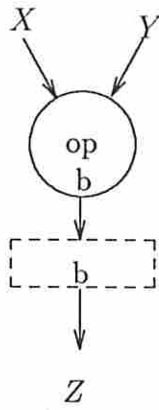
5.3.1 A Simple Model

For the initial model, only the incremental area of the control and data path is analyzed, including multiplexers and a register after each micro-cycle. It is assumed that

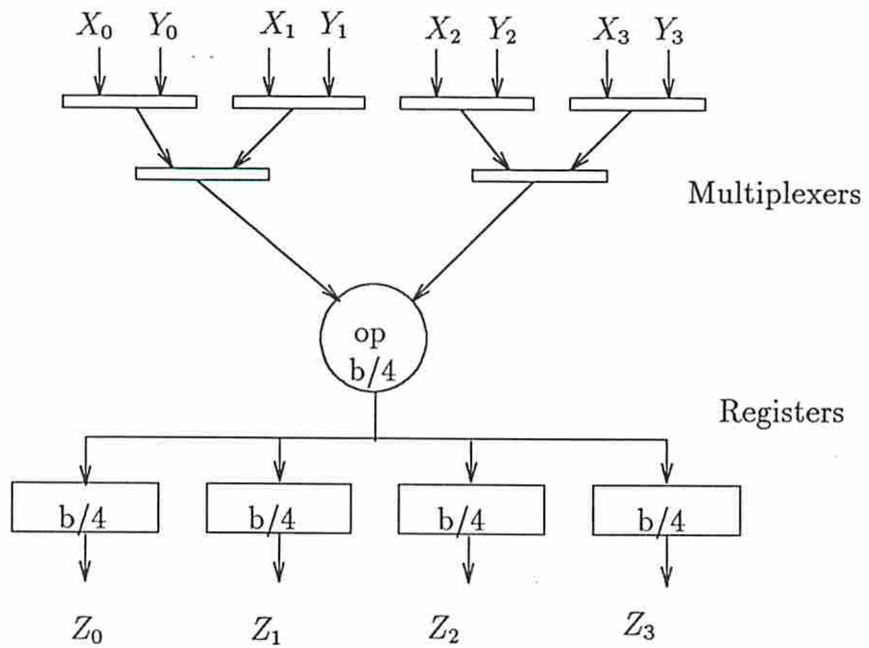
- a dataflow graph partitioned into stages exists,
- the hardware type which implements each operation has been selected,
- the particular operation has been scheduled, but *not* allocated hardware,
- a PLA controller exists, and
- the constructed PLA has ζ states to operate the original (unmodified) data path.

Clearly, sharing of this hardware with other portions of the design impacts bitwidth tradeoffs. For the moment, it is assumed that the operator is not shared. This assumption will be addressed later.

Assume that allocation of the full b-bit hardware would increase the data path area by A_{bi-op} and is *bit-independent*, i.e. the resultant binary value of a given bit position is independent the values of any other bit position. (For example, **1's complement** and **xor** are bit-independent whereas **add** and **compare** are



(a)



(b)

Figure 5.4: Serial versus Parallel Implementation

bit-dependent.) *Op* can be implemented in the data path by a single operator of bitwidth $\geq b$ or b operators of 1-bit. If n is the number of operators of bitwidth b' required to implement *op*, then

$$n = \left\lceil \frac{b}{b'} \right\rceil \qquad (5.3.5)$$

and n, the serialization quotient, can range from 1 to b.

For the following equations, hardware assignment for a single serialized operation will be examined in isolation. Assume that the operator has been divided into integral parts (e.g. $\frac{b}{n}$ is an integer) where only a single operator is repeatedly used (in a loop) to implement the operation. Since intermediate partial results need to be stored, registers will be needed; the combined registers hold the complete result after the last partial operation is finished. Also, multiplexers will be needed to select the appropriate input for each partial output. The additional data path area (exclusive of wiring) required to implement *this* operator, ΔA_{dp}, is

$$\Delta A_{dp} \approx \frac{A_{bi-op}}{n} + bA_{reg} + \frac{yb(n-1)}{n} A_{mux} \qquad (5.3.6)$$

where A_{bi-op} is the area of a b-bit bit-independent operator and y is the number of inputs of bitwidth b for the operator. (Typically, y has a value between two and four.) Note that although the number of distinct registers increases with serialization, the total register bitwidth (and area) is unchanged by serialization as reflected by the second term. The multiplexer area contribution is *worst-case*.

The controller area to operate this additional arbitrary serial path depends upon the current size of the PLA. The controller state bit count increase, δ, is the difference between the original combinational and an arbitrary sequential implementation of the operator as defined in Equation 5.3.7. Since the original schedule reflected no serialization ($n = 1$), only the additional serial states impact control area. Again, this is an upper bound analysis where *any additional states added through serialization cannot use existing control states*, thereby impacting the control area. For example, if the number of control states increases from 14 to 17, then the number of state bits increases from 4 to 5. The controller state bit increase is

Table 5.1: Sensitivity of δ to n and ζ

δ	ζ									
n	2	5	10	20	50	100	150	200	250	300
1	1	-	-	-	-	-	-	-	-	-
2	1	-	-	-	-	-	-	-	-	-
4	2	1	-	-	-	-	-	-	-	-
8	3	1	1	-	-	-	-	-	1	-
16	4	2	1	1	1	-	-	-	1	-
32	5	3	2	1	1	1	-	-	1	-
64	6	4	3	2	1	1	-	-	1	-

Table 5.2: Minimum and Maximal Change in PLA Parameters versus n

Parameter	Minimum	Maximum
Δi (inputs)	$\emptyset$	δ
Δo (outputs)	$\emptyset$	$n-1+\delta$
Δp (p-terms)	$n-1^{*}$	$n-1$

* Assumes new states are required in PLA, else this term is zero.

$$\delta = \lceil \log_2(n-1+\zeta) \rceil - \lceil \log_2 \zeta \rceil \tag{5.3.7}$$

Using Equation 5.3.7, Table 5.1 shows how the sensitivity of δ decreases rapidly with increasing ζ until it becomes nearly independent of n. Notice, however, that near the "power-of-2" boundaries an extra state bit is likely to be added.

Given n and δ, the upper and lower bounds for the changes in the number of inputs, outputs, and product-terms can be determined as provided in Table 5.2. The number of inputs is only affected by state-bit increase. Product-term count is adjusted by the number of additional states. The output count is affected by the state-bit increase (if any) and the number of additional register control lines needed. Note that control of this serialized operation is done in consecutive states; additional multiplexer control lines are not needed since the controller state bits can be used as described in Chapter 3.

A worst-case incremental area associated with the controller can be derived using Equation 5.3.1 and substituting the maximum values listed in Table 5.2 for Δi, Δo, and Δp. The incremental cost of the controller, A_{cp}, is the difference between the controller for the serialized data path and the original control which is

$$\begin{aligned}
\Delta A_{cp} &\approx A_{cp}(i+\Delta i, o+\Delta o, p+\Delta p) - A_{cp}(i,o,p) \qquad (5.3.8)\\
&\approx c_1(2\Delta i+\Delta o)p + c_1(2i+o)\Delta p + c_1(2\Delta i+\Delta o)\Delta p + c_2\Delta p + c_3(3\Delta i+\Delta o)\\
&\approx (c_1 p + c_3)(3\delta+n-1) + c_1(n-1)(2i+o+3\delta+n-1) + c_2(n-1)
\end{aligned}$$

As n increases, the incremental data path area drops roughly as $\frac{1}{n}$ and incremental control path area rises as n^2. The minimum serialized area, or "boundary point", is where the sum of $(\Delta A_{cp}+\Delta A_{dp})$ is minimum with respect to n. Beyond the boundary point, total area *and* time increase yielding only inferior designs.

$$\begin{aligned}
\frac{\partial}{\partial n}(\Delta A_{cp}+\Delta A_{dp}) &= 0 \quad (5.3.9)\\
\frac{\partial}{\partial n}\Delta A_{cp} + \frac{\partial}{\partial n}\Delta A_{dp} &= 0\\
-\frac{A_{bi-op}}{n^2} + \frac{yb}{n^2}A_{mux} + (c_1 p + c_3 - c_1 + nc_1)(3\frac{\partial\delta}{\partial n}+1) + c2 &+\\
c_1(2i+o+3\delta+n-1) + (3\delta+n-1)c_1\frac{\partial p}{\partial n} &+\\
c_1(n-1)(2\frac{\partial i}{\partial n}+\frac{\partial o}{\partial n}) &= 0
\end{aligned}$$

Although in a continuous domain, the problem will be migrated to a discrete domain under the assumption that only a small incremental value for n is considered. Hence, the partial derivatives for i, o, and p can be substituted by Δi, Δo, and Δp. These, in turn, can be replaced by the maximum parameter change values of Table 5.2 and the equation solved for the operator area at the boundary point, $A_{bi-op}b.p.$.

$$\begin{aligned}
A_{bi-op}(b.p.) &\approx 2c_1n^2(i-1) + c_1n^2(p+o+n+3\delta) + n^2(c_2+c_3) \quad (5.3.10)\\
&+ 3n^2\frac{\log_2 e}{n-1+\zeta}(c_1 p + c_3 - c_1) + c_1 n^3(3\delta+1)
\end{aligned}$$

With this equation, the area of the original unserialized operator, A_{bi-op}, can be related to the serialization quotient, n, which results in the minimal circuit area increase. For a given bit-independent operation, its optimal bitwidth implementation can be computed with respect to area. It remains to resolve the PLA input, output, and product-term values.

The basic PLA parameters of Equation 5.3.11 are used in conjunction with Equation 5.3.10 to compute the actual "boundary point" where o_{stage} and o_{reg} are the stage control and register control PLA outputs, respectively. R is the number of original register control lines and ΔR is the additional number of register control lines, if any, required for serialization.

$$\begin{aligned} i &= \lceil \log_2 \zeta \rceil \\ o_{stage} &= \zeta + \lceil \log_2 \zeta \rceil \\ o_{reg} &= R + \Delta R + \lceil \log_2 \zeta \rceil \\ p &= \zeta \end{aligned} \tag{5.3.11}$$

Assuming that any additional control steps require register control, then the change in output lines is the same for either the register control or stage control model. The stage control model will be used throughout the remainder of this section.

A graph showing the tradeoff between incremental control path and data path area versus serialization quotient is included in Figure 5.5 for initial controller states of $\zeta = 1, 10, 100$, and 500. The incremental area is associated with allocating the operator and serial scheduling impact on the controller for some arbitrary design. (If this serialization occurred along some conditional path controlled by the same PLA, then the incremental number of control states *might* be greater than that shown here.) To see the effects of operator serialization only upon the tradeoff, multiplexer and register effects were not included in computing $\Delta Area_{cp}$ and $\Delta Area_{dp}$. As expected, the optimal serialization quotient falls as the size of the PLA increases. Clearly, choosing an incorrect serialization quotient can have a detrimental effect on area when a large PLA is being extended.

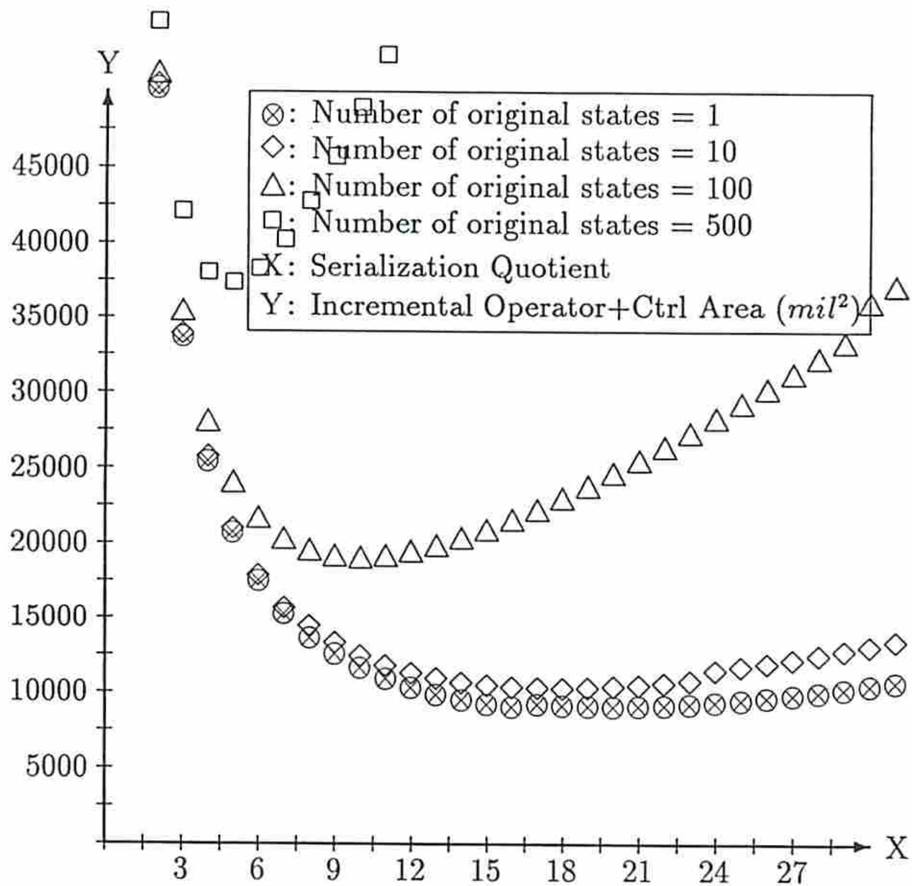


Figure 5.5: Incremental Area vs Serialization Quotient for $A_{bi-op} = 100000$

From this derivation, an optimal n can be computed for a given operator size (A_{bi-op}) with an initial size for the PLA. Figure 5.6 shows the serialization factor for any given bit-independent operator and number of current control states. Choosing the best controller and n value is straightforward. For example, assume the operator area is 5000 mil^2. A vertical line at $(1000 \log_{10} 5000) = 3700$ in Figure 5.6 intersects with the boundary line of 10 control states between four and five. Hence, the design with an n value of four has the least cost (assuming it is feasible); the actual cost of the minimum can then be computed from the sum of Equations 5.3.6 and 5.3.8.

Considering multiplexers and registers affects the boundary curve by introducing a constant term, with the resulting curves shown in Figure 5.7. As compared to Figure 5.6, the serialization quotient n for the boundary point in Figure 5.7 has been reduced for a given design.

Even in this small example, the discrete nature of the results is evident. A likely situation is where the serialization quotient cannot be realized. In this case, the first usable value *on either side* of this serialization quotient should be evaluated using Equations 5.3.6 and 5.3.8 to find the minimum. Although the larger serialization quotient yields an inferior design in the continuous space of the model, it may still be a viable design in the discrete serialization space.

5.3.2 Bit-dependent Operators

A second type of operator is the *bit-dependent* type. These operators, such as add and compare, must be performed in a particular order and have an additional cost to retain the dependencies. For example, an **add** is performed from the low-order to the high-order bits with a carry bit retained at each micro-cycle. On the other hand, the **compare** could have two retained status bits: less-than and greater-than. There are more complex issues regarding serial implementation of a compare, such as "early out" consideration for greater-than or less-than, which would eliminate the need of such bits. However, "early out" is only possible for high-order to low-order comparisons.

Changes to the bit-independent model for *bit-dependent* operators are minimal. One more term is defined, r, the number of "retained" bits (which is *carry*

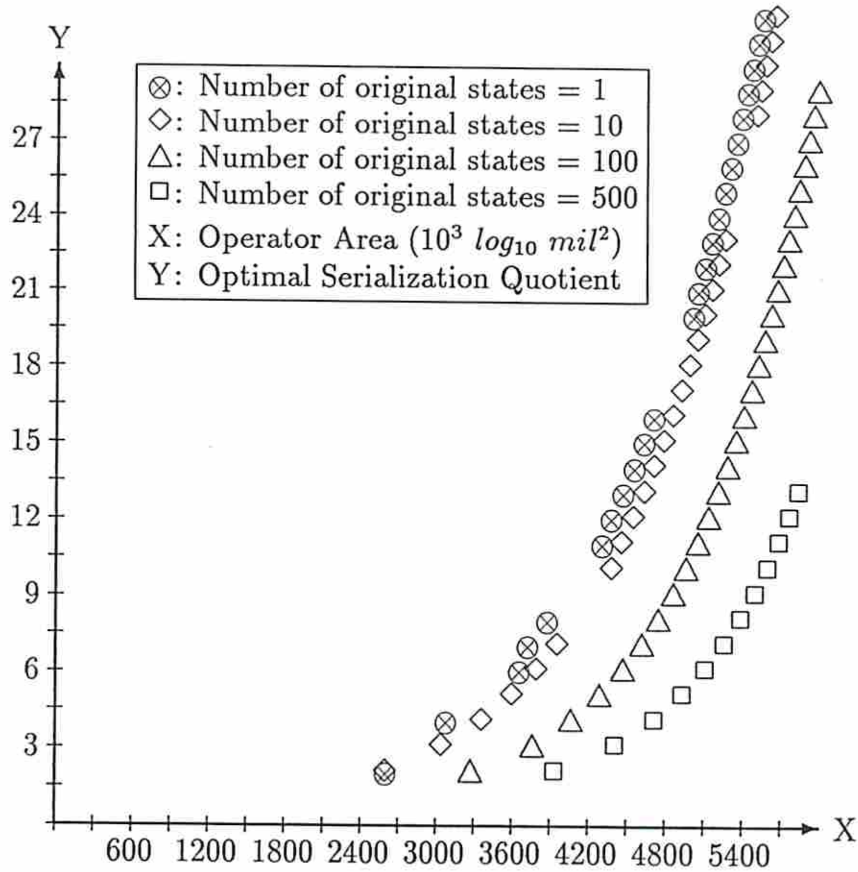


Figure 5.6: Bit-dependent Operator Area versus Maximum Serialization

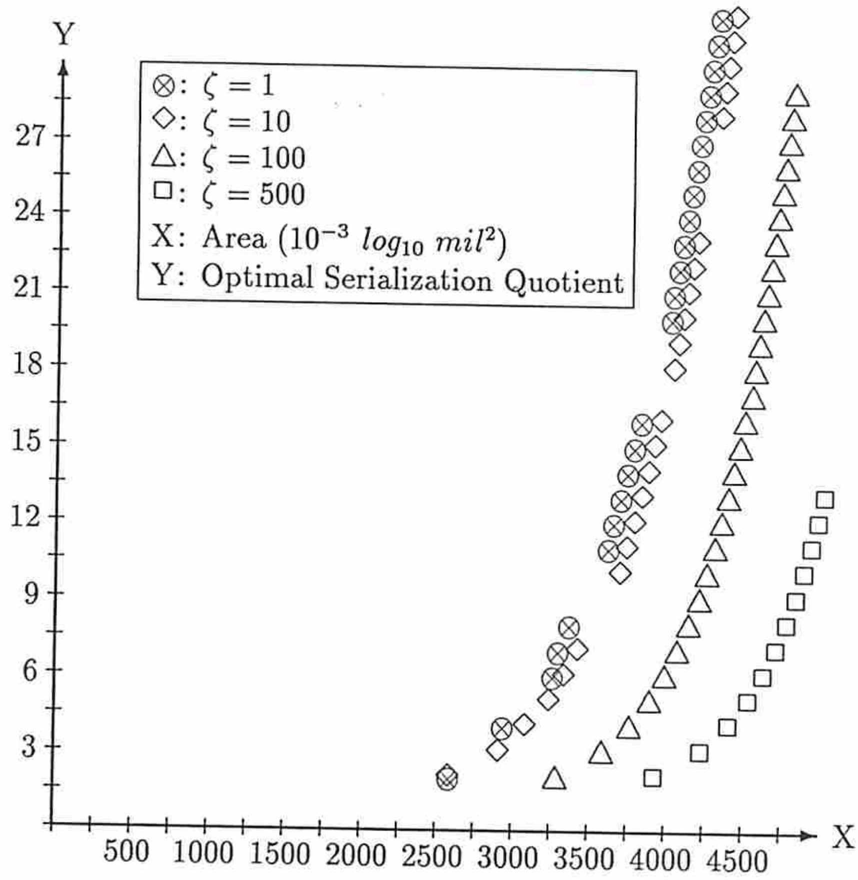


Figure 5.7: Best Operator Area versus Serialization (multiplexers and registers)

for add, *borrow* for subtract, etc.). These retained bits are stored in registers between iterations (such as a carry-bit for an adder). Furthermore, the retained bits are only selected *after* the first iteration; the retained bits are initialized on the first loop (which is zero for the **adder**). Thus, a multiplexer is also needed for each retained bit. The incremental data path area for the bit-dependent operator becomes

$$\Delta A_{dp} \approx \frac{A_{bd-op}}{n} + (b+r)A_{reg} + (n-1)(\frac{by}{n} + r)A_{mux} \quad (5.3.12)$$

where A_{bd-op} is the area of a bit-dependent operator.

Control of *bit-dependent* operations may require one additional control line for multiplexing the "retained" bits. For the first state, a constant value (typically zero) is input into the operator for one side of the multiplexer; remaining cycles use the previous output value of the retained bits available at the other input of the multiplexer. The PLA outputs and boundary point operator area are modified as follows:

$$o_{reg} = R + \Delta R + \lceil \log_2 \zeta \rceil + 1 \quad (5.3.13)$$

and

$$A_{bd-op}(b.p.) \approx A_{bi-op}(b.p.) - rn^2 A_{mux} \quad (5.3.14)$$

with $A_{bi-op}(b.p.)$ defined in Equation 5.3.10. This further flattens the curve shown in Figure 5.7.

5.3.3 Special Operator Bitwidth Considerations

The final class of operators has been labelled "special operator" because they expand into complex implementations of two or more operations. Two members of this category are multiply and divide. A distinguishing feature is that expansion complexity is some function of n which is non-linear.

As an example, Figure 5.8 depicts one method for generating lower bitwidth multipliers, with a serialization quotient of $n = 2$ demonstrated here. The area impact of the multiplier data path for a general n and bitwidth b (where the

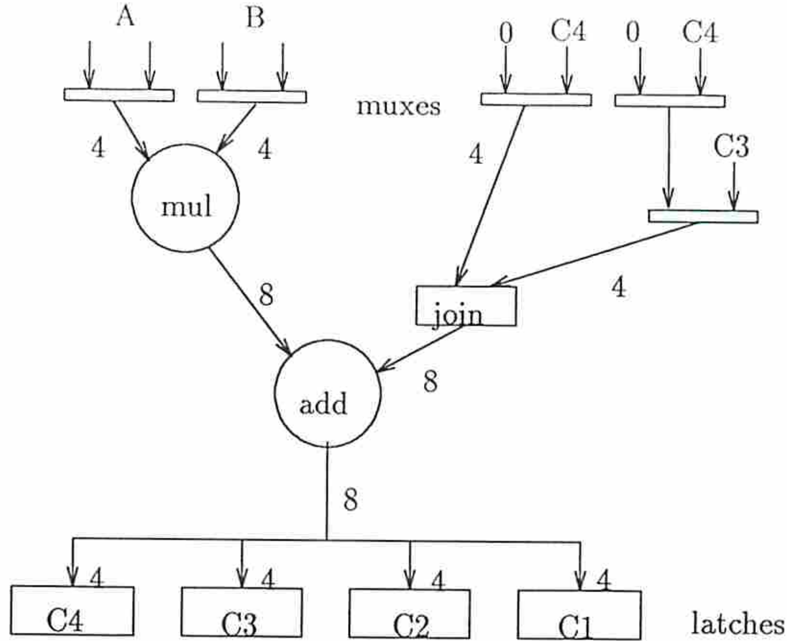


Figure 5.8: 8-bit Multiply using 4-bit Multiplier

number of inputs is fixed at two) is

$$\Delta A_{dp} \approx A_{mul}^{b'} + \frac{2b}{n} A_{add} + (\frac{yb}{n}(n-1) + 3) b A_{mux} + 2bA_{reg} \tag{5.3.15}$$

Based upon one type of hardware multiplier, the area of the multiplier can be estimated as

$$A_{mul}^{q} \approx 2^{\frac{q}{2}} A_{mul}^{2} \tag{5.3.16}$$

where q is the number of bits and A_{mul}^2 is the area of a 2-bit multiplier. Hence,

$$A_{mul}^{b'} \approx 2^{\frac{b}{2n}} A_{mul}^{2} \tag{5.3.17}$$

Other methods for implementing multiply and divide would first need to generate an equation similar to Equation 5.3.15 with which to derive the boundary point line.

5.3.4 Example using the small model

To demonstrate the control path/data path discrete model bitwidth tradeoff, a small dataflow graph was chosen as shown in Figure 5.9, with 16-bit data

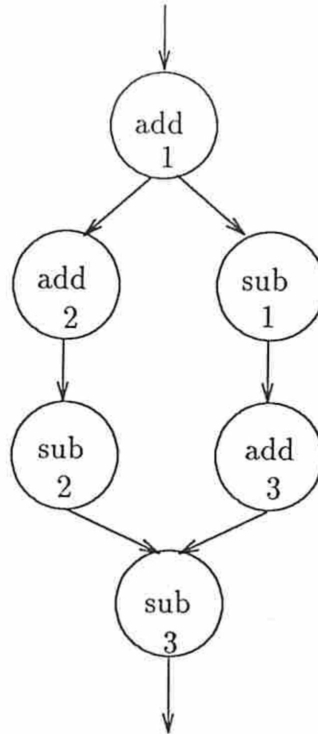


Figure 5.9: Small Dataflow Graph

paths. *Add2* was implemented using smaller bitwidth adders. The first two rows of Table 5.3 contains the results of the original design and optimal bit-serial implementation predicted by the model. The third row shows the best value obtained through exhaustive search of all power-of-2 add bitwidths for *add2* (from 1 to 16) and its associated PLA area as synthesized by **MAHA** and the Berkeley PLA tools, respectively.[5.1]

[5.1] Bitwidths of 1, 2, 4, 8, and 16 were attempted.

Table 5.3: Analysis of Serialization Factor #1 for the Example shown in Figure 5.9

Serial Method	*add2* bitwidth	Time-steps	Area (mil^2)		
			Data Path	Controller	Total
None	16	4	13256	768	14024
Model	4	7	10538	1250	11788
Actual	4	7	10618	1368	11986

Table 5.4: Analysis of Serialization Factor #2 for the Example shown in Figure 5.10

Serial Method	*add12* bitwidth	Time-steps	Area (mil^2)		
			Data Path	Controller	Total
None	16	15	77216	3264	80480
Model	4	18	74162	3868	78030
Actual	4	16	74066	3443	77509

This example demonstrates that the expected bitwidth is the same as that found through exhaustive search. It also highlights a shortcoming of the current model: its local scope. Since *add2* might share hardware with *add1* and/or *add3* prior to serialization, the implementation having the lowest area may have been missed. Secondly, serialization is restricted by any system-level time constraint which may implicitly limit the degree of serialization.

A second example highlights another limitation: the bitwidth expansion model assumes one needs additional control states and output control lines. Here, the 16-bit adder labelled *add12* in the random dataflow graph of Figure 5.10 was chosen for analysis with results in Table 5.4. Despite the very small change in area, the model finds the correct serialization quotient. Note that only one more control state was added for the optimal bitwidth tradeoff versus three states for the computed serialization. Bitwidth expansion which does not affect the number of control states would potentially allow for more serialization than the current model predicts.

5.3.5 Limitations of the Control/Data Bitwidth Model

The model given in this section derives the impact on both the data path and controller of serializing one operation of a given type. If there is more than one operation of this type, they all might be serialized depending upon the data path constraints. For example, if a tight time constraint existed, only those operations with sufficient slack time might be serialized. Conversely, if the constraints are such that the critical path operations can be serialized, then it is possible that

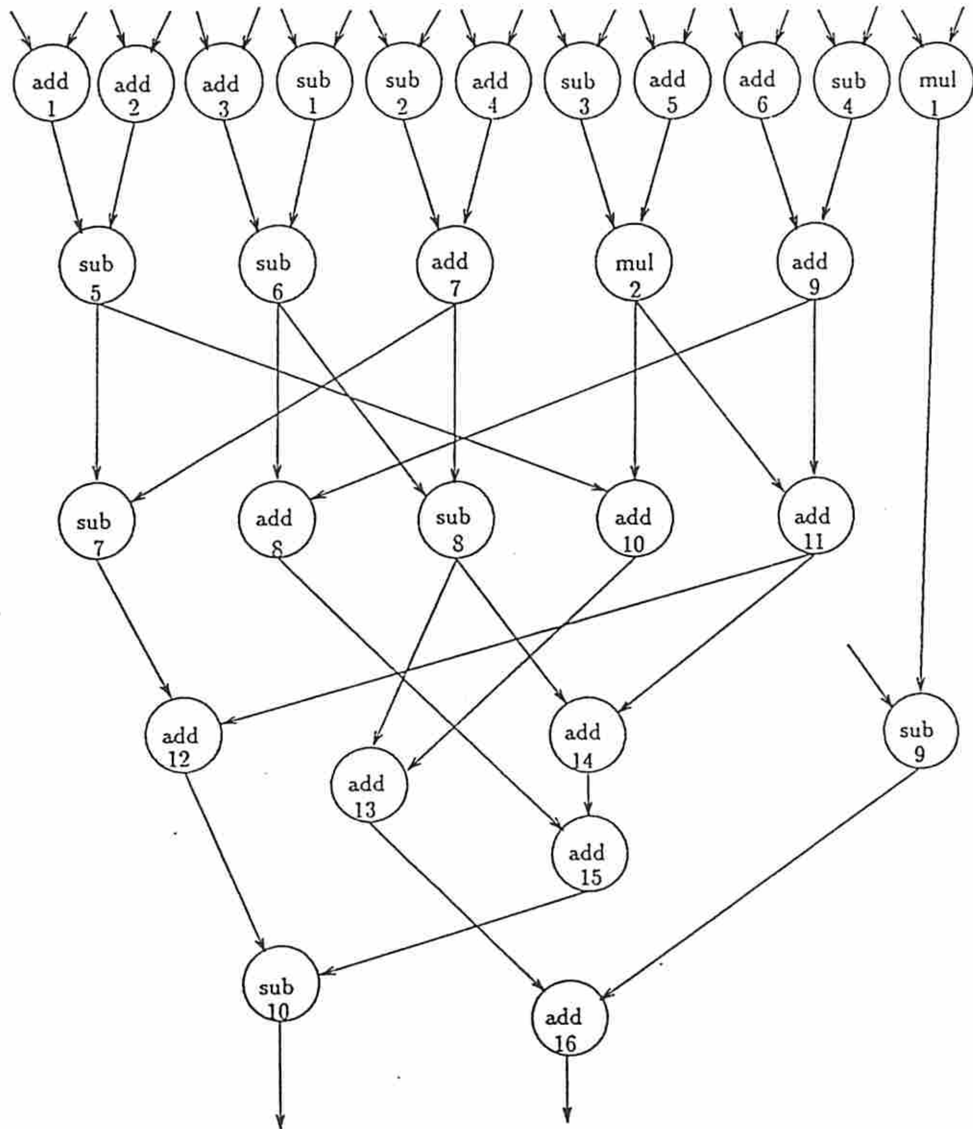


Figure 5.10: Random Graph

all operations of this type can be serialized. Note, however, that the critical path *could* change if serialization along any path is extensive.

Data path cost can also vary depending upon the graph topology. Hardware modules of the desired reduced bitwidth may already exist in the data path which entails no additional hardware area for serialization. Further, after transforming the first operation of a given type, serializing the next might not impact the data path cost. Detailed scheduling information is required to resolve this issue.

In this section, the control impact of serializing a single operation was derived. However, when another operation of *any* type is now serialized, the control cost could range from the model estimate down to zero. The latter case occurs when the operation being serialized has available control states. Furthermore, the control model does not consideration serialization in the presence of conditional paths. The model presented here is thus limited to a local, non-conditional branch, single operation view of the design. Extending it would require knowledge about the connectivity and scheduling in the dataflow graph.

For a single operation having multiple occurrences in the dataflow graph, it is evident that serialization is unlikely to be a binary decision. Given a partial conversion, selection of which operations to serialize becomes an issue that also extends to multiple operation types. One candidate solution technique is the use of integer programming to not only determine which operations to serialize, but to what degree. However, since the dataflow graph, resources, and timing are all potentially altered by a single transformation, it is unlikely that a single closed form model could be derived.

Although there are seemingly numerous limitations to the model given here, its best use would be to become part of a larger transformation engine. By having a model, even one with a limited view, it is possible to make decisions regarding which operations of the given operation type to serialize. Exploratory single-point changes could be made quickly while an intelligent engine guides the search based upon the changes which have been made. It is in this role that the model becomes of value.

5.4 Summary

In this chapter, control path/data path tradeoff types were discussed. Three implicit types of tradeoffs were identified: composition, decomposition, and sequential/combinational. The last tradeoff type includes bitwidth tradeoffs for which an analytical model was developed. Finally, this model was validated against some examples and limitations presented.

It is apparent that deriving analytical models for all different types of tradeoffs would be extremely difficult due to the complexity of the problem. Migration of functionality by altering the bitwidth alone revealed that global analysis is necessary to determine the optimal solution. These limitations can be overcome by resorting to prediction tools which are able to achieve good solutions without the cumbersome execution time associated with synthesis. Tradeoffs could be applied and their impact quickly assessed. This approach is explored in the next chapter.

Chapter 6

Control Path/Data Path Tradeoff Evaluation

6.1 Introduction

In Chapters 2 through 5, discussion centered upon mechanisms which enable a user to perform system area/time analysis. Combined with other tools in the ADAM system, a complete capability to estimate as well as perform actual synthesis for both the data path and control path is available.

In this chapter, the focus is upon the application and evaluation of control path/data path tradeoffs. A method for determining the quality of the tradeoffs is described and a variety of examples dataflow graphs are modified and analyzed. Finally, tradeoff trends observed during the experiments are discussed.

6.2 Control Path/Data Path Tradeoffs Evaluation

Given the types of control path/data path tradeoffs, the effects of applying specific tradeoffs need to be evaluated. There are two approaches for determining the effect of applying a given tradeoff: analytical and empirical. The analytical approach is usually the more desirable approach; given a system of equations, one could readily determine the outcome of applying a transformation. Furthermore, it might be possible to apply the inverse equation set and thereby derive the best solution given the design objectives. As was shown in the last chapter, the discrete nature of the design space makes it extremely difficult to derive any general model. Thus, the analytical approach has limited application.

The second approach for evaluating tradeoffs is synthesis of each design or set of designs as transformations are applied. Although practical for small designs, this is not viable for large circuits. An acceptable alternative is to predict the outcome using estimation techniques. Both of these concepts will be explored.

6.2.1 A Methodology for Evaluating Tradeoffs

A general approach for evaluating control path/data path tradeoffs is to process the modified behavior through a design system and assess the results. Unfortunately, this concept exhibits serious computer runtime limitations for larger implementations. Instead of synthesizing designs, would it not be faster to predict the set of designs which results when a change is made? Only when predicted designs meet user constraints would synthesis of complete designs occur.

A method for evaluating designs using coarse estimates and fine synthesis results is depicted within each vertical path of Figure 6.1. The system starts with a dataflow graph representing the behavior, a library of hardware modules, and a set of design constraints and goals. Initially, designs from the most parallel to the most serial are predicted using the estimation utilities. Should a region in the design space appear promising, actual synthesis of designs is performed in that region. The resultant predicted design curve and discrete synthesized designs will appear similar to that shown in Figure 6.2.

At this stage, many current systems would consider the design complete. However, by transforming the dataflow graph, a designer can effect further structural changes. When a transformation is applied to the dataflow graph, a new design curve is computed as indicated by the loop in Figure 6.1. Predicted designs which appear promising are synthesized, compared against the original results, and either accepted or rejected. This process continues until either the set of applicable transformations is exhausted, the design goals and constraints are reached, or some prior limitation on design time/computer resources has been exceeded.

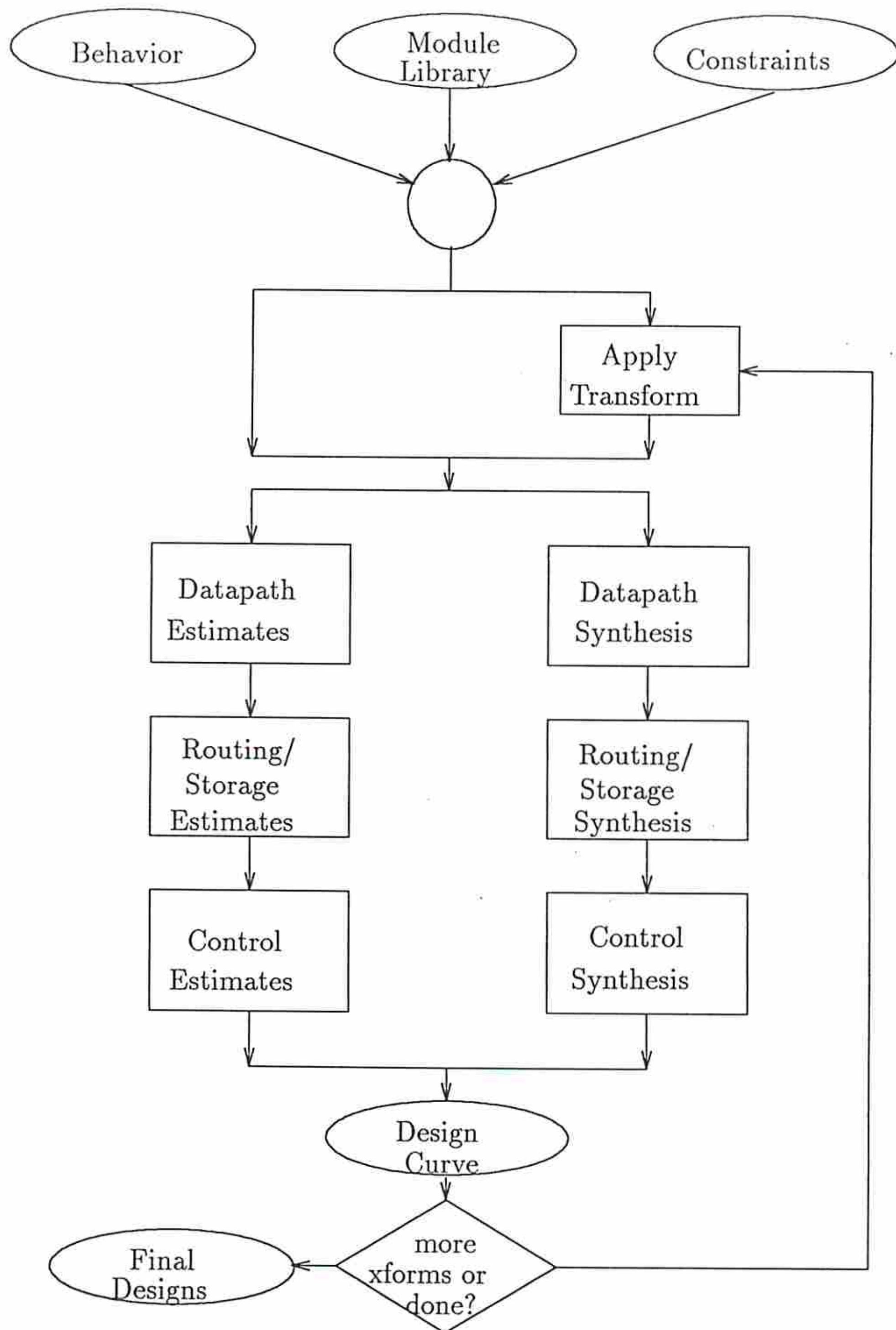


Figure 6.1: Control Path/Data Path Tradeoff Evaluation

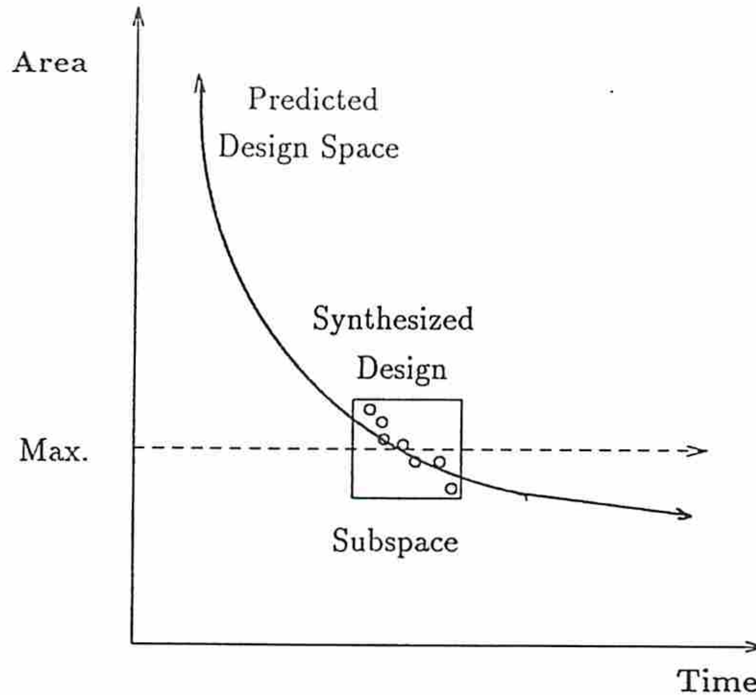


Figure 6.2: Evaluation of Design Space: Prediction and Synthesis

6.2.2 Synthesis versus Prediction: An Example

To illustrate the evaluation technique, a design curve will be produced for the AR lattice filter. Synthesis and estimation results will be compared. Available inputs are the dataflow graph and a library of modules having differing areas and delays. An area constraint of 70000 mil^2 was specified and a non-pipelined design style was chosen using the module library in Table 6.1.[6.1] First we describe the synthesis process, followed by the estimation procedure.

6.2.2.1 Design Synthesis

As defined here, design synthesis program constructs an RT (register transfer) description from the dataflow graph and chosen module set. These results, complete with hardware selection, routing, storage, detailed controller design, and wiring assignment, could be passed to a package which determines the floorplan

[6.1] The addition delays in this module library are somewhat pessimistic due to a historical error in reading a graph, and should not be taken as representative of actual modules.

Table 6.1: Module Library for AR Example

Name	Type	Bits	Area (mil^2)	Delay (nS)
mulf	multiply	16	49000	375
mulm	multiply	16	9800	2950
muls	multiply	16	7100	7370
addf	addition	16	4200	340
addm	addition	16	2880	530
adds	addition	16	1200	1510

and produces the actual chip layout. Because of the detail involved, this process is time consuming.

To begin, the program **SLIMOS** is used to select the module set [JPP88]. (As described in Chapter 1, although **SLIMOS** is intended for pipelined designs, it has shown so far to produce the best module sets for non-pipelined designs as well.) The AR lattice filter consists of 16-bit adds and multiplies. Using this designer's 60/40 rule, an initial area constraint of 28000 mil^2 was entered into **SLIMOS** (40% of 70000) and feasible module sets, each consisting of one adder and one multiplier, were chosen from Table 6.1. There are five module sets which form the entire design space: $\{(mulf, addf), (mulf, addm), (mulm, adds), (mulm, adds), (mulf, adds)\}$; $(mulm, adds)$ best meets the criteria.

Using the chosen module set and dataflow graph, **MAHA** synthesizes designs from parallel to serial. A total of eight non-inferior designs were produced as illustrated in Figure 6.3. During its search, **MAHA** internally generated 80 individual designs and discarded the remaining as inferior in time and/or area to these eight.

With scheduling and operator allocation completed, routing and storage hardware is attached by **MABAL**. Multiple executions of **MABAL** are performed on each of the eight designs in order to obtain the minimum register and multiplexer area.

Next, a PLA controller is constructed from the data path results. The Berkeley tools (PEG/ESPRESSO/EQNTOTT) are used to generate the PLA personality matrix. Layout is produced by MKPLA from this matrix. Results of the

control synthesis, multiplexer and storage allocation are also included in Figure 6.3. The total area including the data path is depicted in Figure 6.4.

Note that Figure 6.3 has "timesteps" instead of "time" along the X axis. This allows a direct comparison of area for similar predicted and synthesized designs while removing any clock cycle time differences. In reality, clock cycle time for predicted and synthesized designs can differ for the same number of timesteps. This would result in both a time and area displacement on the graph and make it difficult to compare similar designs. However, since that is more realistic, future results will reflect area and time. It should also be noted that data represents either control path synthesis of data path synthesized designs or estimated control path area for predicted data path designs. For control synthesis, **PEG** PLA descriptions were written to eliminate redundant output lines.

Finally, the wiring area of each design can be estimated using **PLEST**. **PLEST** is given the total block width for data path, register, and multiplexer hardware plus the number of two-wire nets. The version of **PLEST** used automatically computed the average wire length based upon the RCA 3 *um* CMOS library. Since **PLEST** is not an actual router, but an area estimator, it more correctly belongs in the prediction leg. However, until a router compatible with the rest of the ADAM system is obtained, **PLEST** will continue to be used to factor in wiring results.

6.2.2.2 Design Prediction

The estimated design points were produced in the following manner:

As in synthesis, **SLIMOS** is initially used to generate the module set meeting the constraints for the estimators. Using the chosen module set, the design curve is predicted using **PSADNP** for non-pipelined designs. Besides module library and operation quantities, **PSADNP** requires critical path delay; this value is obtained by executing the first portion of **MAHA** which lists the operations in the nominal critical path and associated delay time prior to actual synthesis.

The output of **PSADNP** gives the number of clock cycles, the anticipated delay, and the predicted area. Results which are closest to the actual synthesis

results are plotted in Figure 6.3. (For the remainder of this example, the remaining predicted points were dropped since a comparison is being made against the synthesized results.)

As can be seen by Table 6.4 for the design with 5 timesteps, a large error exists for time. However, the clock cycle time difference between the predicted and synthesized designs is less than half the minimum clock cycle time; this difference is magnified by the number of time steps giving the larger total time error. Fortunately, it is only in the mid-range of timesteps that such a time error occurs. For timestep larger values, the clock cycle time rapidly approaches the minimum, reducing circuit time errors.

The next step is to estimate register and multiplexer usage with **REGEST** and **MUXEST**. **REGEST** determines the storage requirements from the dataflow graph. Multiplexer estimation in **MUXEST** uses the output from the register estimation as well as the data path prediction results.

Finally, **PASTA** is used to estimate the controller area. A file which contains the data path control information is read by **PASTA**; the size of each PLA personality matrix as well as folded and unfolded area is displayed when **PASTA** is run.

6.2.2.3 Synthesis versus Prediction

During design evaluation, a choice must be made between prediction and synthesis. Prediction is faster than synthesis with an inherent risk in that predictions may be infeasible or inaccurate. A comparison between the predicted and synthesized designs for the AR lattice filter reveal some differences as indicated in Figures 6.3 and 6.4 and Tables 6.2 and 6.3. Only four designs boldfaced in Table 6.4 appear comparable; the remaining four designs are in error. The comparable designs have identical predicted and synthesized operator area (clock cycles of 1, 5, 10, 18). Hence, any error is due to routing, storage, and control.

For seven designs, the actual and predicted register count track closely. Multiplexer and control error are also not significant. For the remaining design with 10 clock cycles, register use is higher than expected (by two) contributing the predominant error. If predicted register assignment is optimal, then two values

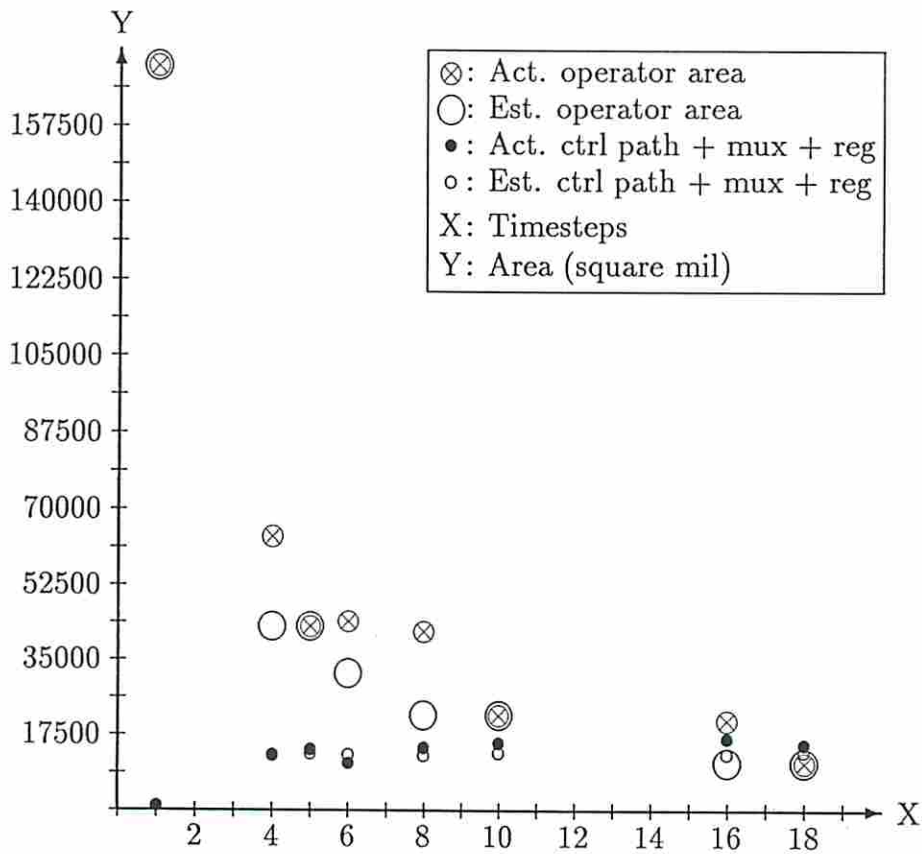


Figure 6.3: AR Filter Area Comparison

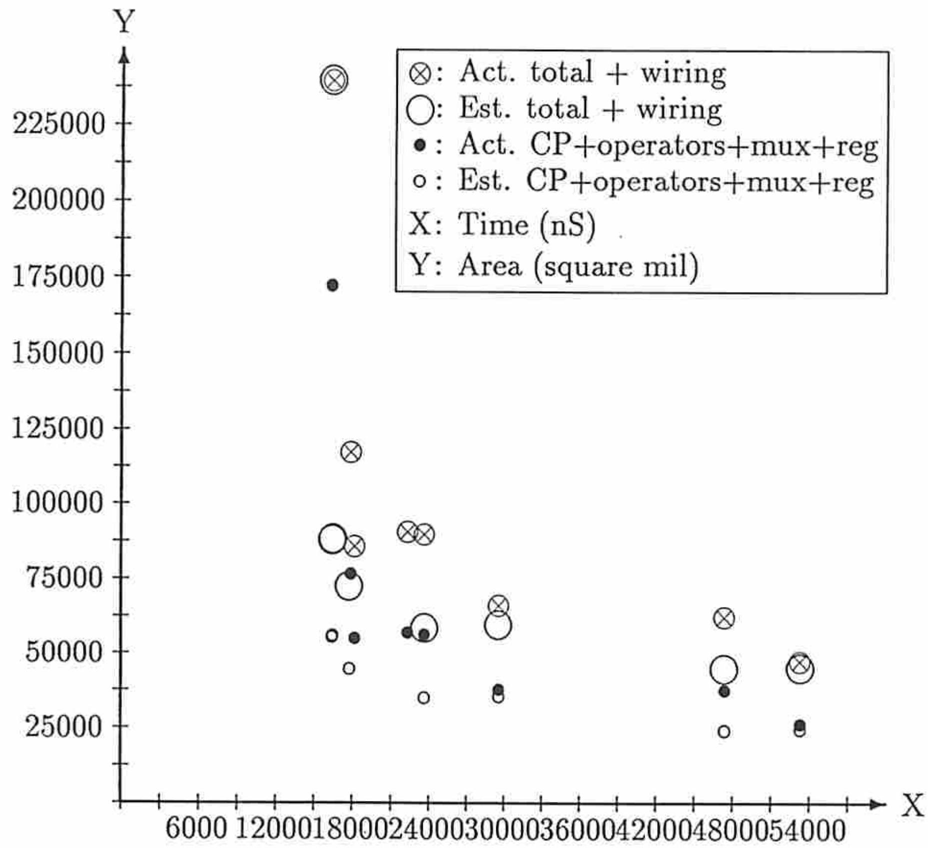


Figure 6.4: AR Filter Totals

Table 6.2: AR Filter Designs: Predicted

Clock Cycles	Op. Area (mil^2)	Regs	Muxes	Ctrl Area (mil^2)	Total w/wiring	
					Time	Area
1	171200	2	0	–	16405	239711
4	42800	5	33	578	16452	87942
5	42800	5	34	764	16485	88462
6	31800	5	33	764	17802	72617
8	22000	5	32	983	23704	58528
10	22000	5	32	1535	29630	59632
16	11000	5	30	1786	47408	44862
18	11000	5	30	2353	53334	45006

Table 6.3: AR Filter Designs: Actual

Clock Cycles	Op. Area (mil^2)	Regs	Muxes	Ctrl Area (mil^2)	Total w/wiring	
					Time	Area
1	171200	2	0	–	16405	239711
4	63600	6	32	578	17892	117485
5	42800	6	35	881	22365	90834
6	44000	6	24	999	18198	85823
8	41600	6	36	1117	23680	89972
10	22000	8	34	1665	29630	66226
16	20800	7	39	1786	47408	62442
18	11000	6	34	2353	53334	47370

in the graph are being stored longer than expected. Indeed, an examination of Figure 6.5, which depicts the scheduling generated by **MAHA**, reveals that adders on the upper left are scheduled later than necessary. By manually reassigning the adders to the earliest clock cycle possible (which does not change the resources or timing), **MABAL** allocates six registers instead of eight and the difference is explained.

For predicted and synthesized designs having the *same* operator area, the combined error in routing, storage, and control area is a small fraction of the overall area. With an average area error of 4.4% (worst case is 9.9%) and average error in time of 6.6%, clearly the prediction results are comparable.

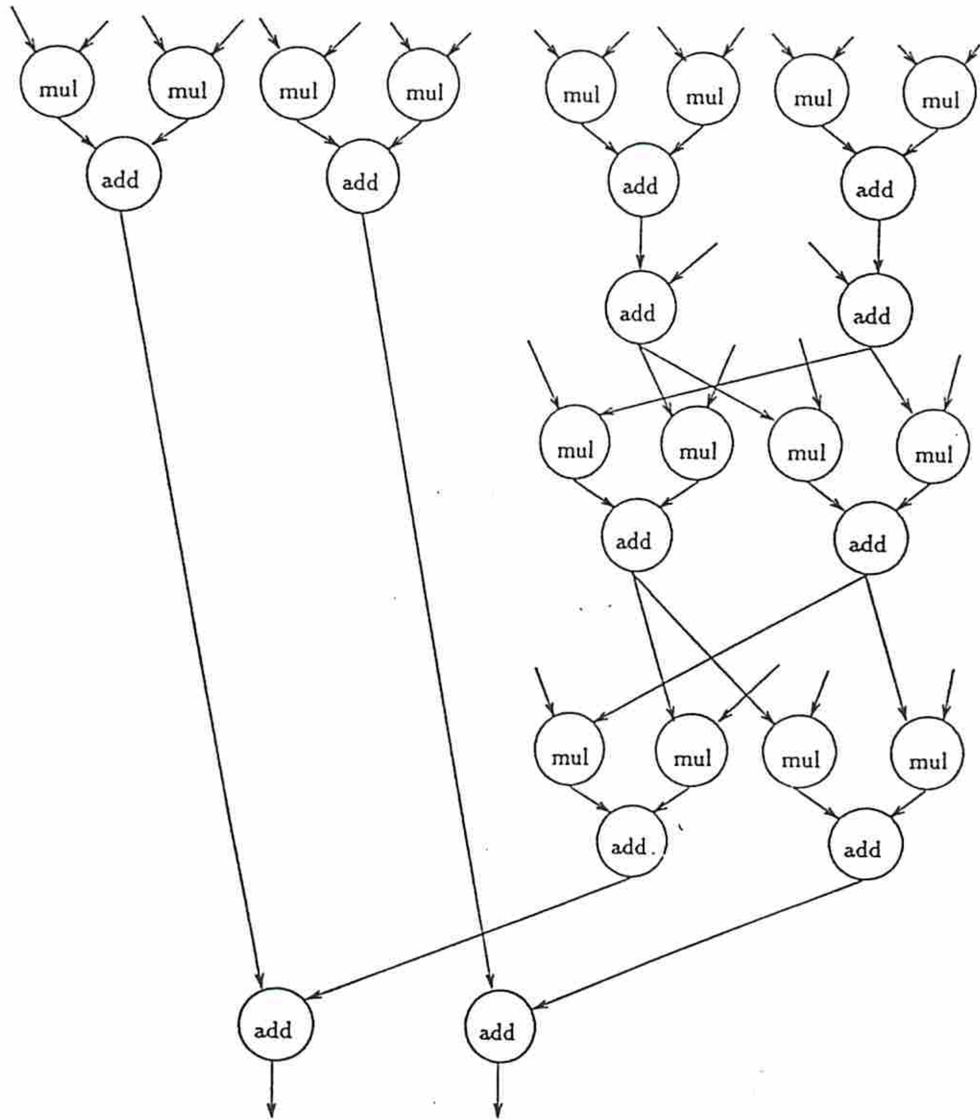


Figure 6.5: AR filter design (clock cycles = 10)

Table 6.4: AR Filter Designs: Error Summary

Clock	w/o wiring		with wiring	
Cycles	Area (%)	Time (%)	Area (%)	Time (%)
1	**0.0**	**0.0**	**0.0**	**0.0**
4	27.7	8.1	25.1	8.0
5	**1.4**	**26.4**	**2.6**	**26.3**
6	18.8	2.3	15.4	2.2
8	38.7	0.0	34.9	0.0
10	**8.0**	**0.0**	**9.9**	**0.0**
16	37.6	0.0	28.1	0.0
18	**6.9**	**0.0**	**5.0**	**0.0**

The four poorer predictions in the AR filter comparison have their error introduced during operator area estimation. Due to data dependencies which **PSADNP** does not consider, the predicted designs for timesteps of 4, 6, 8, and 16 are, in fact, unachievable with the computed resources. The cause of this is detailed by Jain in [Jai89] and will not be elaborated here. This suggests that tradeoff analysis may be relying upon inaccurate data path estimation. However, this error must be taken in context. That is, for the predictors to be usable in tradeoff analysis, they must *bound the design space changes.*

The goal of the predictors is not to establish a precise design, but to ascertain the effects of performing certain tradeoffs. Consequently, absolute errors are less relevant than trends observed when a tradeoff is applied. For a typical analysis, the predictors would be used to bound an area in the design space; the synthesis tools would be targeted to produce designs only in that region. Figure 6.6 illustrates this concept using the AR filter. The predictors give global estimates with synthesis tools producing the actual localized designs. Note how the constraint given initially is met by synthesizing only two designs rather than the entire design space. It should be noted that the 60/40 rule used by the designer was close to the actual data path to total ratio; had a different ratio been taken, additional designs might have been synthesized.

The results of Table 6.4 and Figure 6.4 suggest another simplification, as wiring area is predict to have little impact on the area error. Evaluation of

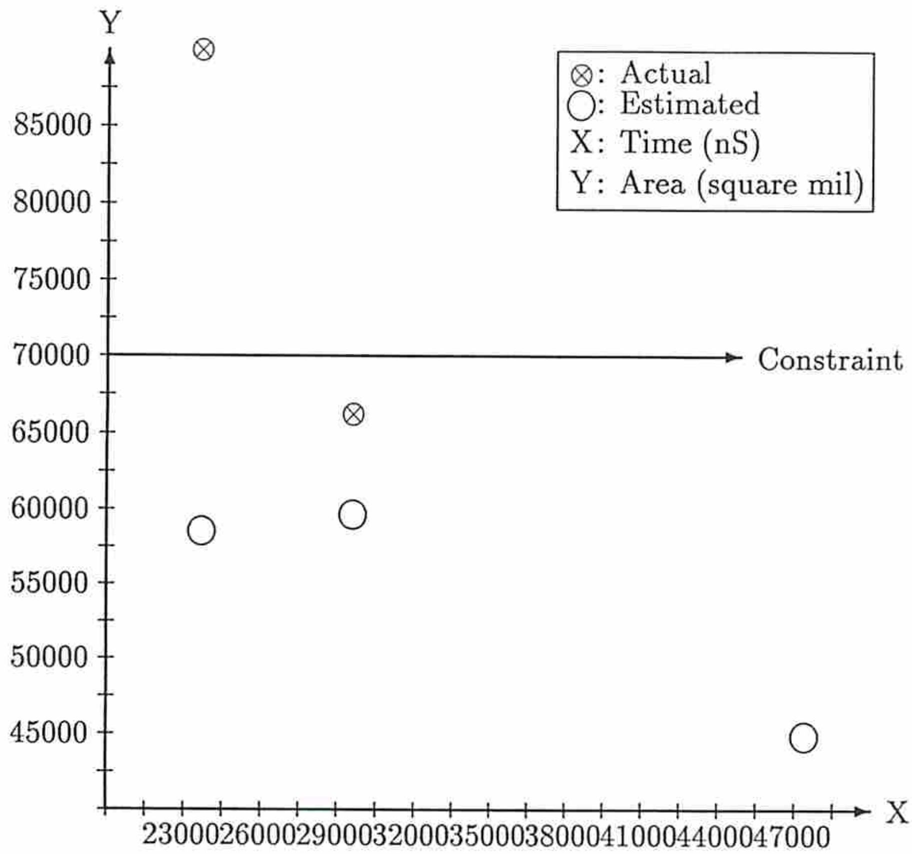


Figure 6.6: AR Filter Design Region

Table 6.5: Non-Pipelined Design Curve Computation Time (sec.)

Function	Synthesis			Prediction		
	Tool	Designs	Time	Tool	Designs	Time
Module Selection	SLIMOS	–	0.00	SLIMOS	–	0.00
Scheduling/Alloc.	MAHA	8	7.64	PSADNP	16	0.02
Routing/Storage	MABAL	8	972.16	REG/MUX	8	1.18
Controller	Berkeley	8	10.10	PASTA	8	0.12
Subtotal	–	8	989.90	Subtotal	8	1.32
Average			202.34	Average		0.17
Interconnect	PLEST	8	628.80	PLEST	8	622.64
Total	–	8	1618.7	Total	8	623.96
Average			202.34	Average		78.00

tradeoffs is reduced in complexity by avoiding this step. Only designs that are close in area might consider wiring effects as a final differentiator.

The computation time expended for tradeoff evaluation is shown in Table 6.5. Clearly, prediction is substantially faster than synthesis. Even if **MABAL** is not iterated to produce the minimum register and multiplexer assignment, synthesis is 36 times slower than prediction for this small dataflow graph. Worst case analysis gives a runtime of $O(n^2)$ for prediction; synthesis takes $O(n^4)$. Thus, even for graphs of moderate size, prediction is essential if one wishes to explore a large design space using control path/data path tradeoffs.

6.3 Tradeoff Examples

With a powerful mechanism for evaluating many parts of a design, accurately evaluating tradeoff effects is possible. In this section, the types of tradeoffs described earlier will be demonstrated using algorithms ranging from a small filter to a large coprocessor and cover both pipelined and non-pipelined architectures.

During the course of evaluation, two difficulties were not possible to overcome: pipeline synthesis and routing/storage synthesis of large graphs. **Sehwa**, the pipeline synthesis utility, does not operate efficiently on large graphs. In fact, **Sehwa** often does not complete designs of dataflow graphs with over 100

nodes and 200 edges. The first design point takes over four hours to reach with successive points taking another three to six additional hours each on a Sun 3/280. Even upon reaching a design, only a subset of the operation scheduling is displayed. Fortunately, the pipelined data path operator prediction results are indistinguishable from actual pipelined synthesis, allowing one to evaluate large designs using estimates with some assurance of accuracy.

A second difficulty was using **MABAL** to attach registers and multiplexers to large designs. Unlike **MAHA** and **Sehwa**, **MABAL** utilizes extensive information about the dataflow graph such as commutivity, explicit number of input and output lines, and sensitivity to bitwidth. At the time this work was done, there was no tool available for converting between the data path synthesis tool and **MABAL** file formats; manual reconstruction of the dataflow graph is therefore necessary. Clearly, manual transformation of large graphs is not practical. Rather, since each utility will ultimately use the EVE database for storage and retrieval of design information, a solution will be available in the future.

The solution to the latter problem was to utilize **MAHA** and **REAL** in place of **MABAL** for large examples. **REAL** is a program for register assignment whose algorithm was incorporated into **MABAL** [KP87]; thus, the results are comparable. Further, **MAHA** can be instructed to allocate multiplexers for operators. Register multiplexers can be determined by counting the true storage instances and actual operator allocation defined by **MAHA**.

For all the designs, a module library was chosen based upon the RCA 3 *um* CMOS standard cells. Coefficients for the PLA model were modified to reflect this type of technology; values in square microns are

$$\begin{aligned} c_1 &= 663.95 \\ c_2 &= 176.42 \\ c_3 &= 5505.28 \\ c_4 &= 27233.46 \end{aligned}$$

Table 6.6 contains the area and delay for the major data path modules used throughout the remainder of this chapter.[6.2]

6.3.1 Bitwidth Tradeoffs

One major category of tradeoffs is that related to bitwidth. Serializing a specific operation into smaller components allows one to trade increased circuit time for less area. In this section, the effects of serialization upon pipelined and non-pipelined designs are explored. As will be observed, the impact is different for each design style due to the method by which pipeline quality is measured.

6.3.1.1 Non-Pipelined Bit Serialization: Floating Point Processor

One design typically constructed in a non-pipelined manner is a floating point coprocessor. Since the coprocessor spends much of its time idle and often does not have instruction prefetch cognizance, the benefit of pipelining is limited. The complexity of building floating point hardware also favors a non-pipelined approach as chip size is usually the limiting factor. (The description for this example is included in Appendix H.)

There is a tradeoff between the size of the chip versus its performance. In particular, floating point hardware boasts large *multiply* and *divide* circuitry which could be bit-serialized to decrease the chip area at the expense of increased circuit delay. Non-pipelined tools will construct the design space from parallel to serial implementations given a fixed module set. Additional portions of the design space can be explored by adjusting the bitwidth of the *multiplier* and *divider*.

The example described here is a floating point coprocessor (courtesy of Michael McFarland from Bell Labs) which contains both a 64-bit *multiplier* and *divider* as well as numerous smaller bitwidth operations: *add*, *subtract*, *negate* and *compare*. Since the construction and serialization of a hardware *divider* is impractical, this was initially implemented using shift-compare-add/subtract hardware which allows it to be bit-serialized.

[6.2] The addition delays in this module library are somewhat pessimistic due to a historical error in reading a graph, and should not be taken as representative of actual modules.

Table 6.6: Module Library

Operation Type	Bit Width	Module Name	Area mil^2	Delay nS
Addition Subtraction Compare	4	add4f	960	120
		add4m	450	220
		add4s	300	540
	16	addf	4200	340
		addm	2880	530
		adds	1200	1510
	32	addf	8850	578
		addm	7540	800
		adds	2880	2530
	64	addf	18520	985
		addm	12000	1220
		adds	6900	4250
Multiply	8	mul8f	13000	360
		mul8m	5000	980
		mul8s	3600	2500
	16	mulf	49000	375
		mulm	9800	2950
		muls	7100	7370
	32	mulf	184700	390
		mulm	19208	8880
		muls	14000	21730
	64	mulf	696000	410
		mulm	38000	26730
		muls	27620	64050

Table 6.7: Multiplier/Divider Floating Point Bit-Serialization

Dataflow Graph	Area				Total	
	Operator	Register	Multiplexer	Control	Time	Area
Original	102828	1065	0	0	0	103893
	85453	10650	4520	4800	53470	105423
	83801	10650	9040	5698	80214	109189
	76401	10650	11865	5826	106972	104742
32-bit	75033	10000	6894	4800	43470	96727
	72181	10000	11660	5698	65229	99539
	50781	10000	15900	5826	86988	82507
	49581	10000	17490	6470	108755	83541
16-bit	107853	9600	10998	4953	14750	133404
	89301	9600	16074	5879	22149	120854
	66001	9600	20727	6011	29531	102339
	56201	9600	15651	7648	51737	89100
	48601	9600	16497	8772	81302	83470
	41501	9600	17343	9474	118256	77918
	41001	9600	18189	9474	155211	78264

The design space was first searched using the conventional parallel-to-serial partitioning with fixed 64-bit modules; design points are listed in the first portion of Table 6.7 as well as shown in Figure 6.7. Then, the *multiplier* and *divider* were bit-serialized into 32- and 16-bit module blocks producing a cheaper implementation. (The *multiplier* and *divider* modules always have the same bitwidth.) Moreover, the coprocessor is now generally *faster* for these serial designs. This unexpected result is due to the large delay associated with the 64-bit *multiplier* and *divider*, which dominate the clock cycle time for non-pipelined designs. (The synthesis and prediction tools used for this research assume that any operation must complete in a single clock cycle.)

For the fastest designs, the 16-bit modules yield the best results. However, this effect is due to the type of module set chosen: shift-and-add/subtract rather than carry-look-ahead or array blocks. When bit-serialized, its area and time are linearly related to its bitwidth. Thus, there is little penalty for lowering the bitwidth. This might not be the case if a tight time constraint were imposed which would result in a different module set being chosen.

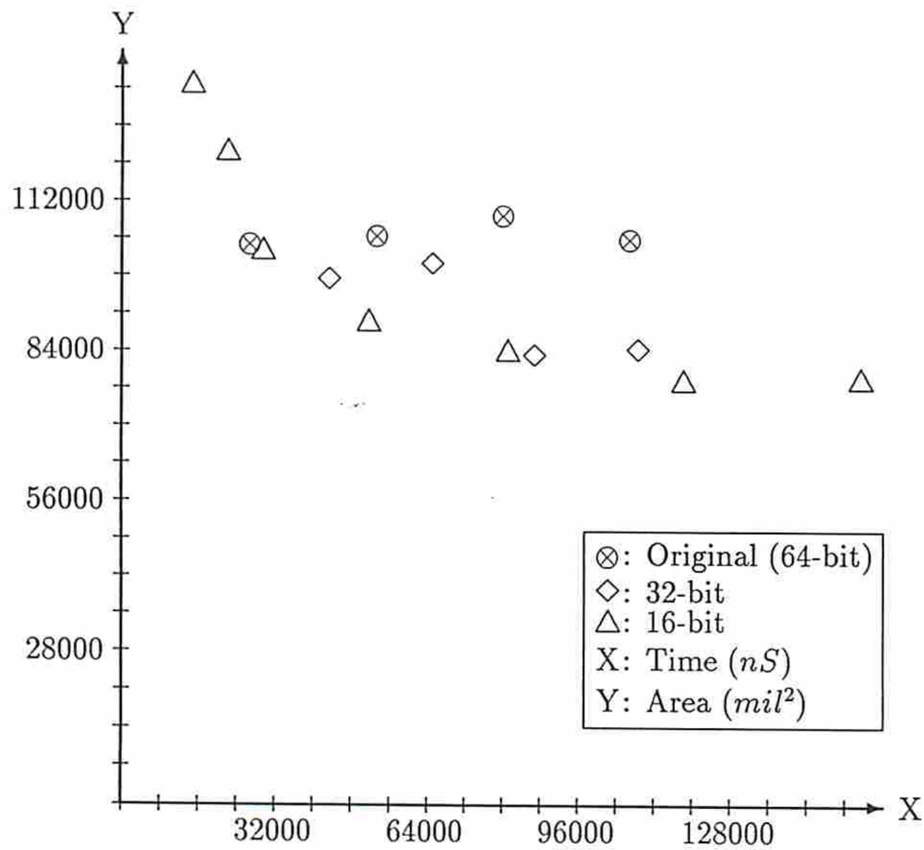


Figure 6.7: Floating Point Coprocessor: Serialize multiply/divide

Table 6.8: Add/Sub/Cmp/Neg Floating Point Bit-Serialization

Dataflow	Area				Total	
Graph	Operator	Register	Multiplexer	Control	Time	Area
Original	85453	10650	4520	7000	53470	107623
	83801	10650	9040	7500	80214	110991
	76401	10650	11865	8243	106972	107159
32-bit	57813	10000	6800	4800	43470	79413
	52081	10000	11200	5698	65229	78979
	34701	10000	13600	5826	86988	64127
	33501	10000	14800	6470	108770	64771
	30621	10000	16000	7698	174008	64319
16-bit	85353	6563	13923	4953	14750	110792
	64401	6563	18207	5879	22149	95050
	44401	6563	21090	6011	29531	78065
	33901	6563	23919	6945	44370	71328
	25101	6563	21777	8772	59160	62213
	16301	6563	21777	9474	118320	54115
	15801	6563	20349	9755	155295	52468

Serialization of the less complex operations in the floating point coprocessor was also evaluated: *add*, *subtract*, *compare*, and *negative*. Results for these implementations, where the five modules always have the same bitwidth, are shown in Table 6.8 and Figure 6.8. Total area here is less than observed for complex operator serialization, but circuit time has increased. There are three times as many of these less complex operations; hence, serialization results in a greater reduction in cost as compared to the original design curve, despite their smaller individual size.

Due to the dominant delay of the complex modules, the cost of a fast design can be reduced by serializing the remaining components. As noted in Figure 6.8, *both* time and cost are reduced even in the high speed region. However, the module set used may not be the best one as this fastest design region was not targeted by **SLIMOS**, the module selection program. One could adjust the constraints and restart the evaluation which may result in selection of a different module set.

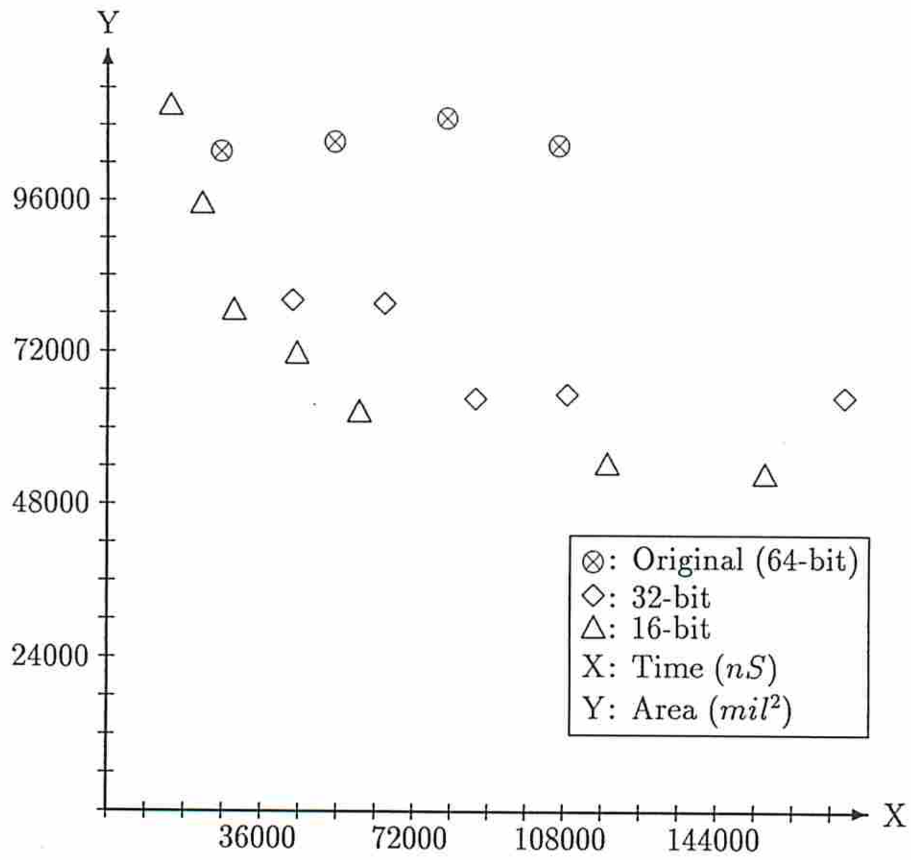


Figure 6.8: Floating Point Coprocessor: Serialize add/sub/cmp/neg

Table 6.9: Combined Floating Point Bit-Serialization

Dataflow Graph	Area				Total	
	Operator	Register	Multiplexer	Control	Time	Area
32-bit	56853	9000	5324	4800	43470	75977
	47601	9000	10648	5698	65229	72947
	29901	9000	14036	5826	86988	58763
	25501	9000	16940	6743	130506	58184
	22301	9000	17424	8243	261012	56968
16-bit	88953	8000	16400	4953	14750	118306
	64401	8000	26000	5879	22149	104280
	46201	8000	14800	6011	29531	75012
	33901	8000	16800	6945	44346	65646
	24301	8000	10800	8772	51873	81302
	15701	8000	11600	9474	118256	44775
	13701	8000	11600	9755	177384	43056

Finally, the two different transformations can be combined to produce a third range of designs as shown in Table 6.9 and Figure 6.9. Here, the difference is significant. Not only are the designs much lower in cost, but the speed is nearly unchanged. Due to the small size of the controller as compared to the data path hardware, full serialization is cost effective. A useful design region has been found as a result of bitwidth transformation.

6.3.1.2 Pipelined Bit Serialization: Elliptical Filter

There are generally two types of pipelined designs: one which only considers throughput and the other which also considers start-to-end delay. Most signal processing filters fall into the first category whereas some general purpose configurable pipelines fall into the second. For this filter example, only the throughput is of importance.

In a pipelined design where throughput (or area) is the prime consideration, serializing a given slowest (or largest) operator potentially yields a better design. An elliptical filter consisting of 16-bit *multiplies* and *adds* is a prime candidate for improvement given the goal of maximizing throughput with an area constraint of 100000. **SLIMOS** chose the fastest possible module set for the original design

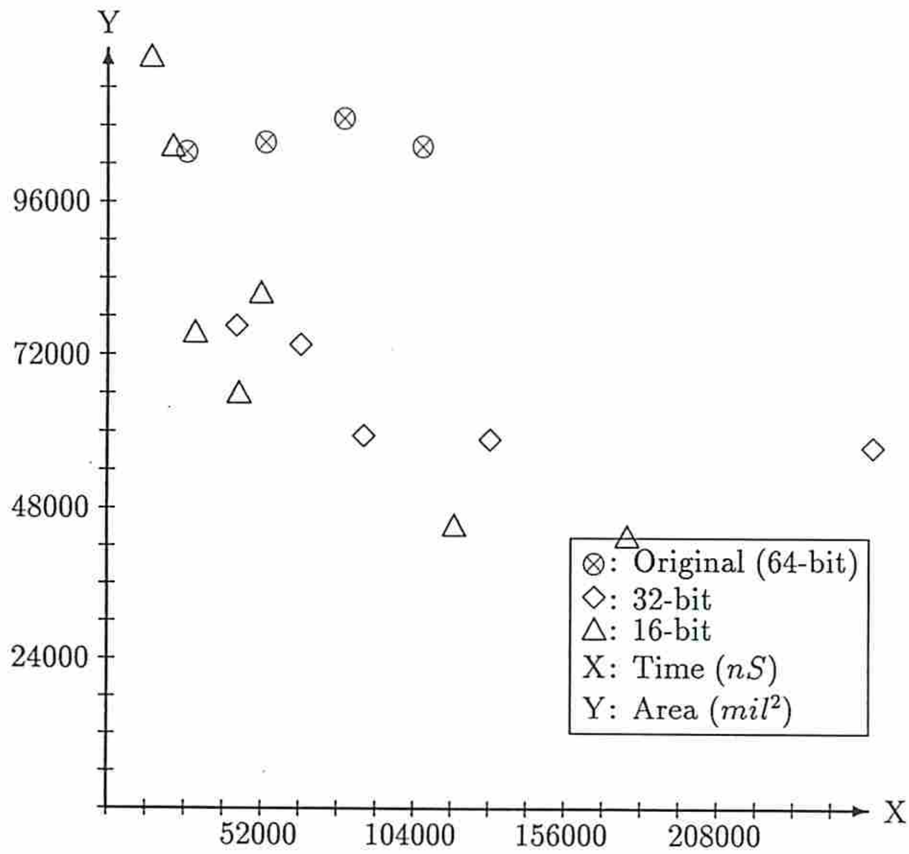


Figure 6.9: Floating Point Coprocessor: Serialize all modules

Table 6.10: Elliptical Filter Module Sets

Design Curve	Modules chosen/quantity			
	Multiply		Add	
	8-bit	16-bit	4-bit	16-bit
Original	–	mulf/8	–	addf/26
8-bit mult.	mul8f/32	–	add4m/40	addf/26
4-bit add.	–	mulm/8	add4s/104	–
Both	mul8f/32	–	add4m/144	–

Table 6.11: Original Elliptical Filter: Predicted

Init. Intrvl	Clock Cycles	Op. Area (mil^2)	Regs	Muxes	Ctrl Area (mil^2)	Total w/wiring Time	Total w/wiring Area
1	14	501200	141	0	–	380	828369
2	14	250600	71	35	412	768	439905
3	15	184800	51	40	674	1164	334878
4	16	127400	41	54	674	1552	246335
5	17	123200	41	54	992	1940	233966
6	18	119000	31	52	1100	2352	227039
7	18	114800	31	50	1246	2744	219318
8	19	65800	31	52	1368	3136	161397
9	20	61600	31	50	1801	3528	145263
13	23	57400	21	48	2508	5148	127869
26	34	53200	21	40	4953	10400	118802

consisting of 26 16-bit *adds* and 8 16-bit *multiplies*. (All the module sets chosen for this example are listed in Table 6.10 and the module set library is given in Table 6.6.)

As discussed earlier, design evaluation of large pipelines is hampered by the limitation of the pipeline synthesis program. **Sehwa** was only able to synthesize the original dataflow graph; synthesis and prediction results are given in Tables 6.11 and 6.12. The average error is 1.6% in area and 0.0% in time; the worst case area error is 3.6%. This small error establishes a high degree of confidence in the estimators. Since no altered dataflow graphs could be synthesized, only prediction tools will be used for further evaluation of pipelined designs.

Table 6.12: Original Elliptical Filter: Actual

Init. Intrvl	Clock Cycles	Op. Area (mil^2)	Regs	Muxes	Ctrl Area (mil^2)	Total w/wiring Time	Total w/wiring Area
1	14	501200	138	0	–	380	827139
2	14	250600	70	35	412	768	438253
3	15	184800	50	47	674	1164	336106
4	17	127400	47	40	674	1552	249047
5	14	123200	31	42	992	1940	228678
6	17	119000	31	46	1100	2352	225599
7	19	114800	29	44	1246	2744	215056
8	21	65800	28	43	1368	3136	157593
9	24	61600	30	40	1801	3528	141883
13	19	57400	19	37	2508	5148	123989
26	32	53200	16	30	4953	10400	112846

The *multiplier* is the largest and slowest component in the dataflow graph. Each 16-bit *multiplier* was translated into a subgraph consisting of 8-bit *multiplies* and 4-bit *adds*. The module set chosen by **SLIMOS** included the fast 8-bit *multiply*, 16-bit *adder*, and medium speed 4-bit *adder*. Since circuit time is still dominated by the 16-bit *add* and 8-bit *multiply*, selecting the fastest 4-bit *adder* could only serve to increase the area. The results for the design curve are plotted in Figures 6.10 and 6.11.

Addition is the most common operation in the graph; serialization of 16-bit *adders* into 4-bit *adders* was performed and included in Figure 6.11. Combining the multiplier and addition bitwidth reductions results in a third design curve which is also shown.

An analysis of the results reveal several things. In Figure 6.10 when only data path hardware is examined, it appears that serializing the *adder* is of dubious quality whereas serializing the *multiplier* is better than the original design. Serializing both is the best; however, this design space does not consider registers, multiplexers, control area, or wiring.

By including routing, storage, and control effects in the area/time computation, the original unaltered dataflow graph remains the best overall as shown in Figure 6.11. Only for the fastest design and cheapest design is a serialized

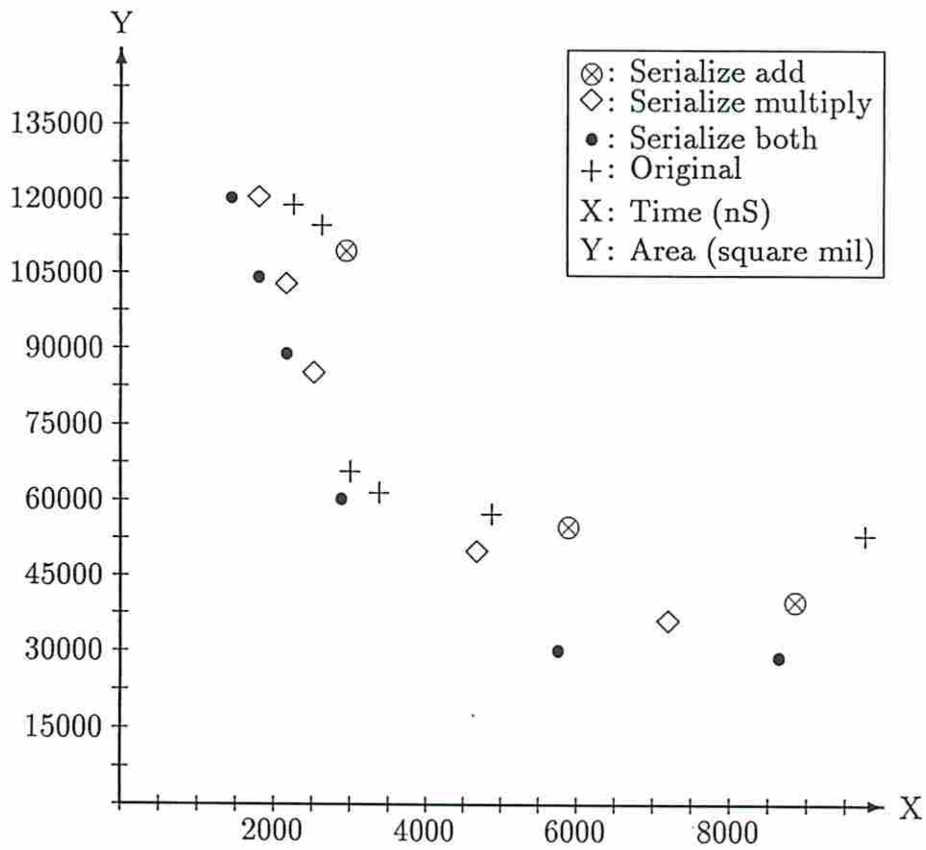


Figure 6.10: Predicted Elliptical Filter: Operators Only

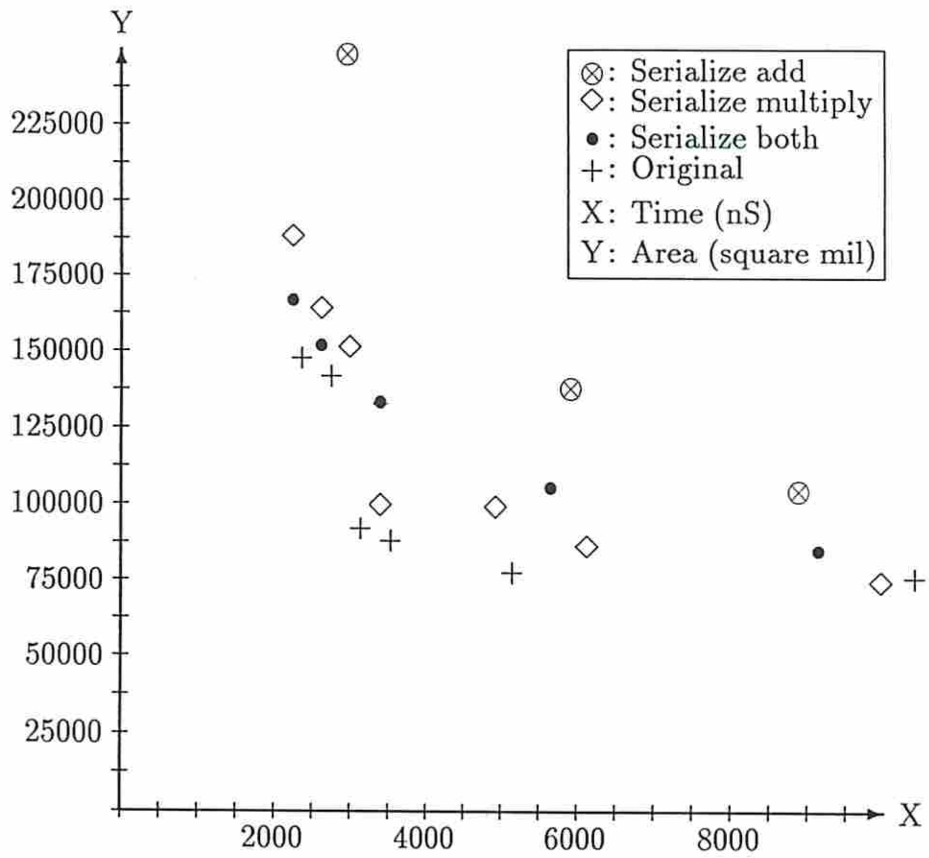


Figure 6.11: Predicted Elliptical Filter: Total Design w/o wiring

multiplier beneficial. Although a serialized multiplier results in two points which extend the design curve, the additional routing, storage, and control erase any benefit for most of the design space. Also, the poor quality of serialized adder designs is clearly evident; excessive delay in the adder as well as the additional registers and multiplexers needed make these designs unsatisfactory.

6.3.2 Composition Tradeoffs

Composition tradeoffs consist of substituting complex modules for specific operations in the dataflow graph to increase module sharing and, in turn, lower circuit area. This tradeoff is at the expense of increased control and signal routing area. Two types of composition tradeoffs are explored: upward equating of bitwidths and ALU substitution.

6.3.2.1 Merging Different Bitwidths: Temperature Controller

The temperature controller shown in Figure 6.12 has multiple bitwidth *adders* and *subtractors* which could be could be merged into single bitwidth *add* and *subtract* operations. There are 8-, 9-bit additions and subtractions, and 10-bit additions being performed. Rather than use three different modules, a 9-bit module could perform both the 8-bit and 9-bit operations which is plotted in the figure.

A 10-bit adder would also suffice for any of the addition operations. This second transformation was combined with transforming the *subtractor* to accommodate a *compare* operation. A third design curve is then realized as shown in Figures 6.13 and 6.14. Due to the size of the dataflow graph, prediction tools were not used.

The tradeoff curve in Figure 6.13 suggests that performing both transformations results in superior designs excepting the fastest. Inclusion of registers, multiplexers, and control shrink the difference as shown in Figure 6.14. A higher area and delay incurred by using larger bitwidth operators and lack of hardware sharing skew the high speed portion of the design space towards the original configuration.

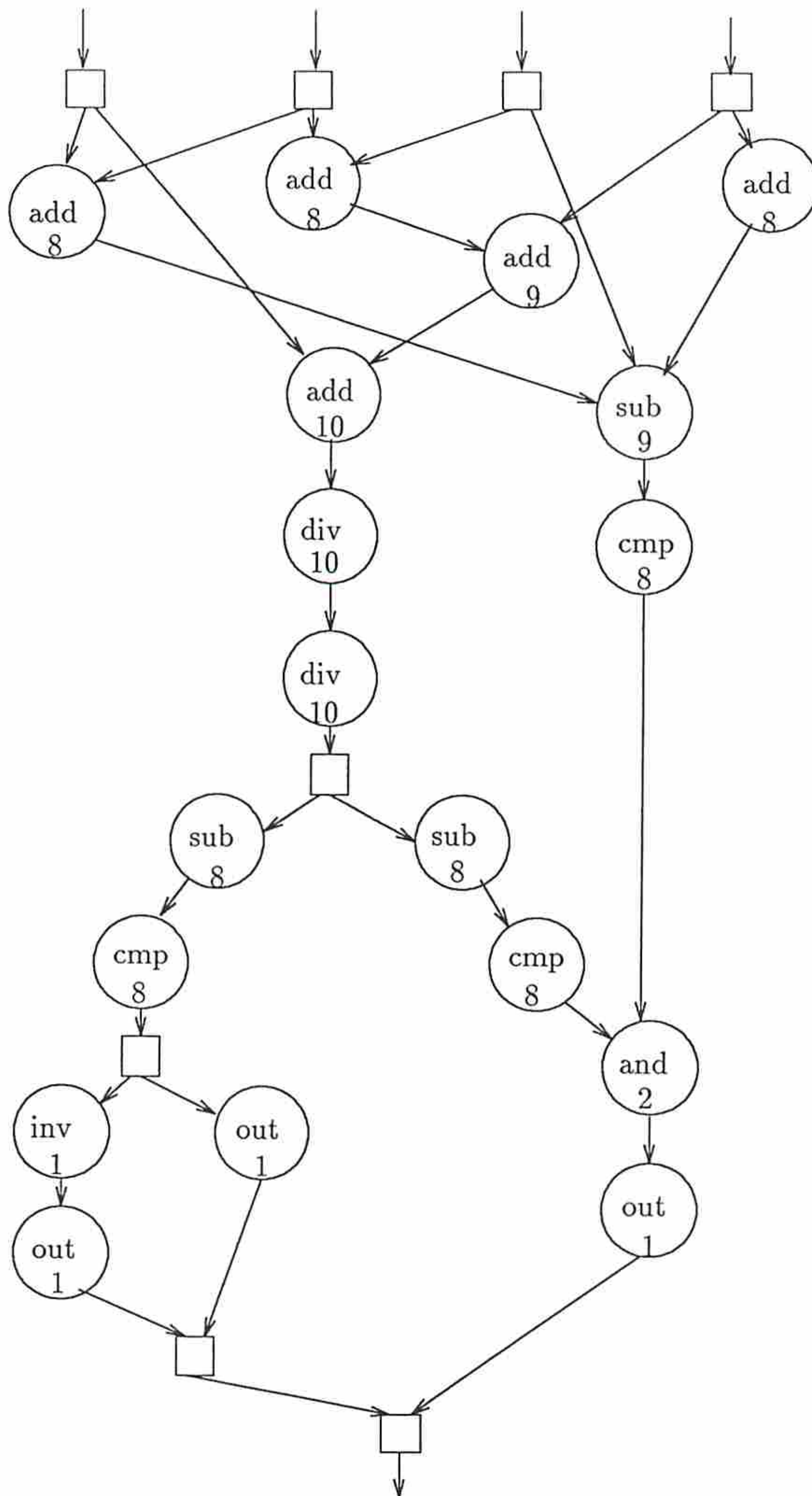


Figure 6.12: Temperature Controller

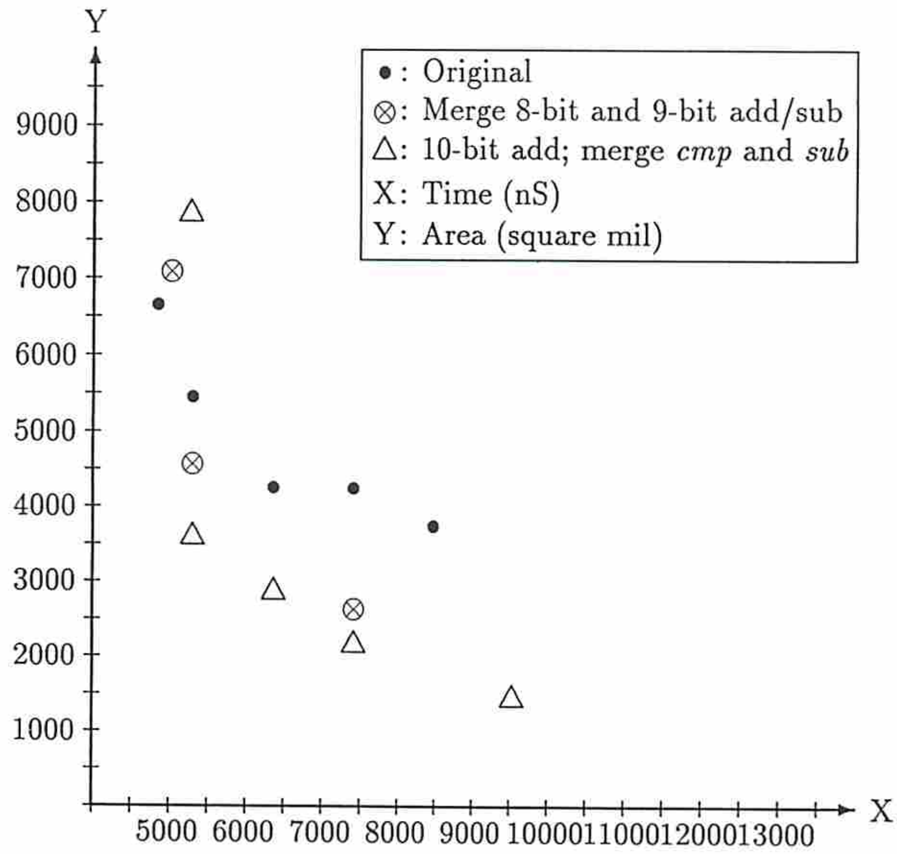


Figure 6.13: Module Bitwidth Tradeoffs: Data Path

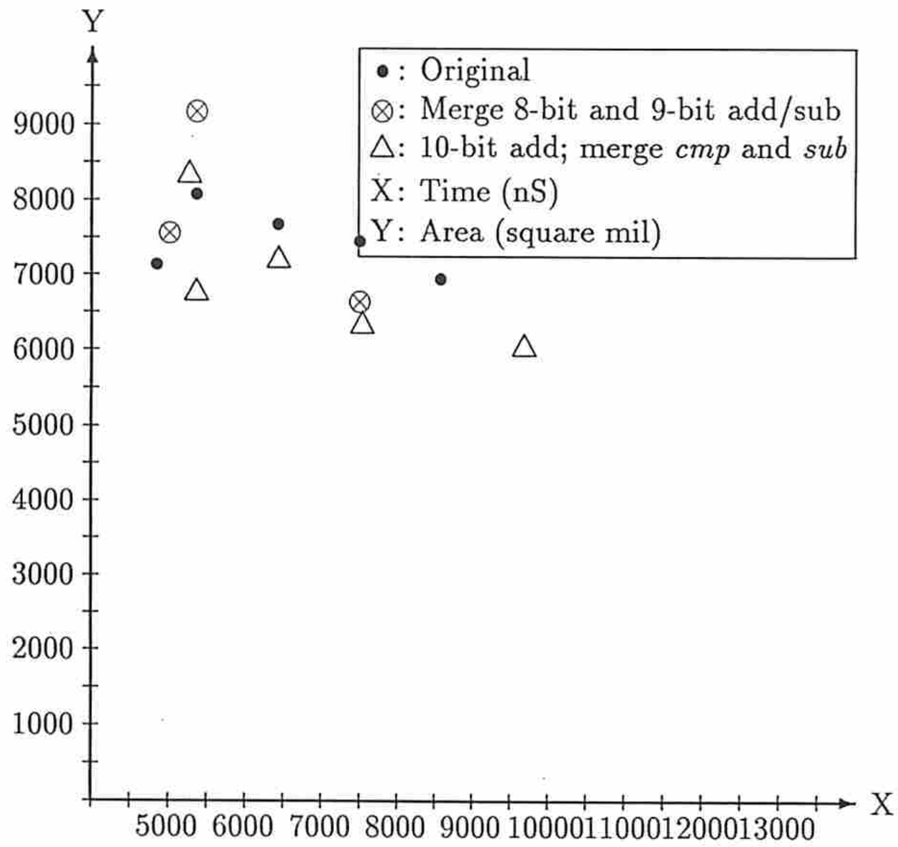


Figure 6.14: Module Bitwidth Tradeoffs: Total

Merging different bitwidths has little impact upon the controller for such simple modules and small dataflow graph; thus, one would expect such a tradeoff to be performed. However, there is additional data path hardware needed to handle the extra bits when executing a smaller bitwidth operation. In an *addition*, a multiplexer is necessary to enter a zero in the next higher bit position as well as select the appropriate carry-out bit. Complex operations may have to multiplex all additional bits and have cumbersome selection logic for the outputs.

6.3.2.2 ALU Substitution: Floating Point Coprocessor

Given an area constraint on the floating point coprocessor, an alternative to bit serialization is the use of ALUs. In the original design, there are a collection of 64-bit and 8-bit closely related operations: *addition*, *subtraction*, *negation*, and *compare*. Combining each bitwidth category into a single ALU increases the area and delay of any one operation, but should result in a cheaper data path for similar serialized scheduling. For each ALU, the controller is burdened with two additional control lines.

To determine the effects of ALU substitution, 64-bit and 8-bit ALUs were separately used in place of their four associated operations, then taken together. Results produced by the estimation tools are presented in Table 6.13 and Figures 6.15 and 6.16. (Note that all extremely slow designs which are impractical were discarded. For example, if the delay increased by two-fold with a 10% decrease in area, this was deemed impractical.) As one would expect, the data path cost drops off more rapidly for the ALU versus the original design curve as revealed in Figure 6.15. This effect is still observed in Figure 6.16 since the control area is small compared to the data path, despite its 25% increase over the original controller. However, it is immediately apparent that 8-bit ALU substitution is not useful in this example as it is inferior to all other designs.

Closer inspection reveals that it is the *mix* of operations which defines the usefulness of ALU substitution. Eleven of the fourteen 64-bit operations are *compare*; the remaining operations listed earlier are each used once. Given the large number of 64-bit *compare* operations, there are likely to be timesteps where one or more are idle. By using ALUs, the total module count is reduced since

Table 6.13: Floating Point Coprocessor with ALU

Dataflow Graph	Area				Total	
	Operator	Register	Multiplexer	Control	Time	Area
ALU (64-bit)	77781	9860	2700	5259	63684	95600
	60721	9860	5850	6061	80229	82492
	51941	9860	8100	6945	160506	76846
ALU (8-bit)	85353	9860	4750	5412	62014	105375
	79653	9860	10925	6197	123988	106635
ALU (both)	77073	9860	3600	5871	63684	96404
	62736	9860	6750	6606	80229	85952
	58161	9860	8100	6569	132188	82690

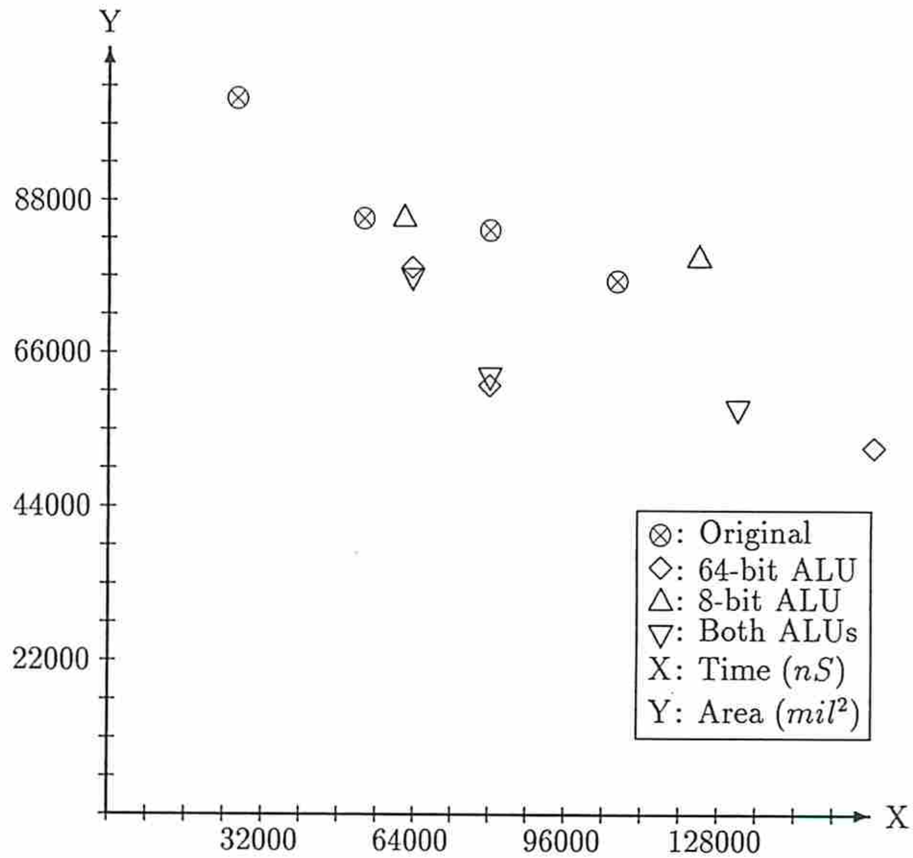


Figure 6.15: Floating Point Coprocessor: Operators

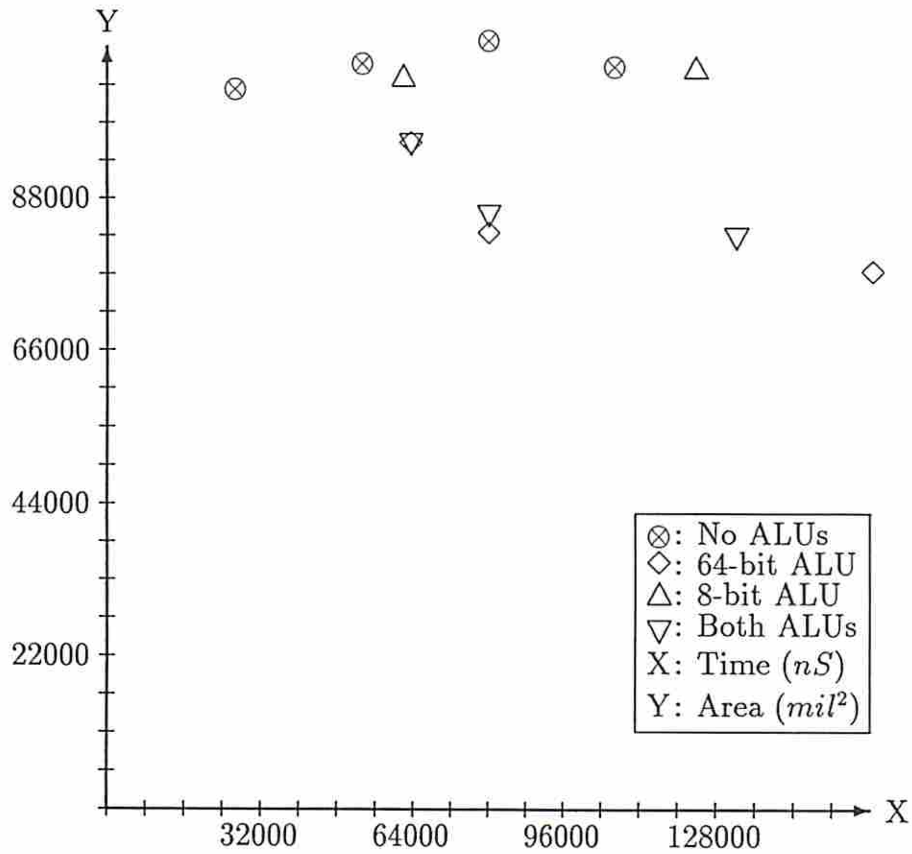


Figure 6.16: Floating Point Coprocessor: Total

the hardware associated with the three separate operations can be reused. In constrast, the sixteen 8-bit operations are evenly spread between *compare*, *addition*, and *subtraction*; ALU substitution does not result in any improvement. Only the increased cost and delay of the ALU is observed.

To confirm this effect, the dataflow graph was altered such that 64-bit operations were evenly spread and *subtract* was set as the dominant 8-bit operation; results are displayed in Figure 6.17. The even spread of the 64-bit operations lessen the improvement of introducing ALUs. However, due to the large area of the 64-bit operations, even a balanced set of operations is benefited by ALU substitution which results in operator sharing. Conversely, concentrating the 8-bit operations into a single operation resulted in improvement when introducing ALUs as compared to the unmodified graph. A tradeoff therefore exists for ALU substitution between the original operation costs and *operation mix* versus the ALU cost.

6.3.3 Multi-processor Tradeoffs

Multi-processor tradeoffs encompass partitioning a large design into smaller individually controlled segments. Although identical operations in different segments may no longer be shared, it is hoped that the number of states associated with controlling the smaller subdesigns would result in a reduced area.

One circuit which could benefit from distributed control transformation is the Intel 8251 Asynchronous Communications circuit. The design is divided into three modules: main (which services the microprocessor connection), receiver, and transmitter (which handle the serial I/O). Under the original design, these modules run independent of one another while sharing some signals and a common data bus. The area and time of each module and the total design (for non-inferior solutions) is detailed in Table 6.14. Although the hardware required for each module is nearly the same, the receiver with its synchronization and error checking requires substantially more control. The objective is to reduce the overall design area while allowing the time to increase.

Although an intuitive design improvement suggests combining the smaller transmitter and main sections while leaving the larger receiver intact, every

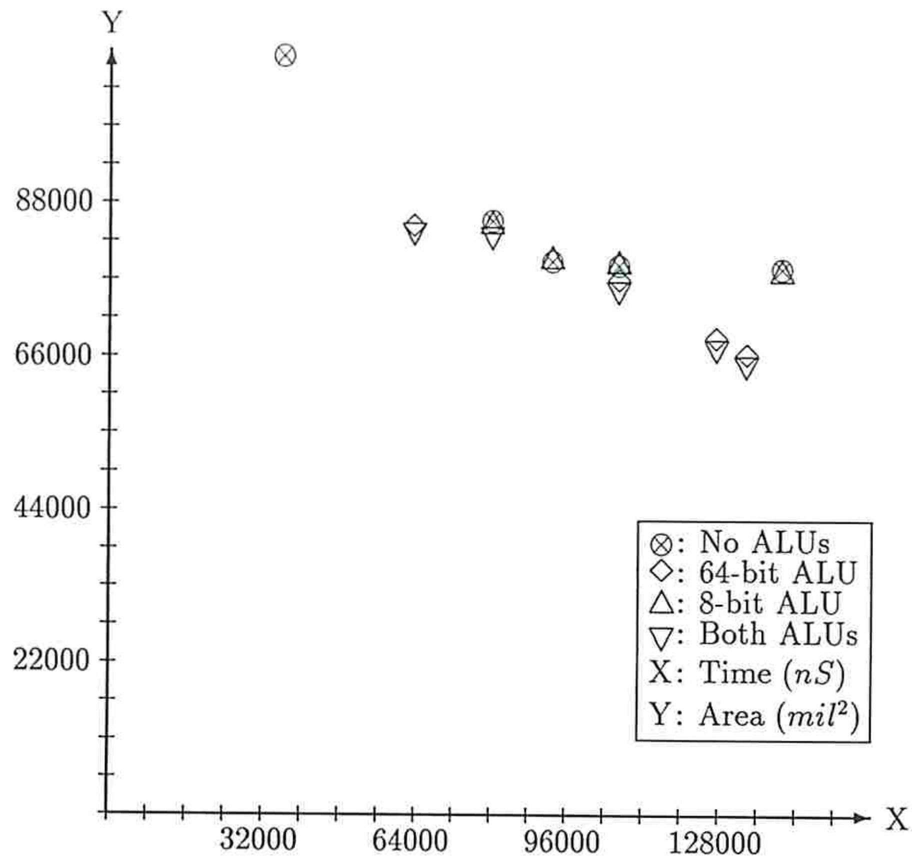


Figure 6.17: Floating Point Coprocessor: Modified (Datapath)

Table 6.14: i8251: Individual designs

Dataflow Graph	Area				Total	
	Operator	Register	Multiplexer	Control	Time	Area
main	1390	640	288	384	718	2702
	1300	640	504	551	1452	2995
rcvr	2390	640	216	7796	718	11042
	2255	640	504	7957	1452	11356
xmtr	1301	896	360	1394	226	3951
	1276	896	576	3333	452	6081
Total	5081	2176	864	9574	718	17695
	4831	2176	1631	11841	1452	20432

Table 6.15: i8251: Parallel Design Combinations

Dataflow Graph	Area				Total	
	Operator	Register	Multiplexer	Control	Time	Area
main-rcvr	3430	896	720	16195	718	21241
	2345	896	936	16889	1077	21066
	2320	896	1224	17062	2178	21502
main-xmtr	2646	1408	792	4720	718	9566
	2531	1408	1152	7693	1077	12784
	2416	1408	1368	7801	2178	12993
rcvr-xmtr	3526	1536	648	4720	718	10430
	2441	1536	1368	7694	1077	13039
	2416	1536	1584	7800	2178	13336
main-rcvr-xmtr	3776	1920	1224	41733	726	48653
	2596	1920	2088	41733	1101	48337
	2441	1920	2520	43755	2202	50636

combination of merging the sections in parallel was attempted. For the parallel designs, the section pairs or trios would share the same controller *and* hardware executed in a parallel fashion. The control area is expected to rise, but the data path cost should drop significantly. Results for the parallel designs are summarized in Table 6.15. Total area and time for completed i8251s is contained in Table 6.16 and Figure 6.18.

For the parallel designs, a large increase in controller area is expected when the receiver is one of the merged sections. The best design curve consists of one controller for the main module and the other for a combined transmitter/receiver. The anomaly for the *rcvr-xmtr* pair is due to the nature of serial I/O control. Whereas the receiver has its control concentrated in the later clock cycles, the transmitter exhibits the opposite characteristics. Hence, their control meshes well and this particular architecture yields a substantial improvement in overall area with little impact on performance.

Since control area dominates the i8251 chip area, combining the individual main, transmitter, and receiver sections in series is another approach for lowering circuit area. A reduction in *both* data path and control path area should be

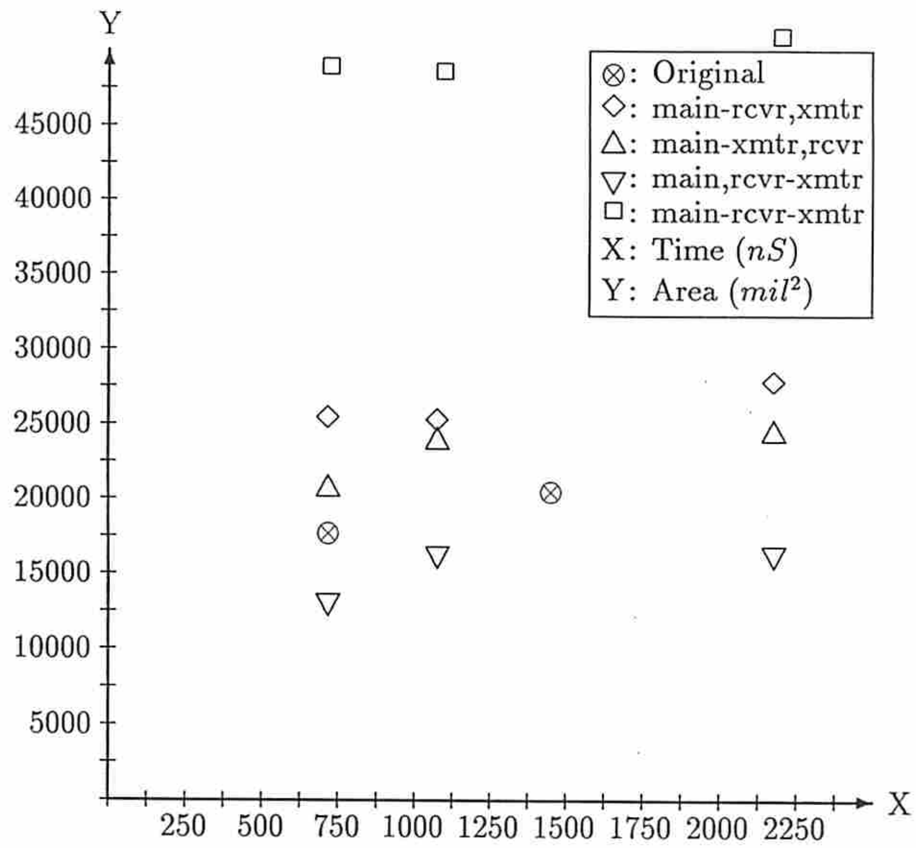


Figure 6.18: i8251: Parallel designs

Table 6.16: i8251: Parallel Design Comparisons

Design Configuration	Time	Area			
		Data Path	Route/Storage	Controller	Total
main/rcvr/xmtr	718	5081	3040	9574	17695
	1452	4831	3807	11841	20432
main-rcvr/xmtr	718	4731	2872	17592	25192
	1077	3646	3088	18283	25017
	2178	3595	3592	20395	27583
main-xmtr/rcvr	718	5036	3056	12516	20608
	1077	4921	3416	15489	23826
	2178	4671	3920	15758	24349
main/rcvr-xmtr	718	4916	3112	5104	13132
	1077	3741	4318	8245	16304
	2178	3716	4264	8351	16331
main-rcvr-xmtr	726	3776	3144	41733	48653
	1101	2596	4008	41733	48337
	2202	2441	4440	43755	50636

observed at the expense of time. Tables 6.17 and 6.18 and Figure 6.19 contain the results of this tradeoff.

In the serial tradeoff, the original conjecture that the receiver is best left by itself holds. Its dominate control area drives this tradeoff to favor combining the smaller main and transmitter controllers together while leaving the receiver intact. Area reduction is observed, but is not as substantial as the parallel tradeoff. It is clear that control area alone cannot be compared when determining the distributed control architecture. Interaction between controllers might result in either a slight increase or large area degradation dependent upon the actual state machine sequencing. Effective distributed control requires a detailed view of the entire control sequencing making this tradeoff difficult to automate or predict.

6.3.4 Tradeoff Trends

During experimentation, ten dataflow graphs were transformed and estimates computed and/or synthesis performed to yield well over 500 designs. Only a

Table 6.17: i8251: Serial Design Combinations

Dataflow Graph	Area				Total	
	Operator	Register	Multiplexer	Control	Time	Area
main-rcvr	3240	640	288	8601	778	12769
	2640	640	504	9150	1089	12934
	2510	640	504	9316	2178	12970
main-xmtr	2932	896	360	1958	758	6156
	2721	896	507	3469	1452	7593
	2606	896	576	3573	2569	7651
rcvr-xmtr	3027	896	360	12910	1110	17193
	2656	896	504	15015	1452	19071
	2606	896	576	15325	2214	19403
main-rcvr-xmtr	3612	896	360	15260	1200	20128
	2746	896	504	15450	1815	19596
	2696	896	576	15260	2611	19428
	2606	896	576	15830	3321	19908

Table 6.18: i8251: Serial Design Comparisons

Design Configuration	Time	Area			
		Data Path	Route/Storage	Controller	Total
main/rcvr/xmtr	718	5081	3040	9574	17695
	1452	4831	3807	11841	20432
main-rcvr/xmtr	778	4541	2184	9995	16720
	1089	3916	2616	12483	19015
	2178	3786	2616	12649	19051
main-xmtr/rcvr	758	5322	2122	9754	17198
	1452	4976	2547	11426	18949
	2569	4861	2616	11530	19007
main/rcvr-xmtr	1110	4417	2184	13294	19895
	1452	4046	2328	15399	21773
	2214	3906	2616	15876	22398
main-rcvr-xmtr	1200	3612	1256	15260	20128
	1815	2746	1400	15450	19596
	2611	2696	1472	15260	19428

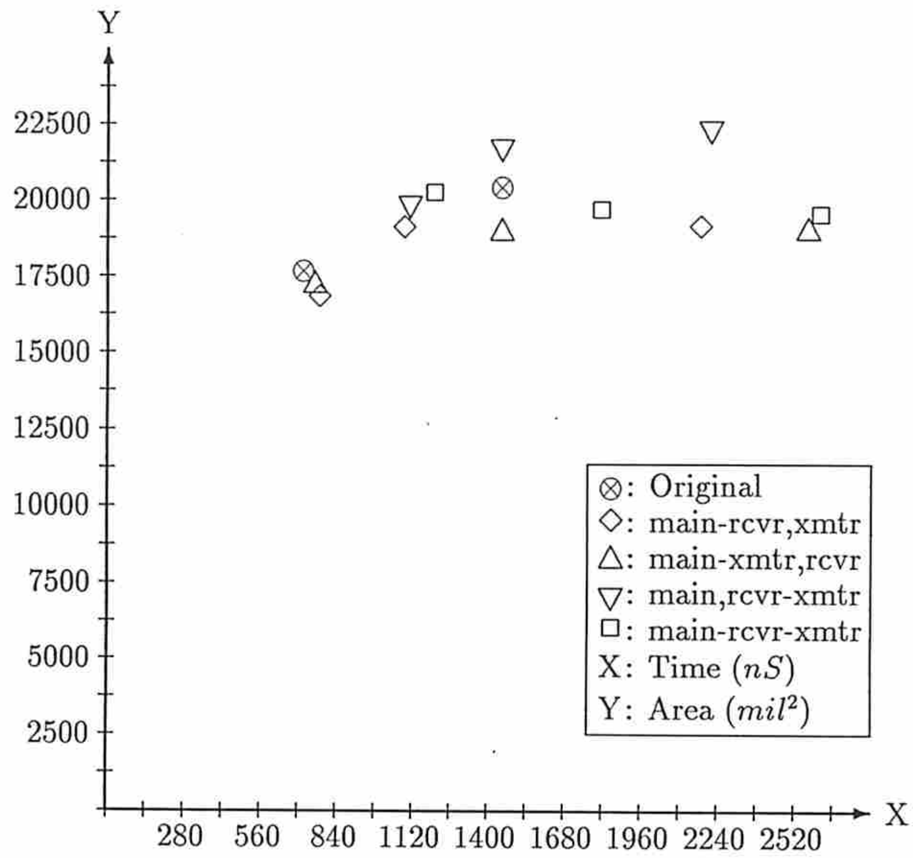


Figure 6.19: i8251: Serial designs

small subset was presented in this chapter. However, with such a large population, a general trend for some of the transforms was observed.

First, it is clear that a distinction must be made between pipelined and non-pipelined designs. Since the method by which design quality is measured differs between these two architectures, tradeoffs which yielded improvements in one architecture were detrimental in the other. Bitwidth tradeoffs underscore this polarization. Although bitwidth trades realized improvements with most non-pipelined designs, especially with slower design goals, many pipelined designs analyzed did not realize any improvements as the pipelined elliptical filter example revealed. In the experimental results obtained, pipelined designs tended to favor larger original bitwidth hardware as compared to non-pipelined designs.

Another trend observed is that bitwidth composition into single modules is useful unless modules are complex or differ by a significant number of bits. As in bitwidth tradeoffs, experimental results revealed that pipelined designs showed much less improvement with bitwidth composition for the faster implementations, but did not degrade the design curve.

Complex operator substitution quality, such as use of an ALU, was almost solely determined by the type mix and quantity of operators for which the ALU would be used. Neither the relative size nor speed of the ALU module appeared to significantly influence this trend. However, an insufficient quantity of operations or a balanced type mix would negate ALU tradeoff benefits. With a large imbalance in operator type mix, even a small quantity of operations would benefit from ALU substitution.

Finally, multiprocessor tradeoffs revealed two effects. First, the relative size of the subcircuits (such as the i8251 main, transmitter and receiver subcircuits) determines the effect of combining these subcircuits in series (where each subcircuit operates in sequence) or parallel (where each subcircuit operates simultaneously). Smaller overall area was observed when subcircuits were combined in series so that they could share data path hardware at the expense of additional control size. Combining these same subcircuits in parallel favored designs which were faster and exhibited smaller control area, but had larger overall area. Hence, whereas the data path dominated serial multiprocessor effects, the control path determined parallel multiprocessor effectiveness.

In applying tradeoffs, it is clear that all aspects of design - data path, control path, routing, and storage - cannot be ignored during evaluation. Although wiring area was not analyzed, designs that are similar in area would need it to distinguish the best design.

Of equal importance is the application of constraints. Area and time constraints not only delineate design quality, but determine the initial model set for both the estimation and synthesis tools as well as which tradeoffs are of use. Large changes in constraint values could result in different design curves and additional regions in the design space to be explored.

6.4 Summary

In this chapter, a methodology for control path/data path tradeoff analysis was explored. A description of a system which can be used to perform tradeoffs using either estimation or synthesis tools was detailed. Finally, tradeoffs were applied to illustrate bit serialization, bitwidth analysis, operator composition, and distributed control.

It is evident that applying numerous tradeoffs entails enormous computation time if only synthesis tools are employed. Despite some inaccuracies, estimation techniques are extremely useful for establishing trends and locating regions which are capable of meeting the design constraints. The execution time of prediction tools also allow numerous detailed tradeoffs to be performed within a short period. This disparity between synthesis and estimation runtime is acute when larger practical designs are attempted.

Finally, it is clear from the examples that tradeoffs can yield unexpected results. Some general trends can be noted, but there seem to be no absolutes. However, these exceptions can be recorded and exploited on subsequent analyses. With a usable framework established for actual performing these tradeoffs, it is now possible to write higher-level tools which could automate the concepts to use and build upon the tradeoff matrix. Further efforts to identify other tradeoffs which designers employ would also be of use.

Chapter 7

Conclusions and Future Research

7.1 Contributions

In this thesis a system was described which allows a user to quickly locate superior designs in a large discrete digital design space. Through the use of system-level tradeoffs, regions which might be ignored by present day design tools can be constructed and analyzed. Through the use of estimation techniques, evaluating the design space after applying a tradeoff allows a designer to ascertain the resultant circuit quality quickly. This approach is of particular importance in today's design community where ever larger chips are being constructed while development time decreases. The concept will also be used by more intelligent systems which transform and locate superior designs automatically.

In order to perform system-level tradeoffs, the mechanism for predicting the circuit parameters must be in place. (In the digital design community, area and time are the most widely used criteria for determining design quality.) Also, only a minimal amount of information should be required from the designer. One begins with a behavioral description in the form of a dataflow graph, module library, and a set of user-defined global constraints and finishes with a candidate set of one or more designs which meet the requirements.

Central to this methodology are a complete suite of synthesis tools and set of accurate prediction tools; these latter utilities rapidly provide a set of estimated designs to establish the design region of interest. Then, slower synthesis tools can be targeted to locate the precise designs within this area. In order for this concept to be viable, both the synthesis and prediction methods must have

comparable analysis capability. Prior to this thesis, there were a number of missing tools; thus, a large portion of the research was involved in closing this gap.

In Chapter 2, extensions to a non-pipelined data path synthesis program **MAHA** were presented and the tool validated. Useful additions to **MAHA** were resource sensitive scheduling, full serialization of the data path, register assignment, and multiplexer assignment. The latter two features are useful if one does not wish to use other utilities for register and multiplexer assignment.

In Chapter 3, an estimation model for PLA control area was derived. Without a control model, one is unable to perform control path/data path tradeoffs using prediction tools; this is the first known research to address this topic. Primary features of the PLA area estimation model are

- applicability to both pipelined and non-pipelined designs,
- alternate PLA control structures depending upon the quantity of data path registers and number of control states,
- handling of conditional branches and loops, and
- applicability to both unfolded and folded PLAs.

An area estimation model for registers and multiplexers comprised Chapter 4. Register estimation employed an empirical approach by bounding the limits and observing the characteristics of real designs. Multiplexer area estimation used a statistical approach which assesses the degree of duplicity of each operator and register input. Parameters available to both of these models are restricted to the information that would be available during data path estimation and does not rely upon any synthesis results. As in the control area estimator, both pipelined and non-pipelined design styles are accommodated.

With a suite of powerful tools to perform design synthesis and prediction, application of tradeoffs was discussed in Chapter 5. The types of tradeoffs were identified.

Finally, in Chapter 6, a methodology for determining the effects of system-level transformations presented. A number of examples were analyzed with different control path/data path tradeoff types applied to real designs. Using these

tools, assessing the impact of performing these tradeoffs was demonstrated including

- operator decomposition into simpler or smaller parts,
- operator composition into complex modules such as ALUs,
- varying the bitwidth of operators versus improved sharing, and
- multi-processor approaches to achieve better throughput.

7.2 Automating System-Level Tradeoffs

Clearly, the effort expended on this research can be used as a basic foundation for future research on system tradeoffs. The primary bottleneck of not having an evaluation methodology and a suite of tools - both estimation and synthesis engines - has been resolved. Additional topics related to system-level tradeoffs would address the non-automated parts of the system.

7.2.1 Detection of Tradeoff Locations

A major problem in system-level tradeoffs is detecting *where* such tradeoffs are applicable. For a tradeoff involving a single data path operation, this is not difficult. However, for a tradeoff which manipulates a group of operations, matching every subgroup of operations in a large dataflow graph against the target *template* becomes an intractable problem. Essentially, the template graph is being compared against all subgraphs of the dataflow graph to find a one-to-one correspondence for both vertices and edges. Henceforth, the problem will be called *graph matching* although some mathematicians prefer describing a matching template as being *embeddable* in the dataflow graph.

Graph matching is currently used to some extent in synthesis and compiler design. Researchers in hardware synthesis use matching to determine allocation and sharing of hardware [Bra75]. Compiler developers use graph matching for

common subexpression elimination and code generation [AU77]. Graph matching in synthesis is applied over the entire graph, but is confined to single operator matching; matching in compiler design may involve several operations, but is generally restricted to small local subgraphs or expressions.

The difficult graph matching problem to resolve for system-level tradeoffs is a global search of a graph to find a particular subgraph which is isomorphic to some particular template. Given a specific subgraph, κ, to be matched against a dataflow graph, Ω, the complexity of the problem can be written as

$$C \propto |\Omega|^{|\kappa|} \tag{7.2.1}$$

where $|\kappa| << |\Omega|$ and $|S|$ means the cardinality (number of vertices or operations) in set S.

Equation 7.2.1 reveals that increasing the template size results in an exponential increase in search time. Therefore, keeping the *number* as well as the *size* of the templates small is important; this is at odds with initial analysis showing that, in general, functions having numerous permutations are common. For example, a single multiplier operation is an apparent candidate for expansion into a variety of multiplier/add or shift/add components.

At first glance, the problem of both *locating* a particular subgraph as well as *detecting* its myriad of permutations appears unsolvable. Reducing the scope to only subgraph matching lowers the complexity. Since system-level tradeoffs operate in the behavioral space, any permutations such as the multiplier example above will be detected: the behavioral description will remain unchanged, only the structural implementation is different. Thus, the series-of-adders is known to be a constant multiplier from the original specification and also demonstrates one advantage of working down from a higher-level description.

Another aspect of template matching is detection of common subgraphs. In this case, a subgraph of some arbitrary shape and function is duplicated within the dataflow graph. Detection of these graphs is considerably more difficult than common subexpression detection in compiler design. Subexpression analysis considers a tree and constructs the groups always starting with the leaf nodes; common subgraphs share the same leaf nodes (values) and vertices (operations).

For a dataflow graph, there is no distinct starting point for building a subgraph. Furthermore, unlike compilers, subgraphs are considered similar if they perform the same set of operations in the same order. Each may actually operate on a different set of values. (In compilers, both the operation and the value must match.) However, the commonality among values can be used to determine the *binding attraction* of two common subgraphs. There are two methods known for accomplishing this task in an automated fashion. *Template reduction* based upon compiler design is one approach and *dataflow graph reduction* based upon graph theory and dataflow machine analysis is another candidate.

7.2.2 Ordering System-Level Tradeoffs

With a set of procedures for locating tradeoff templates or detecting multiple occurrences of a given subgraph, it is important to determine *when* such tradeoffs should be performed. Clearly, blind application of the most useful localized template may not result in any global improvement in some application. Therefore, a *strategy* for applying transforms (templates) to the graph which result in a better solution - based upon user constraints - is an important aspect of design transformation. Given a complete set of allowable transformations and an efficient graph matching algorithm, is there an optimal *order* AND *location* to apply the transforms? *Ordering* is necessary since there is often more than one transform that can be applied; a *location* schema arbitrates in the (likely) event that the best transform can be applied at more than one graph location. To illustrate the problem, the AR filter shown earlier is used as an example.

If a template comprised of two **multipliers** connected to an **adder** exists and can be used as a single complex operation to replace the subgraph, there are a total of 256 transformed graphs. Of course, each implementation can be partitioned into a number of timesteps to produce a variety of designs. Due to the discrete nature of the design space, it is unrealistic to assume that a single analysis will result in the optimal design. If an unknown number of steps are necessary to achieve an optimal design, it is desirable to be able to halt the search and still note an improvement towards the optimal design.

7.2.3 Automated Application of Tradeoffs

Once methods for determining where and when to apply tradeoffs have been determined, a general system-level analysis package can be produced in an automated fashion. Since this approach is intended to automate an important step in the design process, it should entail minimal user input. However, to allow for the expertise of individual designers, the design space can be bounded by the user and rules and/or templates eliminated for consideration by the user before analysis begins.

Clearly, it would be difficult to capture all possible transformation templates in a single utility. Rather, a useful subset would be included to account for the most significant design changes. Ordering of the tradeoffs would result in the most substantial improvements being performed early in the tradeoff analysis with successively smaller improvements at each step. A constraint of minimum design improvement should be introduced such that the process has a finite runtime.

7.3 Future Research

As in all research, a solution to one problem only results in the uncovering of numerous others. In fact, some problems that are discovered actually change the direction of the thesis substantially. Such was the case in this research.

Originally, the research topic involved determining the types of system-level tradeoffs and show how high-level partitioning affects the ultimate design. These tradeoffs would be performed in an automated manner to transform a design into the best structure for the given constraints and goals. Although many types of tradeoffs were addressed by this thesis, there may be others yet unexplored. Due to the effort required in generating a system that could perform system-level tradeoffs, the initial goal of automating tradeoffs was never realized. A major portion of the research focused upon area estimation models for control, routing, and storage hardware. It was only after these tools were complete could performing system-level tradeoffs be realized.

Much of the potential research reflects the importance of accurate estimation models. In particular, both the control and routing models can be extended. The PLA model should be reformulated into a more general model which accommodates multi-level logic and microcontrollers as well as PLAs. Although a multi-processor tradeoff was shown in the previous chapter, a model which describes the effects of distributed and/or hierarchical controllers would enhance the work. Nearly all large designs in production today, especially microprocessors, use a complex hierarchy of controllers to increase parallel operation or to minimize area.

Another model lacking entirely is area estimation for bus access logic. Although this would seem to be a simple revision of the multiplexer model, there are more issues in buses which affect their size such as the scheduling and value usage. Even more difficult is a method for estimating the steering logic area where both buses and multiplexers are used.

The importance of estimation models cannot be understated. Although synthesis techniques will continue to improve, there is unlikely to be a major breakthrough in one aspect of real design: computation time. Even with the advances in computer throughput, the complexity of chips being produced has also risen; absolute computation time has virtually remained constant. For large designs, solely using synthesis tools would be too costly in computer runtime and user patience to be of any use. However, more robust and precise models allow the user to explore extremely large design spaces with numerous tradeoffs in a reasonable time. With these additional tools, a system capable of producing high quality designs automatically from a general set of user constraints and goals becomes practical.

Reference List

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Massachusetts, 1974.

[AU77] A. Aho and J. Ullman. *Principles of Compiler Design.* Addison-Wesley, Massachusetts, 1977.

[AUS79] A. Aho, J. Ullman, and V. Sethi. *Compiler Design and Techniques.* Addison-Wesley, Massachusetts, 1979.

[Bal84] D. Baldwin. *Automatic Evaluation of Design Choices in Digital Controller Synthesis.* PhD thesis, Yale University, December 1984.

[Bar73] M. Barbacci. ISP: A Notation to Describe a Computer's Instruction Sets. *COMPUTER*, March 1973.

[BB87] N. Biswas and C. Bhat. A Maximum PLA Folding Algorithm. In *Proc. of ICCD*, pages 686–689. IEEE, October 1987.

[BD71] H. Barsamian and A. DeCegama. Evaluation of Hardware-Firmware-Software Trade-offs with Mathematical Modeling. In *Spring Joint Computer Conference*, pages 151–161, 1971.

[Bra75] F. Bradshaw. Directed Graph Models for Hardware/Software Design. In *1975 Intl. Symp. on Computer Hardware Description Languages and Their Applications Proc.*, pages 7–15. ACM SIGDA, ACM SIGARCH, IEEE Comp. Soc. Tech. Com. on Computer Architecture, September 1975.

[Cam85] R. Camposano. Synthesis Techniques for Digital System Designs. In *Proc. 22nd Design Automation Conference*, pages 475–481, June 1985.

[Cas85] A. E. Casavant. *Algorithms for Logic Design Automation.* PhD thesis, University of Illinois at Urbana-Champaign, 1985. Dept. of Computer Science.

[DSVV83] G. DeMicheli, A. Sangiovanni-Vicentelli, and T. Villa. Computer-aided Synthesis of PLA Based Finite State Machines. In *Proc. Intl. Conf. on Computer Aided Design*, pages 154–156, November 1983.

[Eve79] S. Even. *Graph Algorithms.* Computer software engineering series. Computer Science Press, Rockville, MD, 1979.

[FSC84] G. Frank, C. Smith, and J. Cuadrado. An Architecture Design and Assessment System for Software/Hardware Codesign. White Paper, Research Triangle Park, November 1984.

[GE88] C. Gebotys and M. Elmasry. VLSI Design Synthesis with Testability. In *Proc. 25th Design Automation Conference*, pages 16–21, June 1988.

[Gir87] E. Girczyc. Loop Winding - A data flow approach to functional programming. In *IEEE Intl. Symp. on Circuits and Systems*, pages 382–385, May 1987.

[GK84] E. Girczyc and J. Knight. An ADA to Standard Cell Hardware Compiler Based on Graph Grammars and Scheduling. In *Proc., 1984 Intl. Conference on Computer Design - ICCD*, pages 726–729, October 1984.

[GP82] J.J. Granacki and A.C. Parker. The Effect of Register-Transfer Design Tradeoffs on Chip Area and Performances. In *Proc. 20th Design Automation Conference*, December 1982.

[GR83] R. Galivanche and S. Reddy. A Parallel PLA Minimization Program. In *Proc. Intl. Conference on Computer Aided Design*, pages 600–607, November 1983.

[GR87] R. Galivanche and S. Reddy. A Parallel PLA Minimization Program. In *Proc. 24th Design Automation Conference*, pages 600–607. IEEE and ACM, June 1987.

[H+82] J. Hennessy et al. Hardware/Software Tradeoffs for Increased Performance. *Computer Architecture News*, 10(2):2–11, March 1982.

[Ham83] G. Hamachi. *Designing Finite State Machines with PEG.* UC Berkeley, 1983.

[HC88] R. Hartley and P. Corbett. A Digit-Serial Silicon Compiler. In *Proc. 25th Design Automation Conference*, pages 646–649. IEEE/ACM, June 1988.

[HE89] B. Haroun and M. Elmasry. Architectural Synthesis for DSP Silicon Compilers. *IEEE Trans. on CAD*, 8(4):431–447, April 1989.

[HP81] L. Hafer and A. Parker. A Formal Method for the Specification, Analysis and Design of Register-Transfer Level Digital Logic. In *Proc. 18th Design Automation Conference*, pages 846–853, June 1981.

[HP83] L. Hafer and A. Parker. A Formal Method for the Specification Analysis, and Design of Register-Transfer Level Digital Logic. *IEEE Trans. on Computer-Aided Design*, CAD-2(1):4–17, January 1983.

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

[JMP88] R. Jain, M. Mlinar, and A. Parker. Area-Time Model for Synthesis of Non-Pipelined Designs. In *Proc. Intl. Conference on Computer-Aided Design*, pages 48–51, November 1988.

[JPP87] R. Jain, A. Parker, and N. Park. Predicting Area-Time Tradeoffs for Pipelined Design. In *Proc. 24th Design Automation Conference*, pages 35–41, July 1987.

[JPP88] R. Jain, A. Parker, and N. Park. Module Selection for Pipelined Designs. In *Proc. 25th Design Automation Conference*, pages 542–547, June 1988.

[KC86] Y. Kuo and W. Chou. Generating Essential Primes for a Boolean Function with Multiple-Valued Inputs. In *Proc. 23rd Design Automation Conference*, pages 193–199, June 1986.

[KP84] F. Kurdahi and A. Parker. Area estimation of standard cell designs. DISC Report 84-2, EE-Systems Dept. USC, 1984.

[KP86] F. J. Kurdahi and A. C. Parker. PLEST: A Program for Area Estimation of VLSI Integrated Circuits. In *Proc. 23rd Design Automation Conference*, pages 467–473, June 1986.

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

[KP90] K. Kucukcakar and A. C. Parker. MABAL - A Software Package for Module And Bus ALlocation. *To appear in Intl. Journal of Computer-Aided VLSI Design*, 2(4), 1990.

[KT83] T.J. Kowalski and D.E. Thomas. The VLSI Design Automation Assistant: Prototype System. In *Proc. 20th Design Automation Conference*, pages 479–483, 1983.

[Kur87] F. J. Kurdahi. *Area Estimation of VLSI Circuits.* PhD thesis, University of Southern California, August 1987.

[KY85] Y. Koseki and T. Yamada. PLAYER: A PLA Design System for VLSIs. In *Proc. 22nd Design Automation Conference*, pages 766–769, June 1985.

[LA83] Wentai Liu and Daniel Atkins. Bounds on the Saved Area Ratio due to PLA Folding. In *Proc. 20th Design Automation Conference*, pages 538–544. IEEE and ACM, June 1983.

[Lei81] G. W. Leive. *The Design, Implementation, and Analysis of an Automated Logic Synthesis and Module Selection System.* PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, January 1981.

[Liu68] C. L. Liu. *Introduction to Combinatorial Mathematics.* McGraw-Hill Book Company, New York, 1968.

[Ltd85] INMOS Ltd. *OCCAM Programming Manual*, July 1985.

[LVSV82] M. Luby, U. Vazirani, and A. Sangiovanni-Vincentelli. Some Theoretical Results on the Optimal PLA Folding Problem. In *Intl. Conference on Circuits and Computers*, pages 165–170. IEEE, 1982.

[Man72] R. L. Mandell. Hardware/software trade-offs - Reasons and directions. In *Fall Joint Computer Conference*, pages 453–459, 1972.

[Mar86] R. M. Marshall. *Synthesis of Hardware Systems from Very High Level Behavioural Specifications.* PhD thesis, University of Edinburgh, 1986.

[McF78] M. McFarland. The value trace: A data base for automated digital design. Master's thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., December 1978.

[McF81] M. McFarland. Allocating Registers, Processors, and Connections. Internal Paper, 1981.

[McF86] M. McFarland. Using Bottom-Up Design Techniques in the Synthesis of Digital Hardware from Abstract Behavioral Descriptions. In *Proc. 23rd Design Automation Conference*, pages 474–480, June 1986.

[McF87] M. McFarland. Reevaluating the Design Space for Register-Transfer Hardware Synthesis. In *Intl. Conference on Computer-Aided Design*, pages 262–265, November 1987.

[MPC88] M. C. McFarland, A. C. Parker, and R. Camposano. Tutorial on High-Level Synthesis. In *Proc. 25th Design Automation Conference*, June 1988.

[MT86] Darrell Makarenko and John Tartar. A Statistical Analysis of PLA Folding. *IEEE Trans. on Computer-Aided Design*, CAD-5(1):39–51, January 1986.

[Nag80] A. Nagle. *Automatic Design of Sequencers for The Control of Digital Hardware.* PhD thesis, Carnegie-Mellon University, October 1980.

[NCP82] A. Nagle, R. Cloutier, and A. Parker. Synthesis of Hardware for the Control of Digital Systems. *IEEE Trans. on Computer-Aided Design*, CAD-1(4):201–212, 1982.

[NP81] A. Nagle and A. Parker. Algorithms for Multiple-Criterion Design of Microprogrammed Control Hardware. In *Proc. 18th Design Automation Conference*, pages 486–493, June 1981.

[NT86] J. Nestor and D. Thomas. Behavioral Synthesis with Interfaces. In *Intl. Conference on Computer Aided Design*, November 1986.

[OHM+84] J. Ousterhout, G. Hamachi, R. Mayo, W. Scott, and G. Taylor. MAGIC: A VLSI Layout System. In *Proc. 21st Design Automation Conference*, pages 152–159, June 1984.

[Pan88] B. Pangrle. SPLICER: A Heuristic Approach to Connectivity Binding. In *Proc. 25th Design Automation Conference*, pages 536–541, June 1988.

[Par85] N. Park. *Synthesis of High Speed Digital Systems.* PhD thesis, University of Southern California, August 1985.

[Pen86] Z. Peng. Synthesis of VLSI Systems with the CAMAD Design Aid. In *Proc. 23rd Design Automation Conference*, pages 278–283, June 1986.

[PG87] B. Pangrle and D. Gajski. Design Tools for Intelligent Silicon Compilation. *IEEE Trans. on Computer-Aided Design*, CAD-6(6):1098–1112, November 1987.

[PH87] A. Parker and S. Hayati. Automating the VLSI Design Process using Expert Systems and Silicon Compilation. *Proc. IEEE*, 75(6):777–785, 1987.

[PK87] P. Paulin and J. Knight. Force-Directed Scheduling in Automatic Data Path Synthesis. In *Proc. 24th Design Automation Conference*, pages 195–202, July 1987.

[PKM84] A. Parker, F. Kurdahi, and M. Mlinar. A General Methodology for Synthesis and Verification of Register Transfer designs. In *Proc. 21st Design Automation Conference*, June 1984.

[PP85a] N. Park and A. Parker. Synthesis of Optimal Clocking Schemes. In *Proc. 22nd Design Automation Conference*. ACM IEEE, June 1985.

[PP85b] N. Park and A. Parker. Synthesis of optimal pipeline clocking schemes. Technical Report DISC/85-1, Dept. of EE-Systems, University of Southern California, January 1985.

[PP86] N. Park and A.C. Parker. Sehwa:A Program for Synthesis of Pipelines. In *Proc. 23rd Design Automation Conference*, pages 454–460, July 1986.

[PP88] N. Park and A. Parker. Sehwa: A Software Package for Synthesis of Pipelines from Behavioral Specifications. *IEEE Trans. on Computer Aided Design*, 7(3), March 1988.

[PPM86] A. Parker, J. Pizarro, and M. Mlinar. MAHA: A Program for Datapath Synthesis. In *Proc. 23rd Design Automation Conference*, pages 461–466, July 1986.

[RSV86] R. Rudell and A. Sangiovanni-Vincentelli. Exact Minimization of Multiple-Valued Functions for PLA Optimization. In *Proc. Intl. Conference on Computer Aided Design*, pages 352–355, November 1986.

[Sno78] E. Snow. *Automation of Module Set Independent Register Transfer Level Design.* PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April 1978.

[Tho86] D. E. Thomas. *Design Methodologies*, pages 401–439. Elsevier Science Publishers B. V., North-Holland, 1986.

[Tri85] H. Trickey. *Compiling Pascal Programs into Silicon.* PhD thesis, Dept. of Computer Science, Stanford University, July 1985.

[WC88] C.-L. Wey and T.-Y. Chang. PLAYGROUND: Minimization of PLAs with Mixed Ground-True Outputs. In *Proc. 25th Design Automation Conference*, pages 421–426, June 1988.

Appendix A

Notation

Some notation is common throughout the thesis. Their meanings are listed here.

- E is the number of edges in a dataflow graph.
- N is the number of nodes in a dataflow graph.
- ζ is the number of control states.
- P is the number of stages into which a dataflow graph is partitioned.
- M is the number of *unique* multiplexer control lines.
- S is the quantity of input status lines.
- A is the number of additional control lines (ALUs, etc.).
- R_c is the number of control lines associated with registers.
- R is the number of *unique* registers being controlled in the data path.
- i is the number of inputs on the PLA.
- o is the number of outputs on the PLA.
- p is the number of product-terms on the PLA.
- $c_1 \ldots c_4$ are coefficients for determining the area of a PLA; they are technology dependent parameters.
- $PL(\alpha_i)$ is the length of a loop α_i in stages where $PL(\alpha_i) > 0$.

- $N(\alpha_i)$ is the number of iterations of loop α_i where $N(\alpha_i) > 1$.
- K_{reg} is a value from 0 to 1 which reflects the degree of register uniqueness per loop. If a value stored at a given step is put into a different register each loop, $K_{reg} = 1$. If the same register is used each loop, then $K_{reg} = 0$.
- $PC(\beta_i)$ is the length of conditional path β_i in stages where $PC(\beta_i) > 0$ (at least one partition is reached or crossed).
- C is the number of conditional paths in the entire graph.
- $\mathcal{P}$ is an edge partition cutset of the dataflow graph. Its members are the edges which divide the dataflow graph into exactly two subgraphs where *root* and *outport* are not members of the same subgraph.
- $\mathcal{C}$ is the set of all partition cutsets, $\mathcal{P}$.
- c is the clock cycle time.
- p is the number of stages (of clock cycle c) into which the dataflow graph has been partitioned.
- $\mathcal{V}(e_i)$ is the value assigned to edge e_i.
- L is the height of the dataflow graph. This height or maximum length (in operations) from *root* to *outport* is exclusive of the end nodes.
- n_i is the number of operations of type i in the dataflow graph.
- o_i is the number of operators of type i ubbsed in the actual design.
- l is the initiation interval of a given pipelined design. Initiation interval is defined as the number of clock cycles between two successive inputs.
- $\bar{E}_{in}$ is the set of edges from *root* to unique destination nodes.
- $\bar{E}_{out}$ is the set of edges to *outport* from unique source nodes.
- R is the number of registers needed for a given design.

- R_{min} and R_{max} are the minimum and maximum number of registers required for a given dataflow graph (any given cutset of $\mathcal{C}$).

- Rnp_{min} and Rnp_{max} are the minimum and maximum number of registers required for any non-pipelined design of a given dataflow algorithm, respectively.

- Rp_{min} and Rp_{max} are the minimum and maximum number of registers required for a given pipelined design, respectively.

- Rnp_{est} and Rp_{est} are the estimates on the number of registers for a given non-pipelined and pipelined design, respectively.

- Rnp_{lim} is the empirical limit on the maximum number of registers required for a non-pipelined design.

- u is the number of microcycles for a pipelined design.

- V is the number of unique values in a given dataflow graph.

- V_{reg} is the number of unique values which are assigned to registers for a given design.

- M_i is the number of 2:1 multiplexers needed at each input of operator type i in a given design.

- M_r is the number of 2:1 multiplexers needed for all registers in a given design.

Appendix B

MAHA Usage: Inputs and Outputs

MAHA is a non-pipelined synthesis program that relies upon files for describing a dataflow graph, the module library, and constraints. **MAHA** can be run interactively as well as in batch mode.

B.1 MAHA Inputs

Although **MAHA** usually relies on an interactive environment to control program flow, fixed input data cannot be entered at runtime. Two data input files are required by **MAHA**: a dataflow description and a module library. The dataflow description consists of a single file which contains both a list of nodes and a list of edges; the module library consists of modules and their associated cost, speed, and bit-width. Although both the dataflow file and library file may be named in any manner of your choosing, it is recommended that the primary names be identical while changing the extension to indicate the type of file. For example, the dataflow might be contained in *garbage.dfg* and the module library in *garbage.lib*.

The *dataflow description file* contains the description of a complete dataflow graph. The graph has a number of restrictions on it:

- The graph can contain no more than 1000 nodes or 2000 edges.
- The graph must be acyclic (no loops).
- The top of the graph is indicated by the reserved word *root* which cannot be defined elsewhere. *Root* has NO input edges, only output edges. If

you do not have a *root* node, **MAHA** will add one for you along with the necessary edges emanating from it. If you have a *root* node, it should be of operation type *dummy* since it does not represent any physical operation.

- The bottom of the graph is indicated by the reserved word *outport* and cannot be defined elsewhere. *Outport* has NO output edges, only input edges. If you do not have a *outport* node, **MAHA** will add one for you along with the necessary edges connected to it. If you have a *outport* node, it should be of type *dummy*.

- Conditional branches are indicated by the reserved words *dist* for fork (distribution) nodes and *join* for join nodes. *Dist* has one input arc and one or more output arcs; *join* is the converse.

- Parallel branches may be indicated by the reserved words *parbeg* where the parallel branches begin and *parend* where the parallel branches recombine. However, **MAHA** assumes multiple edges are in parallel implicitly, so these nodes are more for readability.

The *dataflow description file* has both node and edge information in a one node (edge) per line format as follows:

```
node-description-1
node-description-2
    ...
node-description-n

edge-description-1
edge-description-2
    ...
edge-description-m
```

Note the blank line between the node and edge descriptions; this is a REQUIREMENT. Also, one can insert comments into the description. A comment line is indicated by the presence of a pound sign ('#') in the first column; comments may appear anywhere and are ignored by **MAHA**.

B.1.1 Node Description

A node description contains the node name, node type, and bitwidth as follows:

```
node-name node-type bitwidth
```

Node-name is any 1 to 15 character name which is *unique* to the dataflow graph description. The node-type is also a name of up to 15 characters which specifies the function of the node; this node-type MUST match one or more module functions in the module library. Bitwidth is a positive integer from 0 to whatever; a bitwidth of 0 informs **MAHA** that this node is an *implied* node (e.g. one that has no associated cost or delay). For example,

```
add1        add        8
```

names an adder add1 which performs an add function and is of bitwidth 8. There must be at least one add in the module library. Keep in mind that case is important; hence, add and Add are NOT the same.

A brief example of of some valid node names are shown below.

```
c3          dummy    8
x1_p1       bit-read 8
storage1    d-flop 140
add7        add      8
subtraction3sub      5
no_op       dummy    0
flag_check  cmp      5
```

B.1.2 Edge Description

An edge description follows the node description in a dataflow file with an intervening blank line. It consists of a source node, destination node, and bitwidth.

```
source-node   destination-node   bitwidth
```

Like the node description, source-node and destination-node are names up to 15 characters in length. The node names MUST match names previously included in the node description.

An example of edge descriptions using previously listed node names are shown below.

```
x1_p1          add7        8
subtraction3flag_check 5
c3             add7        8
```

B.1.3 Module Description

The second input data file is the module library which consists of a list of individual functional modules and some of their physical parameters. The form of the module library is:

```
module-description-1
module-description-2
    ...
module-description-p
```

Similar to the dataflow graph description, comments are allowed in the module library. Any line with a pound sign ('#') in the first column is ignored by **MAHA**.

Each module-description is of the form:

```
module-name   module-function   bit-width   prop-delay   cost   std-width   nets
```

Module-name is a name of up to 15 characters which is *uniquely* identified in the module library. Module-function is also a name of up to 15 characters and

describes the general function of the module. Some typical general functions are add, sub, or, and, and d-flop. The node-type of every node description MUST match the module-function of one or more modules. Bit-width, prop-delay, and cost are **integer** values which describe the bit width of the module, propagation delay in some arbitrary units, and cost (usually area, although could be power, etc.) in some arbitrary units. Finally, std-wdith and nets are any associated standard cell width and (internal) 2-wire nets; these are only used if one wishes to execute the wiring area estimation program **PLEST** upon completion of **MAHA**. Otherwise, the values can be set to zero as they are not used by **MAHA** for synthesis.

When read into **MAHA**, the module library is reduced to a list which is linked to the node list. Specifically, for each node-type named in the node list, the module library is scanned as follows:

1. If the node-type and module-function are the same, proceed to step 2, else proceed to step 7.

2. If the module bit-width is less than or equal to zero, proceed to step 3, else proceed to step 5.

3. If the module bit-width is zero, the actual cost and prop-delay are defined as the module cost times the node bit-width and module prop-delay times the node bit-width, respectively. Proceed to step 6.

4. If the module bit-width is less than zero, the actual cost is the module cost times the node bit-width whereas the prop-delay is simply the module prop-delay (no adjustment). Proceed to step 6.

5. If the module bit-width is less than the node bit-width, proceed to step 7.

6. The cost and prop-delay information is added to the matching module list.

7. If there are more modules to check, proceed to step 1, else proceed to step 8.

8. An average module is created which has as its prop-delay (cost) the *average* of all prop-delays (costs) from the matching module list.

A pass over the module library is made for each node to create the *average module* which implements each node function. If two nodes generate the same matching module list, they will point to the same *average module*.

Since an *average module* is used by **MAHA**, a module library should contain identical functions with different prop-delays, costs, and bit-widths. If you wish to have direct control over the module library, then only a single compatible module should be included in the library for each node type.

B.2 Executing MAHA

There are two methods for executing **MAHA**: interactive execution or command line execution. Each will be detailed.

B.2.1 Interactive Execution of MAHA

MAHA is executed for interactive synthesis by running the program with no other parameters.

```
maha
```

Once MAHA begins execution, it first inquires for the name of the dataflow description file.

```
Dataflow graph filename?
```

After you enter the name of your file, **MAHA** reads the node and edge list and attaches *root* and *outport* nodes if missing. **MAHA** then marks the conditional paths using a node coloring algorithm. The user is prompted for optional display of the completed node coloring.

```
Show node coloring? (Y/N):
```

If answering **Y**, **MAHA** displays the list of nodes and associated color. Next, **MAHA** prompts for the name of a constraint file.

Constraint filename?

MAHA accepts a set of limited local constraint equations. These are not the *global* goals and constraints on time and area, but rather constraints on individual node area or relative delay between two nodes. Section B.5.3 contains a description of the constraint file. Enter the name of the constraint file. If there are no constraints, as is usually the case just press RETURN with no name.

The last file **MAHA** needs is the module library.

Module library filename?

After the module library filename is entered, **MAHA** generates the *average module* library as described in the previous section. It displays the number of *average modules* and the maximum delay associated with any module; this is also the minimum clock cycle time.

Besides allocation and scheduling of operations, **MAHA** is capable of analyzing register and multiplexer area and delay at the option of the user.

Consider register cost and delay? (Y/N):

If you do not wish to include register analysis, then answer **N**. Answering **Y** brings up a list of the current cost per bit and delay.

Current values: Regcst/bit = 16, regtim = 5. Change? (Y/N):

If these values are acceptable, one should respond with **N**. Otherwise, **MAHA** will prompt for the new cost per bit and delay. Enter the two integer numbers separated by one or more blanks. **MAHA** constructs registers of any length needed using the single-bit values for cost; delay is independent of bitwidth.

The same questions given the user for register analysis are now given for multiplexer analysis. The analysis is specific to the multiplexer being 2:1 single-bit. **MAHA** constructs multiplexers of any bitwidth and any $n : 1$ multiplex

using these 2:1 values. Delay is dependent upon the height of the $n : 1$ tree and is given by

$$\lceil \log_2 n \rceil \times mux_delay$$

At this time, you are prompted for a number of self-explanatory items.

Echo to output file? (Y/N):

Print the node list? (Y/N):

Print the edge list? (Y/N):

Print the module library (Y/N):

For the first of the above inquiries, **MAHA** will prompt for a filename if you choose to echo the output to a file. Echoing is similiar to a script file as all console output is also directed to the specified file.

MAHA now calculates the nominal *critical path* and lists the nodes in it, the total critical path time, and the minimum clock cycle time. (*Critical path* is the longest delay path from *root* to *outport*. Only a single critical path is returned even if there is more than one critical path.) **MAHA** also displays the input processing time and the list of nodes in the critical path in order from left to right starting from *root* and ending at *outport*.

You are now asked for constraints which direct the search for a solution.

Enter maximum time (0 to minimize):

Enter maximum cost (0 to minimize):

There are three choices you have regarding the constraints assigned:

1. You can specify time (some positive integer) and set cost to 0 (to minimize). **MAHA** will search for the cheapest design which meets the time constraint.

2. You can specify cost (some positive integer) and time to 0 (to minimize). **MAHA** will search for the fastest design which meets the cost constraint.

3. You can specify BOTH cost and speed. **MAHA** will find the best design that meets both constraints. Due to the way **MAHA** functions, it will produce the fastest design that meets both of the constraints.

Notice that the case where BOTH maximum time and cost are set to zero (minimize both) is NOT currently allowed. **MAHA** will print a message and ask for the time and cost again.

After displaying the constraints, **MAHA** inquires whether the user wishes to manually control the search

Do you wish to manually control the search? (Y/N):

Normally, you will want the **MAHA** algorithm to find the result for you; in this case, answer **N** to the above question. If, for some reason, you wish to bypass the **MAHA** search algorithm to go directly to a specific design point, answer **Y** to the question. Since **MAHA** acts differently dependent on the answer, **manual** and **automatic** search are discussed separately.

B.2.1.1 MAHA Automatic Operation

When you specify automatic operation, **MAHA** will internally search for the best result. First, the user is prompted for the type of allocation and scheduling.

Allocation: 0 = ASAP only, 1 = ASAP/ALAP :

MAHA has the capability to perform allocation in "as early as possible" or "as late as possible" and take the best of the two results. However, this could result in an execution time which is three times longer than just ASAP allocation. When you first enter a graph or have a very large graph, it is recommended that you respond with **0**. However, for small graphs, you may wish to try both despite the longer execution time. In this case, answer **1**.

MAHA will optionally show all of its internal operations.

Show freedoms and status (detailed information)? (Y/N):

If you answer **Y** to the above question, the "grundgy" detail of **MAHA** operation is shown: bounding the time range to search, buying and sharing of modules, current cost, and percent completion. If you wish to avoid this extensive detail, which is generally only useful for seeing the algorithm in action, enter **N** at the question.

Show ONLY non-inferior solutions? (Y/N):

If you answer **Y**, **MAHA** will give a table of the best results IGNORING the constraints. However, **MAHA** will highlight the best solution. Answering **N** lists all designs **MAHA** produces. The best design meeting the constraints is again listed.

B.2.1.1.1 Manual Operation If manual search has been specified, the user directs all operations.

Enter partition count (positive #),
or clock cycle time (as negative #),
or RETURN to exit:

Manual control of **MAHA** allows either directly specifying the number of partitions (time slots) to cut the dataflow graph or entering the clock cycle time. (To distinguish a partition count from a clock cycle time, the former is entered as a negative number.)

A distinct feature of manual control of MAHA is that you are allowed to exceed the original constraints.

Manual Partitioning:

MAHA will partition a dataflow graph between 1 and n partitions, where n is the number of partitions realized when using the *minimum clock cycle time* discussed earlier. If the number of partitions is within this range, **MAHA** inquires

Allocation: 0 = ASAP only, 1 = ASAP/ALAP :

MAHA has the capability to perform allocation in "as early as possible" or "as late as possible" and take the best of the two results. However, this could result in an execution time which is three times longer than just ASAP allocation. When you first enter a graph or have a very large graph, it is recommended that you respond with **0**. However, for small graphs, you may wish to try both despite the longer execution time. In this case, answer **1**.

MAHA will optionally show all of its internal operations.

Show freedoms and status (detailed information)? (Y/N):

If you answer **Y** to the above question, the "grundgy" detail of **MAHA** operation is shown: bounding the time range to search, buying and sharing of modules, current cost, and percent completion. If you wish to avoid this extensive detail, which is generally only useful for seeing the algorithm in action, enter **N** at the question.

Manual Clock-cycle Entry:

MAHA will partition a dataflow graph based on a preset clock cycle time. The only restriction is that the time must equal or exceed the *minimum clock cycle time* discussed earlier. If the clock cycle time is within this range, **MAHA** inquires

Allocation: 0 = ASAP only, 1 = ASAP/ALAP :

MAHA has the capability to perform allocation in "as early as possible" or "as late as possible" and take the best of the two results. However, this could result in an execution time which is three times longer than just ASAP allocation. When you first enter a graph or have a very large graph, it is recommended that you respond with **0**. However, for small graphs, you may wish to try both despite the longer execution time. In this case, answer **1**.

MAHA will optionally show all of its internal operations.

Show freedoms and status (detailed information)? (Y/N):

If you answer **Y** to the above question, the "grundgy" detail of **MAHA** operation is shown: bounding the time range to search, buying and sharing of modules, current cost, and percent completion. If you wish to avoid this extensive detail, which is generally only useful for seeing the algorithm in action, enter **N** at the question.

B.2.2 Command Line Execution of MAHA

MAHA can be executed directly from the command line with no user interaction. The format is

maha dfg modlib constraints outfile [regflg [muxflg]]

where dfg is the dataflow graph file, modlib is the module library file, constraints is the constraint file (if any), and outfile is the output filename. If there is no constraints file, then a dash ("-") should be entered. Regflg and muxflg are optional flags.

In the command line mode, **MAHA** does not normally consider register or multiplexer costs. However, if regflg is 1, then register costs will be considered. Likewise, if muxflg is 1, then multiplexer costs will be considered.

After scanning the command line and opening the output file, **MAHA** generates all possible designs. These are written in a computer-readable format as described in Section B.5.4. Unless some sort of run-time error occurs, no messages will appear at the console. **MAHA** performs its tasks silently in an identical manner as the automatic search described in Section B.2.1.1; both ASAP and ALAP scheduling are attempted for every design and the best chosen.

B.3 MAHA Interactive Output

Once **MAHA** has completed allocation of the graph, it displays the final clock cycle time, cost, and total time for the graph.

Show the hardware map? (Y/N):

If you wish to see the final allocated results, answer **Y** to the question. **MAHA** will output a table that looks like

HARDWARE	NODE.SLOT	
add8	add8.000	add5.001
r-shift10	div1.001	
add8	add3.001	

The first column contains the list of all hardware purchased, only the function is actually listed. (Notice there is one r-shift10 and two add8s in the example.) At the columns to the right of the hardware is the list of all nodes which are bound to that piece of hardware and the time slot associated with it.

In the example, add1 in the first partition (slot #0) and add5 in the second partition (slot #1) share the same hardware - add8. Add3 and div1 were put into the second partition (slot #1) and do not share their hardware with any other operators.

B.4 An Interactive Example

In this section, the example that was included in the **MAHA** paper, "MAHA: A data path Synthesis Program" by Alice Parker, Jorge Pizarro, and Mitchell Mlinar, ACM/IEEE 23rd Design Automation Conference, June, 1986. The dataflow graph used in this example is reproduced below.

Each node is assigned a distinct name (not to exceed 15 characters) for use by **MAHA**. Below is a copy of the dataflow graph file, example.dfg, which is accessible from the maha directory.

Since the paper was written, there have been some minor changes to **MAHA** - namely, the separation of conditional and parallel branches. The example dataflow graph in the paper has unconditional branches.

```
root       dummy     0
outport    dummy     0
add1       add       8
add2       add       9
add3       add       10
add4       add       9
add5       add       8
sub1       sub       9
sub2       sub       8
sub3       sub       8
div1       r-shift   10
div2       r-shift   10
cmp1       cmp       8
cmp2       cmp       8
cmp3       cmp       8
and1       and       2
inv1       inv       1
out1       buf       1
out2       buf       1
out3       buf       1
D5         parbeg    0
```

D6	parbeg	0
J5	parend	0
J6	parend	0
root	add1	8
root	add5	8
root	add1	8
root	add4	8
root	add2	8
root	add4	8
root	add3	8
root	add5	8
add1	add2	9
add2	add3	10
add3	div1	10
add4	sub1	9
add5	sub1	9
div1	div2	9
div2	D6	8
D6	sub3	8
sub1	cmp3	8
D6	sub2	8
sub3	cmp1	8
sub2	cmp2	8
cmp2	and1	1
cmp3	and1	1
and1	out3	1
cmp1	D5	1
D5	inv1	1
D5	out2	1
inv1	out1	1
out1	J5	1

out2	J5	1
out3	J6	1
J5	J6	1
J6	outport	1

Notice how the above example follows the rules outlined previously.

- there is a single *root* node which starts the dataflow graph
- there is a single *outport* node which ends the dataflow graph
- the dummy-type nodes have a bitwidth of 0. Since the *root* and *outport* nodes are algorithmic conveniences, a bitwidth of 0 informs **MAHA** to ignore and cost and delay for this node.
- there is a blank line between the node list and the edge list

The associated module library for this dataflow graph, example.lib, is reproduced below:

```
dummy     dummy     0  0  0  0
dist      dummy     0  0  0  0  0
join      join      0  0  0  0  0
parbeg    parbeg    0  0  0  0  0
parend    parend    0  0  0  0  0
add2      add       2  40  80  0  0
add4      add       4  72  120  0  0
add8      add       8  120  180  0  0
add12     add       12  150  220  0  0
add16     add       16  200  300  0  0
addn      add       0  20  45  0  0
sub2      sub       2  50  90  0  0
sub4      sub       4  84  130  0  0
sub8      sub       8  140  200  0  0
sub12     sub       12  225  250  0  0
```

sub16	sub	16	240	360	0	0
subn	sub	0	25	50	0	0
mul2	mul	2	80	140	0	0
mul4	mul	4	150	300	0	0
mul8	mul	8	280	640	0	0
mux2	mux	2	30	55	0	0
mux4	mux	4	54	100	0	0
cmp4	cmp	4	70	110	0	0
cmp8	cmp	8	130	180	0	0
cmp12	cmp	12	190	240	0	0
and2	and	2	10	18	0	0
and3	and	3	14	22	0	0
r-shiftn	r-shift	0	44	88	0	0
r-shift4	r-shift	4	44	250	0	0
r-shift8	r-shift	8	44	400	0	0
r-shift12	r-shift	12	44	510	0	0
r-shift16	r-shift	16	44	600	0	0
inv1	inv	1	8	14	0	0
inv2	inv	1	8	25	0	0
buf1	buf	1	10	14	0	0
buf2	buf	1	30	100	0	0
buf3	buf	1	50	150	0	0

The sample module library points out some of the features and restrictions described earlier:

- Each module has a *unique* name.
- The set of module operations is well defined: addition, subtract, multiply, cmp (compare), and, buffer driver, inverter, l-shift (left shift register), r-shift (right shift register/divider), distribute, and join.
- Even fictitious node operations such as parbeg, dummy, and parend MUST be declared in the module library. (Other fictitious nodes include dist and join, but are not used in this example.)

- All of the delay and cost values are positive integers (including zero).

Here is a sample run of **MAHA** using the example.

```
eve[1] maha



                    MAHA v7.01
            USC Design Automation Group

            Mitchell Mlinar, July 1988



Dataflow graph filename? dataflow.dfg

Reading in the nodelist.
There are 24 nodes: roots = 1, outports = 1.

Reading in edgelist.
There are 32 edges.
Checking for extra edges required.
---> 0 extra edges added.
Coloring conditional paths ...

Show node coloring? (Y/N): n

Constraint filename?

Module (mod2) library filename? example.lib

There are 13 modules, minimum time is 240.
```

```
Consider register cost & delay? (Y/N): n

Consider mux cost & delay? (Y/N): n
Input process time:  0.060 seconds.

Echo to output file? (Y/N): n

Print the node list? (Y/N): n

Print the edge list? (Y/N): n

Print the module library? (Y/N): y
```

Module name	Width	Delay	Cost	Size	Nets
dummy0	0	0	0	0	0
add8	8	160	240	0	0
add9	9	200	300	0	0
add10	10	200	300	0	0
sub9	9	240	360	0	0
sub8	8	190	280	0	0
r-shift10	10	44	600	0	0
cmp8	8	130	180	0	0
and2	2	12	20	0	0
inv1	1	8	19	0	0
buf1	1	30	88	0	0
dist0	0	0	0	0	0
join0	0	0	0	0	0

```
Press RETURN to continue --
```

Notice how **MAHA** calculates the *average* of the module library. Each node in the dataflow will have a single module associated with it (but not alloacted yet).

```
Finding the nominal critical path.

The nominal critical path has 13 nodes with a time of 1010.
The minimum clock time is 240.
Critical path process time:  0.000 seconds.


The critical path is:

root            add1            add2            add3
div1            div2            D6              sub2
cmp2            and1            out3            J6
outport


Enter maximum time (0 to minimize):
```

Now that the critical path has been found, we can try to perform the synthesis with a time constraint.

```
Enter maximum time (0 to minimize): 3000

Enter maximum cost (0 to minimize): 0


Constraints:
```

```
   Time: 3000   Cost: minimize

Do you wish to manually control the search? (Y/N): n


Automatic search ...


Allocation:  0 = ASAP only, 1 = ASAP/ALAP : 1

Show freedoms and status (detailed information)? (Y/N): n

Show ONLY non-inferior solutions? (Y/N): n


Partitions  Clock    Time    Cost  Regs  Muxes
     1       1010    1010    3707     3     0
     2        560    1120    3707     6     0
     3        362    1086    3707     8     0
     4     (Cannot reach this)
     5     (Cannot reach this)
     6        240    1440    3167    12     0
Linking add5 between add1 and add2
     2        650    1300    3407     5     0
     3        444    1332    2627     7     0
     4        362    1448    3227     9     0
     5     (Cannot reach this)
     6     (Cannot reach this)
     7        240    1680    2987    13     0
Tagging node div2 as start of new partition
     2        764    1528    2807     5     0
     3        444    1332    2627     7     0
     4        406    1624    2627     8     0
```

```
5        320      1600     2627     11      0
6     (Cannot reach this)
7        240      1680     2387     13      0


Best is time of 1680, clock of 240, cost of 2387  (13 regs, 0 muxes)

Analysis time:  0.020 seconds.

Recalculate the best case showing the hardware map? (Y/N): y


HARDWARE          NODE.SLOT (+ for register)
dummy0            root.000            outport.006
add8              add1+000            add5+001
add9              add4+000            add2+002
add10             add3+003
r-shift10         div1+004            div2.005
dist0             D6.005              D5.006
sub8              sub2+005            sub3+005
cmp8              cmp3+003            cmp1.006            cmp2.006
and2              and1.006
buf1              out1+006            out2+006            out3+006
join0             J6.006              J5.007
inv1              inv1.006
sub9              sub1+002


Bye.
eve[2]
```

Since the cost constraint was never met (minimized cost), the solution with the lowest cost was selected.

B.5 File Formats

Included here is a brief summary of all file formats used by **MAHA**.

B.5.1 Dataflow Description File

The *dataflow description file* has both node and edge information in a one node (edge) per line format as follows:

```
node-description-1
node-description-2
        ...
node-description-n

edge-description-1
edge-description-2
   ...
edge-description-m
```

Note the blank line between the node and edge descriptions; this is a REQUIREMENT. Also, one can insert comments into the description. A comment line is indicated by the presence of a pound sign ('#') in the first column; comments may appear anywhere and are ignored by **MAHA**.

B.5.1.1 Node Description

A node description contains the node name, node type, and bitwidth as follows:

```
node-name node-type bitwidth
```

Node-name is any 1 to 15 character name which is *unique* to the dataflow graph description. The node-type is also a name of up to 15 characters which specifies the function of the node; this node-type MUST match one or more module functions in the module library. Bitwidth is a positive integer from 0 to

whatever; a bitwidth of 0 informs **MAHA** that this node is an *implied* node (e.g. one that has no associated cost or delay). For example,

```
add1        add        8
```

names an adder add1 which performs an add function and is of bitwidth 8. There must be at least one add in the module library. Keep in mind that case is important; hence, add and Add are NOT the same.

A brief example of of some valid node names are shown below.

```
c3           dummy     8
x1_p1        bit-read  8
storage1     d-flop 140
add7         add       8
subtraction3sub        5
no_op        dummy     0
flag_check   cmp       5
```

B.5.1.2 Edge Description

An edge description follows the node description in a dataflow file with an intervening blank line. It consists of a source node, destination node, and bitwidth.

```
source-node   destination-node   bitwidth
```

Like the node description, source-node and destination-node are names up to 15 characters in length. The node names MUST match names previously included in the node description.

An example of edge descriptions using previously listed node names are shown below.

```
x1_p1        add7       8
subtraction3flag_check 5
c3           add7       8
```

B.5.2 Module Description File

The second input data file is the module library which consists of a list of individual functional modules and some of their physical parameters. The form of the module library is:

module-description-1
module-description-2
...
module-description-p

Similar to the dataflow graph description, comments are allowed in the module library. Any line with a pound sign ('#') in the first column is ignored by **MAHA**.

Each module-description is of the form:

module-name module-function bit-width prop-delay cost std-width nets

Module-name is a name of up to 15 characters which is *uniquely* identified in the module library. Module-function is also a name of up to 15 characters and describes the general function of the module. Some typical general functions are add, sub, or, and, and d-flop. The node-type of every node description MUST match the module-function of one or more modules. Bit-width, prop-delay, and cost are **integer** values which describe the bit width of the module, propagation delay in some arbitrary units, and cost (usually area, although could be power, etc.) in some arbitrary units. Finally, std-wdith and nets are any associated standard cell width and (internal) 2-wire nets; these are only used if one wishes to execute the wiring area estimation program **PLEST** upon completion of **MAHA**. Otherwise, the values can be set to zero as they are not used by **MAHA** for synthesis.

When read into **MAHA**, the module library is reduced to a list which is linked to the node list. Specifically, for each node-type named in the node list, the module library is scanned as follows:

1. If the node-type and module-function are the same, proceed to step 2, else proceed to step 7.

2. If the module bit-width is less than or equal to zero, proceed to step 3, else proceed to step 5.

3. If the module bit-width is zero, the actual cost and prop-delay are defined as the module cost times the node bit-width and module prop-delay times the node bit-width, respectively. Proceed to step 6.

4. If the module bit-width is less than zero, the actual cost is the module cost times the node bit-width whereas the prop-delay is simply the module prop-delay (no adjustment). Proceed to step 6.

5. If the module bit-width is less than the node bit-width, proceed to step 7.

6. The cost and prop-delay information is added to the matching module list.

7. If there are more modules to check, proceed to step 1, else proceed to step 8.

8. An average module is created which has as its prop-delay (cost) the *average* of all prop-delays (costs) from the matching module list.

A pass over the module library is made for each node to create the *average module* which implements each node function. If two nodes generate the same matching module list, they will point to the same *average module*.

Since an *average module* is used by **MAHA**, a module library should contain identical functions with different prop-delays, costs, and bit-widths. If you wish to have direct control over the module library, then only a single compatible module should be included in the library for each node type.

B.5.3 Constraint Description File

The current version of **MAHA** optionally accepts a limited set of time constraints. These are contained in a constraint file. Each line is of the form

node-time   cmp-op   node-time [± time-constant]

where node-time is a reference to a specific node time, cmp-op is a comparison operator, and time-constant is some positive integer number.

Node-time is a reference to a specific time of a given node. It consists of a letter, colon (":"), and nodename.

B:nodename	Beginning time of specific node
E:nodename	End time of specific node
D:nodename	Delay time of node operation

The letter specifies the time value to use associated with the node. The nodename MUST have match a *unique* name which is present in the dataflow description file.

There is one exception to the generalized time constraint equation: if the delay of node appears on the left-hand side, the right-hand side may only contain a single constant. This is necessary since the module library is chosen prior to synthesis; **MAHA** does not have the capability to reconstruct a module library on the fly.

Cmp-op allowed are less than (<) or greater than (>); however, **MAHA** implicitly assumes that these operations are less-than-or-equal-to (≤) or greater-than-or-equal-to (≥).

Finally, the constraint file accepts comment lines. Any line with a pound sign in the first column ("#") is ignored. In addition, blank lines are also ignored. A sample file is included below.

```
#
# Constraints file for myex0a.dfg
#
# Comment lines start with #
#

# Blank lines are also allowed
```

```
# Times that make sense (case of 1st char not important):
# B:nodename Beginning of op nodename
# E:nodename End of op nodename
# D:nodename Delay of op nodename
# constant just a number (always positive)
#
# Operations consist only of:
#
# time < time
# time > time
# time < time + constant
# time < time - constant
# time > time + constant
# time > time - constant
#

b:mulc > e:add2
#b:add4 < b:mulc + 1000
b:mula > b:mulc - 7500
d:add4 < 600
```

B.5.4 MAHA Output File

When executed using command line mode, **MAHA** writes a machine readable synthesis file. The format of this file is as follows.

```
design-desription-1
***
design-description-2
***
    ...
```

```
***
design-description-n
***
```

Each design description is offset by a line containing a blank followed by three astericks. A design description is two or more lines as shown.

```
parts [ clock area width nets regs muxes (inserts) ]
nodename-1 nodetype-1 bound partition regflag color
nodename-2 nodetype-2 bound partition regflag color
            ...
nodename-n nodetype-n bound partition regflag color
```

Parts is the number of partitions into which the dataflow graph has been cut with an associated clock cycle time of clock. Area, width, and nets are the calculated area, standard-cell width, and number of 2-wire nets of the resulting design. Regs and muxes are the maximum number of registers and multiplexers needed for this design. Finally, the number of delay nodes inserted into the dataflow graph is indicated by inserts.

If a valid design has been reached, parts will be a positive integer; otherwise, parts will be negative and **no other values will be present on the design header line.** Also, if **MAHA** was executed without considering registers or multiplexers, then the values displayed are invalid and should be discarded.

Following the design header line, the results for each node in the dataflow graph are listed. nodenode-i is the *unique* node name as contained in the dataflow graph description file; nodetype-i is the associated module type and is often a name generated internally in **MAHA**. Bound is an integer index value; nodes which have the same index share the same hardware. Note that two different nodes having the same nodetype do not necessarily have the same bound index. Although they implement the same operation, each is accomplished in a different hardware module.

Partition is the partition index number where the node operation is performed. The value for partition ranges from zero to parts - 1. Regflg indicates whether

the output value(s) from this node are needed in later clock cycles; a value of one (1) indicates that a register is needed to save its value. If not, then a zero (0) is present.

Finally, the conditional coloring of the node is given in the form

color1[:color2[:...]]

where the coloring identifier is one or more 2-digit hexadecimal numbers separated by a colon. Colors which have a length greater than one indicate that this operation is located on some conditional path; the length specifies the depth at which this conditional exists.

A partial **MAHA** output file is shown below and matches the example described earlier.

```
***
4 362 3227 0 180 9 0 (1)
root        dummy0       0   0  0  01
outport     dummy0       0   3  0  02
add1        add8         1   0  0  01
add2        add9         3   1  1  01
add3        add10        4   2  0  01
add4        add9         3   0  1  01
add5        add8         2   0  1  01
sub1        sub9        14   1  1  01
sub2        sub8         8   3  0  02:02
sub3        sub8         8   3  0  02:01
div1        r-shift10    5   2  0  01
div2        r-shift10    6   2  1  01
cmp1        cmp8         9   3  0  02:01
cmp2        cmp8         9   3  0  02:02
cmp3        cmp8         9   2  1  01
and1        and2        10   3  0  02:02
inv1        inv1        13   3  0  02:01:01
out1        buf1        11   3  1  02:01:01
```

```
out2      buf1      11   3   1   02:01:02
out3      buf1      11   3   1   02:02
D5        dist0      7   3   0   02:01
D6        dist0      7   2   0   02
J5        join0     12   4   0   02:01
J6        join0     12   3   0   02
***
```

Appendix C

PLA Loop Counter Area Evaluation

Using a counter as part of the loop control has a variety of implementations. Two primary schemes involve building a counter as part of the PLA or attaching a separate piece of hardware. Four operational considerations are

1. initialization of the counter,

2. decrement of the counter,

3. testing for count completion,

4. and uniquely defining the state (for clocking the correct data path register).

Initialization of the counter must be performed directly by the PLA for the internal counter. Although an external counter could have a prewired starting value, this is less attractive since it may preclude using the counter for handling other loops (if they have a different number of iterations). Hence, the external counter will be initialized by the PLA, which requires n additional output lines for an n-bit counter.

For the **decrement** loop operation, an external counter is supplied a single output trigger by the PLA. Conversely, the internal counter has input lines containing the previous value and product terms to determine the next value. The optimal number of product terms is the sum of the minimal covering for each state bit and is demonstrated in Figure C.1, where the bits are numbered from 1 up. Karnaugh map coverage dictates that a given bit position, m, where 1 is the least-significant bit position, can be minimally covered with m terms. Hence, for a given n-bit counter, the total number of product terms is the summation from

Terms	Covers	
10000...0	1	
011xx...x	2^{n-2}	
0101x...x	2^{n-3}	n
01001...x	2^{n-4}	
...	...	
01000...1	2^{n-n}	
↑ bit posn 'n'	2^{n-1}	

Figure C.1: Minimal Covering of product terms for Counter

1 to n or $\frac{n(n+1)}{2}$ for a 2^n iteration loop. If the number of iterations for a given loop, $N(\alpha_i)$, falls in the range of $2^{n-1} < N(\alpha_i) < 2^n$, then the *minimal* number of product terms is the lower bound.

$$p_{int\ cntr} = \frac{n'(n'+1)}{2} \tag{C.1}$$

where $n' = \lfloor \log_2 N(\alpha_i) \rfloor$.

The **termination test** entails checking the counter for a value of zero. Since it is far cheaper to check for a zero in external hardware already given the external counter, a single PLA input status bit suffices. A product term is necessary for both internal and external counters to determine the next state for loop termination or continuation.

Finally, a unique value *may* be required for each loop to distinguish the register being controlled. (For example, a 16-bit multiply implemented using an 8-bit multiplier would store the intermediate values in separate registers in each of the four loops.) This also entails decoding the contents of the counter to determine the iteration number which, when combined with the loop state, yields a unique register reference. Decoding can be performed internal to the PLA, but external decoding is more cost effective as shown in Table C.1. The large difference in areas would be further compounded by having larger state machines or directly building a smaller decoder from CMOS transistors rather than **inverter** and **nand** standard cells. Consequently, decoding of the unique

Table C.1: Comparison of Internal versus External Decoding of Loop Counter

Iterations	Δ PLA Area [1]	External Decode Area [2]
2	153	3
4	355	21
8	574	50
16	809	115

Area in mil^2

[1] The PLA area was based upon the *minimal* number of terms to achieve the decode for the simplest 2-state PLA.
[2] The external area was based upon a standard decoder using inverters and nand gates from the RCA CADDAS library.

register value, if necessary, should always be performed external to the PLA itself.

To compare the cost of each implementation, the impact on the PLA parameters is summarized in Table C.2.

Collecting the terms in Table C.2, assuming the number of iterations, $N(\alpha_i)$, is some positive integer power-of-2, and ignoring the decoding area (which is the same for both the internal and external realizations), the *minimum* area contribution for an internal PLA counter can be derived by using the area equation for the PLA based upon its inputs (i), outputs (p), and product terms (p),

$$A_{cp} \approx c_1(2i + o) + c_2 p + c_3(2i + o) + c_4 \tag{C.2}$$

where $c_1, \ldots, c_4$, are constants for a specific PLA tool and technology. Assuming the PLA exists and only the internal counter is being added, the incremental PLA area associated with this counter is

$$Area_{int\ cntr} \approx 3nc_1 \left(\frac{n'(n'+1)}{2} + 1 \right) + c_2 \left(\frac{n'(n'+1)}{2} + 1 \right) + 3c_3 n \tag{C.3}$$

For an external counter, the incremental area of the PLA is

$$Area_{ext\ cntr} \approx c_1(n+3) + c_2 + c_3(n+3) + nA_{cntr} \tag{C.4}$$

Table C.2: Summary of Counter Impact on PLA Parameters

Type	Term	Δ input	Δ output	Δ product terms
Internal	initialize	–	n	–
	decrement	n	–	$\frac{n'(n'+1)}{2}$
	termination test	–	–	1
	unique reg op [1]	–	– (2^n)	– (2^n)
External	initialize	–	n	–
	decrement	–	1	–
	termination test [2]	1 (n)	–	1
	unique reg op [1]	–	– (2^n)	– (2^n)

$$n = \lceil \log_2 N(\alpha_i) \rceil$$
$$n' = \lfloor \log_2 N(\alpha_i) \rfloor$$

Values in the columns represent the *incremental* number of terms due to each loop operation
[1] The value in parentheses is included if register control signals must be unique for each iteration and the decoding is performed internal to the PLA.
[2] The value in parentheses is included if there is no external signal which detects a count of zero.

Clearly, Equation C.3 grows more rapidly than Equation C.4 as n increases. Even for a value of $N(\alpha_i) = 2$, the PLA counter has an area 222 mil^2 larger than an external counter. Thus, counters used for the control of loops should be realized using external hardware controlled by the PLA rather than extending the PLA to include an internal counter.

C.1 Product Terms in the PLA for the Register Control Method

In Chapter 3, the assertion was made that the number of product terms needed for "register control" was the same as for the "stage control" method. That claim is justified in this appendix.

In a PLA having ζ states, the number of output lines, o, is determined by the number of state bits ($\lceil \log_2 \zeta \rceil$) which must be fed back to the input through a register, plus other required control lines. These include the number of output lines used to control registers (R_c), multiplexers (M), auxiliary hardware such

as ALUs (A), and the number of additional output lines associated with loop control (o_{loop}) and conditional branches (o_{cond}). Under the "register control" method, the number of outputs is

$$o = \lceil \log_2 \zeta \rceil + R_c + M + A + o_{loop} + o_{cond} \tag{C.1.5}$$

With the "stage control" method, the number of product terms is fixed. This may not be true in the "register control" method if there are conditionals, loops, multiplexer and auxiliary control. Since the goal is to minimize the number of product terms for "register control", M, A, o_{loop}, and o_{cond} will not be considered for this discussion.

The R_c register control lines are individually labeled as R^1, R^2, ..., R^{R_c}. Operation of a given R^y output line in a PLA is its list of product terms, pt_α (where α is some unique identifier for each term) that are ORed together,

$$R^y = pt_a \cup pt_b \cup \ldots \tag{C.1.6}$$

Similarly, each next state output line, St^i, is also constructed by ORing together one or more product terms.

By the uniqueness requirement of the register output lines where no two lines may produce identical control over all states,

$$R^x \neq R^y, x \neq y \tag{C.1.7}$$

In other words, R^x and R^y must differ by *at least* one product term or they are not unique.

In the "stage control" method, a product-term is constructed for each data path state. Hence, the number of product terms (p) is equal to the number of control states for a basic PLA or

$$p = \zeta \tag{C.1.8}$$

This is also clearly the maximum number of product terms needed for "register control". In order to reduce the number of product terms, two conditions must hold:

1. There exists at least one product term, pt_a, which can be combined with another product term, pt_a, where $a \neq b$.

2. In *any* PLA output line where pt_a is one of the ORed terms shown in Equation C.1.6, pt_b must also be one of the ORed terms.

These conditions will now be examined.

1: Each product-term can be described by ANDed elements of one or more input lines ($\imath$) (or their inverse $\bar{\imath}$) as

$$pt_\alpha = \imath_1^\alpha \cap \ldots \tag{C.1.9}$$

The "size" of the product-term, $\mid pt_\alpha \mid$, is the total number of elements which are ANDed together. (An element is either some input line or its inverse.)

Given two product terms, pt_i and pt_j, to combine them into a single product term, $pt_k = pt_i \cap pt_j$, two criteria must be met. First, it must be feasible to combine the Boolean terms. If the "size" of one differs from the other, it is not possible to combine them. Thus,

$$\mid pt_i \mid = \mid pt_j \mid \tag{C.1.10}$$

Every element in pt_i *must* also identically match some element in pt_j excepting *one-and-only-one* element in pt_i and pt_j which do not match. This element in pt_i, $\imath_a^i$, and in pt_j, $\imath_b^j$, must be the inverse of one another.

$$\imath_a^i = \bar{\imath}_b^j \tag{C.1.11}$$

If pt_i and pt_j match in all elements, then pt_i and pt_j are redundant which is not allowed. If pt_i and pt_j have two or more elements which are inverses as defined in C.1.11, it is not possible to combine the two product terms.

Since the PLA as constructed in this thesis has sequential binary representations for each state, this implies that there are two states either 1, 2, or some

power-of-2 states apart that can have their inputs combined into a single product term.

2: The second criterium for combining pt_i and pt_j entails that neither of these product terms are used individually. This exclusion must apply to both register control and state output lines. Hence, we have the following conditions.

1. There exists no register control line R^a such that

$$pt_i \in R^a \wedge pt_j \notin R^a \vee pt_i \notin R^a \wedge pt_j \in R^a \tag{C.1.12}$$

   Thus, for every register control line where pt_i forms part of the control equation, pt_j is also part of this equation. This implies that R^a is asserted for some states that are exactly 1, 2, 4, or some power-of-2 number of states apart.

2. There also exists no state bit control line St^b such that

$$pt_i \in St^b \wedge pt_j \notin St^b \vee pt_i \notin St^b \wedge pt_j \in St^b \tag{C.1.13}$$

   Since state bits are sequential binary representations in **PASTA**, this condition will usually hold, unless the number of states in the PLA is *not* a power-of-2.

These severe conditions limit the circumstances where the number of product terms in the register control method can be reduced to less than the stage control method. Not only must a register be operated in specific states, but the PLA should be a power-of-2 number of states.

It is possible to construct artificial designs where the conditions described can be met. However, upon examining typical designs, it is clear that normal designs do not have the degree of control symmetry necessary. Registers, particularly those which have been optimally shared, have rarely exhibited any control ordering in the designs produced in this thesis. Only two cases have arisen experimentally where these conditions were met; each had one less product-term than expected.

Appendix D

PASTA Usage: Inputs and Outputs

PASTA is self-prompting for all parameters needed. However, it does rely on an input file and has a terse display format, both or which are explained here.

D.1 PASTA Input

The input file to PASTA contains as much information as the user can estimate (or the actual results) for the data path scheduling. This file contains one or more descriptions of the data path to be controlled; each description is in the following form:

```
= stages registers [multiplex]
[(expression describing loops and conditionals)]
```

The first line of each description is started by an equal sign ('=') which signifies the start of a new PLA estimate. On the same line in order are:

- *stages* which is the number of stages into which the data path is divided
- *registers* which is the number of registers used in the data path (either an estimate or the actual value)
- *multiplex* which is the number of additional multiplexer control lines needed, if any (optional). Normally, the state bits of the PLA are usually sufficient for controlling any arbitrary $n : 1$ multiplexer in practical designs. However, not all designs can be controlled in this fashion efficiently so this option has been provided.

If the data path being controlled has any conditional paths or loops, then an expression describing this additional control is necessary. (In the absence of this expression, it is assumed that the data path has no loops or conditionals.) The entire expression must be surrounded by parentheses '()' and can be up to 1024 characters over multiple-lines. It has the following pseudo-BNF definition; explicit characters are in single quotes.

```
expr := '('expr')' |
        expr PATHOP expr |
        con-expr |
        loopv LOOPOP expr |
        constant

loopv := desig[:constant] |
         constant[:constant]

con-expr := desig CONOP '{' expr [',' expr ... ] '}'

desig := variable |
         variable '[' constant ']'

variable := alpha-string

alpha-string := alpha-char |
                alpha-char string

string := alpha-char |
          digit

constant := digit |
            digit constant

digit := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

```
alpha-char := 'A-Za-z'

PATHOP := '&' |
          '+'

CONOP := '|'

LOOPOP := '*'
```

The order of operations and their meaning recognized listed from highest precedence to least is:

- | for conditional paths. (Ex: var|4,3 means 4 states OR 3 states are executed, depending upon the value of 'var'.)
- * for loop designator (Ex: 4*3 means 4 iterations of 3 states)
- & for parallel paths (Ex: 4&3 means 4 states in parallel with 3)
- + for serial paths (Ex: 4+3 means 4 states followed by 3 more)

Conditionals (and sometimes loops) refer to a variable which controls is used to make a conditional (loop) decision. This variable name may be up to 31 characters and should be the same name for all loops/conditionals which utilize that variable.

The variable is optionally followed by a bitwidth in '[]'; a default value of one bit is otherwise assumed. Note that the bitwidth designator can *only* be used at the first occurrance of the variable otherwise an error condition is produced.

Each branch of a multi-way conditional is separated by a comma in the conditional expression. Regardless of whether the conditional variable has had its bitwidth declared, PASTA will increase its bitwidth (if necessary) so that it can accomodate the multi-way selection. Thus, the user is free to leave off the bitwidth for selection of multi-way conditionals and PASTA will automatically make it the minimum number possible.

There are two types of loops which can be specified: constant and variable. A constant loop will either be unwound or controlled via an external counter, depending upon the number of states involved and the loop count. A variable loop uses 'desig' as the variable to test after each loop to determine continuation. In addition, there is a constant after the colon (':') which can be used to specify the degree of register reuse for each loop (see the paper for a description of this term, Kreg). This value can ONLY be between 0 and 100. A 0 indicates that all registers within the loop are reused during each iteration; 100 indicates that each loop uses a different register.

D.2 PASTA Output

The output header and results appears as follows:

```
                                          Area (mil sq)
Sta Reg  Ty  Ins  Out  Pts  Fld  Dnsity   Unfold     Folded
```

where

```
  'sta' is the estimated number of PLA states
  'reg' is the number of registers
  'ty' is the PLA type: "St" is the stage control method PLA
"Re" is the register control method PLA
  'ins' is the number of inputs into the PLA (including state bits)
  'outs' is the number of outputs from the PLA (including state bits)
  'pts' is the number of product-terms in the PLA
  'fld' is the expected number of output folds possible in the PLA
  'dnsity' is the cross-point density or the number of transistors
        divided by the number of intersections (output side ONLY)
  'unfold' is the estimated for the unfolded PLA area
  'folded' is the estimated for the folded PLA area
```

The area includes any external counters or other hardware needed by the controller itself.

Table D.1: Fixed Values in **PASTA**

Name	Meaning
c1	PLA aggregate block size for all "interior" blocks these have in/out/pterms passing through them.
c2	PLA block size of pterm pre-charge/load blocks
c3	PLA block size of in/out pre-charge/load/driver blocks
c4	PLA area offset constant not related to in/out/pterms
a_cntr	Area of 1-bit cascadable counter with load/decr/zero detect
a_demux	Area of 1-bit 1 : 2 demux
a_and	Area of 2-input AND gate

D.3 Notes

The **PASTA** program is currently hardcoded for the Berkeley toolset as well for the external counter plus associated hardware. Static values in the file "pasta8.h" can be altered and PASTA recompiled with the new coefficients. These are:

D.4 PASTA Example

```
************************************************************
************************************************************
***                                                      ***
***                                                      ***
***     USC - Advanced Design Automation System (ADAM)   ***
***                                                      ***
***             PLA Control Area Estimation              ***
***                                                      ***
***     for Pipelined and Non-Pipelined Designs          ***
***                                                      ***
***                                                      ***
***                  Mitchell Mlinar                     ***
***                                                      ***
***       Version 2.73             Aug 1989              ***
***                                                      ***
***                                                      ***
************************************************************
************************************************************
```

```
Enter PASTA input filename: testout.pasta

                                            Area (mil sq)
Sta Reg  Ty  Ins  Out  Pts  Fld  Dnsity   Unfold     Folded
  8   1  Re    3    4    8    1  0.6250   212.06     212.06
 14   1  Re    4    5   14    1  0.4643   367.21     321.33
 14   1  Re    5    5   15    1  0.4062   446.57     396.57
 12   1  Re    5    5   13    1  0.3810   413.09     367.21
```

```
 14   1  Re     5    5   15    1  0.4062    446.57      396.57
 22   1  Re     7    6   24    2  0.4653    713.41      646.95

CPU processing time:    0.040 seconds

adam[2]
```

Appendix E

REG Usage: Inputs and Outputs

REG estimates the register count for pipelined and non-pipelined designs given the dataflow graph. For pipelined designs, the number of microcycles (partitions/initiation interval) must also be given.

E.1 REG Input

REG can be run interactively by entering:

`reg`

The program will prompt for the parameters it needs. Upper case words signify program output.

`ENTER P FOR PIPELINED OR N FOR NON_PIPELINED DESIGN:`

Type P if register estimation of pipelined design is to be performed. For non-pipelined design style type N.

`ENTER DATAFLOW GRAPH NAME:`

Enter the name of the file containing the data flow graph. For the format of the dataflow graph, see the documentation on MAHA.

`ENTER MODULE LIBRARY NAME:`

Enter the name of the module library. For the format of the module library, see the documentation on MAHA.

ENTER PARTITIONS AND LATENCY:

This question is *only* asked for pipelined design style. Type the initiation interval and the partition count. From these two parameters the value of the micro cycle is computed which is used to estimate the register count for pipelined designs.

E.2 REG Output

Included here is a sample of **REG** output. Both non-pipelined and pipelined examples are shown.

E.2.1 Non-pipelined Register Estimation

```
************************************************************
************************************************************
***                                                      ***
***                                                      ***
***    USC - Advanced Design Automation System (ADAM)    ***
***                                                      ***
***              Register Area Estimation                ***
***                                                      ***
***     for Pipelined and Non-Pipelined Designs          ***
***                                                      ***
***                                                      ***
***                  Mitchell Mlinar                     ***
***                                                      ***
***      Version 1.92                Nov 1989            ***
***                                                      ***
***                                                      ***
************************************************************
************************************************************
```

```
Enter P for pipelined or N for non-pipelined design: n

Enter dataflow graph name: /usr/eve3/mlinar/mahac/ar.dfg

Non-pipelined design with graph length of 8

Minimum registers (partitions = 1):    2  (32 regbits)
Estimate registers for partitions > 1: 5 (80 regbits)

Runtime:  0.160 seconds

adam[2]
```

E.2.2 Pipelined Register Estimation

```
************************************************************
************************************************************
***                                                      ***
***                                                      ***
***     USC - Advanced Design Automation System (ADAM)   ***
***                                                      ***
***              Register Area Estimation                ***
***                                                      ***
***     for Pipelined and Non-Pipelined Designs          ***
***                                                      ***
***                                                      ***
***                  Mitchell Mlinar                     ***
***                                                      ***
***      Version 1.92              Nov 1989              ***
***                                                      ***
***                                                      ***
```

```
***********************************************************
***********************************************************




Enter P for pipelined or N for non-pipelined design: p

Enter dataflow graph name: /usr/eve3/mlinar/mahac/ar.dfg

Pipelined design with graph length of 8
Setup runtime:  0.180 seconds


Enter partitions and latency: 6 1
RPmin = 22   RPmax = 72   Regs = 47  (752 regbits)  [ 0.000 seconds]

Enter partitions and latency: 12 4
RPmin = 16   RPmax = 36   Regs = 26  (416 regbits)  [ 0.000 seconds]

Enter partitions and latency: 19 16
RPmin = 14   RPmax = 24   Regs = 19  (304 regbits)  [ 0.000 seconds]

Enter partitions and latency:

adam[3]
```

Appendix F

MUX Usage: Inputs and Outputs

MUX estimates the multiplexer count and size (in bits) for pipelined and non-pipelined designs given the dataflow graph. For pipelined designs, the number of microcycles (partitions/initiation interval) must also be given.

F.1 MUX Input

MUX can be run interactively by entering:

```
mux
```

The program will prompt for the parameters it needs. Upper case words signify program output.

```
ENTER P FOR PIPELINED OR N FOR NON_PIPELINED DESIGN:
```

Type P if multiplexer estimation or pipelined design is to be performed. For non-pipelined design style type N.

```
ENTER DATAFLOW GRAPH NAME:
```

Enter the name of the file containing the data flow graph. For the format of the dataflow graph, see the documentation on MAHA.

```
ENTER MODULE LIBRARY NAME:
```

Enter the name of the module library. For the format of the module library, see the documentation on MAHA.

ENTER PARTITIONS (i-j) AND NUMBER OF REGISTERS (k-l):

This is asked for non-pipelined design style. Enter the number of partitions which must lie between integers i and j, and the number of registers which must lie between k and l. The number of registers can be obtained from register estimation or actual synthesis.

ENTER PARTITIONS (i-j) LATENCY (m-n) AND NUMBER OF REGISTERS (k-l):

This is the question asked for pipelined design style. Enter the number of partitions which must lie between integers i and j, the value of initiation interval (i.e. the number of clock cycles between two successive initiations of the pipeline), and the number of registers which must lie between k and l. The number of registers can be obtained from register estimation or synthesis.

ENTER NUMBER OF MODULES FOR (module-name):

Type the quantity of modules of type module-name.

F.2 MUX Output

Included here is an example showing MUX output.

```
*************************************************************
*************************************************************
***                                                       ***
***                                                       ***
***      USC - Advanced Design Automation System (ADAM)   ***
***                                                       ***
***             Multiplexer Area Estimation               ***
***                                                       ***
***      for Pipelined and Non-Pipelined Designs          ***
***                                                       ***
***                                                       ***
***                   Mitchell Mlinar                     ***
```

```
***                                                        ***
***        Version 1.42              Oct 1989              ***
***                                                        ***
***                                                        ***
**************************************************************
**************************************************************


Enter P for pipelined or N for non-pipelined design: n

Enter dataflow graph name: fir.dfg

Enter module library name: lib.ff
Values in graph: 47   Incoming: 24   Outgoing: 1
Input processing time:   0.020

Non-pipelined design with graph length of 9

Enter partitions (1-25) and # of registers (1-23): 2 8

Enter # of modules for add16 [maximum: 15, default: 8]: 8

Enter # of modules for mul16 [maximum: 8, default: 4]: 7

    Op:       8 add16      Mux/pin:   6   Subtotal 2:1 = 192
    Op:       7 mul16      Mux/pin:   1   Subtotal 2:1 =  32
    Registers =   8        Muxes:     3   Subtotal 2:1 =  48

    Total 2:1 muxes = 272 in 2 stages  [runtime of   0.080 sec.]

Enter partitions (1-25) and # of registers (1-23): 5 6
```

```
Enter # of modules for add16 [maximum: 15, default: 3]: 4

Enter # of modules for mul16 [maximum: 8, default: 2]: 2

    Op:        4 add16      Mux/pin:   9   Subtotal 2:1 = 288
    Op:        2 mul16      Mux/pin:   5   Subtotal 2:1 = 160
    Registers =    6        Muxes:     4   Subtotal 2:1 =  64

    Total 2:1 muxes = 512 in 5 stages  [runtime of    0.160 sec.]

Enter partitions (1-25) and # of registers (1-23): 15 5

Enter # of modules for add16 [maximum: 15, default: 1]:

Enter # of modules for mul16 [maximum: 8, default: 1]:

    Op:        1 add16      Mux/pin:   9   Subtotal 2:1 = 288
    Op:        1 mul16      Mux/pin:   6   Subtotal 2:1 = 192
    Registers =    5        Muxes:     1   Subtotal 2:1 =  16

    Total 2:1 muxes = 496 in 15 stages  [runtime of    0.160 sec.]

Enter partitions (1-25) and # of registers (1-23):

adam[3]
```

Appendix G

Floating Point Coprocessor Description

Mike McFarland of Boston College and AT&T Bell Laboratories developed the floating point coprocessor example used in this thesis. Both the Value Trace and Park Normal Form descriptions are included.

G.1 Floating Point Coprocessor Value Trace

```
v26 %v25 001001001000010000-US * * REPEAT1;
i1 r15 <8> A.EXP
i2 r16 <63> T.FRACT
i3 r18 <8> T.EXP
x1 +:US (v26.i1:A.EX) (*.c2=1)
p1 * <9> *
x2 <r> (x1.p1) (*.c3=0)
p1 r15 <8> A.EXP
x3 SR0:US (v26.i2:T.FR) (*.c1=1)
p1 r16 <63> T.FRACT
x4 EQL:US (v26.i3:T.EX) (x2.p1:A.EX)
p1 * <1> *
x5 SELECT (x4.p1)
b1.x5 BRANCH [TRUE]
b1.x5 ENDBR
b2.x5 BRANCH [0]
b2.x5 ENDBR
```

```
x5 ENDSEL
x8 LEAVE @v25:PREN (x2.p1:A.EX) (x3.p1:T.FR)
o1 r15 <8> A.EXP
o2 r16 <63> T.FRACT
----------

v27 %v25 001001001000010000-US * * REPEAT2;
i1 r18 <8> T.EXP
i2 r16 <63> T.FRACT
i3 r15 <8> A.EXP
x1 +:US (v27.i1:T.EX) (*.c2=1)
p1 * <9> *
x2 <r> (x1.p1) (*.c3=0)
p1 r18 <8> T.EXP
x3 SR0:US (v27.i2:T.FR) (*.c1=1)
p1 r16 <63> T.FRACT
x4 EQL:US (x2.p1:T.EX) (v27.i3:A.EX)
p1 * <1> *
x5 SELECT (x4.p1)
b1.x5 BRANCH [TRUE]
b1.x5 ENDBR
b2.x5 BRANCH [0]
b2.x5 ENDBR
x5 ENDSEL
x8 LEAVE @v25:PREN (x2.p1:T.EX) (x3.p1:T.FR)
o1 r18 <8> T.EXP
o2 r16 <63> T.FRACT
----------

v28 %v24 001001000000010000-US * * NORMALIZE;
i1 r13 <63> A.FRACT
i2 r15 <8> A.EXP
i3 r30 <140> WAIT
```

```
i4 r9 <5> FP.IR
i5 r16 <63> T.FRACT
i6 r18 <8> T.EXP
i7 r17 <1> T.SIGN
i8 r14 <1> A.SIGN
i9 r19 <1> C
i10 r4 <1> FP.ERROR
i11 r20 <63> FP.REG
i12 r21 <8> EXP.REG
i13 r22 <1> SIGN.REG
i14 r6 <1> READY
i15 r5 <16> DATA
i16 r7 <1> Z
i17 r8 <1> N
i18 r2 <1> GO
x1 <r> (v28.i1:A.FR) (*.c4=62)
p1 * <1> A.FRACT
x2 EQL:US (x1.p1:A.FR) (*.c5=1)
p1 * <1> *
x3 SELECT (x2.p1)
b1.x3 BRANCH [TRUE]
b1.x3 ENDBR
b2.x3 BRANCH [0]
b2.x3 ENDBR
x3 ENDSEL
x5 SL0:US (v28.i1:A.FR) (*.c1=1)
p1 r13 <63> A.FRACT
x6 -:TC (v28.i2:A.EX) (*.c2=1)
p1 * <9> *
x7 <r> (x6.p1) (*.c3=0)
p1 r15 <8> A.EXP
x8 LEAVE @v24:INST (x5.p1:A.FR) (x7.p1:A.EX)
o1 r13 <63> A.FRACT
```

```
o2 r15 <8> A.EXP
----------

v29 %v24 001001000000010000-US * * IN.NORMALIZE;
i1 r13 <63> A.FRACT
i2 r15 <8> A.EXP
i3 r9 <5> FP.IR
i4 r16 <63> T.FRACT
i5 r18 <8> T.EXP
i6 r17 <1> T.SIGN
i7 r14 <1> A.SIGN
i8 r19 <1> C
i9 r4 <1> FP.ERROR
i10 r20 <63> FP.REG
i11 r21 <8> EXP.REG
i12 r22 <1> SIGN.REG
i13 r6 <1> READY
i14 r5 <16> DATA
i15 r7 <1> Z
i16 r8 <1> N
i17 r2 <1> GO
x1 <r> (v29.i1:A.FR) (*.c4=62)
p1 * <1> A.FRACT
x2 EQL:US (x1.p1:A.FR) (*.c5=1)
p1 * <1> *
x3 SELECT (x2.p1)
b1.x3 BRANCH [TRUE]
b1.x3 ENDBR
b2.x3 BRANCH [0]
b2.x3 ENDBR
x3 ENDSEL
x5 SLO:US (v29.i1:A.FR) (*.c1=1)
p1 r13 <63> A.FRACT
```

```
x6 -:TC (v29.i2:A.EX) (*.c2=1)
p1 * <9> *
x7 <r> (x6.p1) (*.c3=0)
p1 r15 <8> A.EXP
x8 LEAVE @v24:INST (x5.p1:A.FR) (x7.p1:A.EX)
o1 r13 <63> A.FRACT
o2 r15 <8> A.EXP
----------

v25 %v24 00100000000010000-US * * PRENORMALIZE;
i1 r15 <8> A.EXP
i2 r18 <8> T.EXP
i3 r16 <63> T.FRACT
x1 NEQ:US (v25.i1:A.EX) (v25.i2:T.EX)
p1 * <1> *
x2 SELECT (x1.p1)
b1.x2 BRANCH [TRUE]
x3 GTR:US (v25.i1:A.EX) (v25.i2:T.EX)
p1 * <1> *
x4 SELECT (x3.p1)
b1.x4 BRANCH [0]
x5 CALL @v26:REPE (v25.i1:A.EX) (v25.i3:T.FR) (v25.i2:T.EX)
p1 r15 <8> A.EXP
p2 r16 <63> T.FRACT
b1.x4 ENDBR (x5.p2:T.FR) (x5.p1:A.EX) (v25.i2:T.EX)
b2.x4 BRANCH [1]
x6 CALL @v27:REPE (v25.i2:T.EX) (v25.i3:T.FR) (v25.i1:A.EX)
p1 r18 <8> T.EXP
p2 r16 <63> T.FRACT
b2.x4 ENDBR (x6.p2:T.FR) (v25.i1:A.EX) (x6.p1:T.EX)
x4 ENDSEL
p1 r16 <63> T.FRACT
p2 r15 <8> A.EXP
```

```
p3 r18 <8> T.EXP
b1.x2 ENDBR (x4.p3:T.EX) (x4.p2:A.EX) (x4.p1:T.FR)
b2.x2 BRANCH [0]
b2.x2 ENDBR (v25.i2:T.EX) (v25.i1:A.EX) (v25.i3:T.FR)
x2 ENDSEL
p1 r18 <8> T.EXP
p2 r15 <8> A.EXP
p3 r16 <63> T.FRACT
x7 LEAVE @v25:PREN (x2.p1:T.EX) (x2.p3:T.FR) (x2.p2:A.EX)
o1 r18 <8> T.EXP
o2 r16 <63> T.FRACT
o3 r15 <8> A.EXP
----------

v24 %v23 00100000010010000-US * * INST.LOOP;
i1 r2 <1> GO
i2 r3 <6> INST.IN
i3 r20 <63> FP.REG
i4 r16 <63> T.FRACT
i5 r21 <8> EXP.REG
i6 r18 <8> T.EXP
i7 r22 <1> SIGN.REG
i8 r17 <1> T.SIGN
i9 r13 <63> A.FRACT
i10 r15 <8> A.EXP
i11 r14 <1> A.SIGN
i12 r19 <1> C
i13 r4 <1> FP.ERROR
i14 r6 <1> READY
i15 r5 <16> DATA
i16 r7 <1> Z
i17 r8 <1> N
x1 <r> (v24.i1:GO) (*.c3=0)
```

```
p1 r2 <1> GO
x2 WAIT (x1.p1:GO)
p1 r30 <140> WAIT
x3 <r> (v24.i2:INST) (*.c3=0)
p1 r3 <6> INST.IN
x4 <r> (x3.p1:INST) (*.c3=0)
p1 r9 <5> FP.IR
x5 <r> (x4.p1:FP.I) (*.c6=3)
p1 * <2> OP
x6 SELECT (x5.p1:OP)
b1.x6 BRANCH [0]
x7 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x8 [r] (v24.i3:FP.R) (x7.p1:REG)
p1 r16 <63> T.FRACT
x9 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x10 [r] (v24.i5:EXP) (x9.p1:REG)
p1 r18 <8> T.EXP
x11 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x12 [r] (v24.i7:SIGN) (x11.p1:REG)
p1 r17 <1> T.SIGN
x13 EQL:US (x8.p1:T.FR) (*.c8=0)
p1 * <1> *
x14 SELECT (x13.p1)
b1.x14 BRANCH [TRUE]
b1.x14 ENDBR
b2.x14 BRANCH [0]
b2.x14 ENDBR
x14 ENDSEL
x16 EQL:US (v24.i9:A.FR) (*.c8=0)
p1 * <1> *
```

```
x17 SELECT (x16.p1)
b1.x17 BRANCH [TRUE]
x18 @ (x10.p1:T.EX) (x12.p1:T.SI)
p1 * <9> *
x19 @ (x18.p1) (x8.p1:T.FR)
p1 * <72> *
x20 <r> (x19.p1) (*.c3=0)
p1 r13 <63> A.FRACT
x21 <r> (x19.p1) (*.c9=63)
p1 r14 <1> A.SIGN
x22 <r> (x19.p1) (*.c10=64)
p1 r15 <8> A.EXP
x23 <r> (x4.p1:FP.I) (*.c11=2)
p1 * <1> MODE
x24 SELECT (x23.p1:MODE)
b1.x24 BRANCH [TRUE]
x25 NOT:US (x21.p1:A.SI)
p1 r14 <1> A.SIGN
b1.x24 ENDBR (x25.p1:A.SI)
b2.x24 BRANCH [0]
b2.x24 ENDBR (x21.p1:A.SI)
x24 ENDSEL
p1 r14 <1> A.SIGN
b1.x17 ENDBR (x24.p1:A.SI) (x22.p1:A.EX) (x20.p1:A.FR)
b2.x17 BRANCH [0]
b2.x17 ENDBR (v24.i11:A.SI) (v24.i10:A.EX) (v24.i9:A.FR)
x17 ENDSEL
p1 r14 <1> A.SIGN
p2 r15 <8> A.EXP
p3 r13 <63> A.FRACT
x27 CALL @v25:PREN (x17.p2:A.EX) (x10.p1:T.EX) (x8.p1:T.FR)
p1 r18 <8> T.EXP
p2 r16 <63> T.FRACT
```

```
p3 r15 <8> A.EXP
x28 <r> (x4.p1:FP.I) (*.c11=2)
p1 * <1> MODE
x29 XOR:US (x17.p1:A.SI) (x12.p1:T.SI)
p1 * <1> *
x30 XOR:US (x28.p1:MODE) (x29.p1)
p1 * <1> *
x31 SELECT (x30.p1)
b1.x31 BRANCH [0]
x32 +:US (x17.p3:A.FR) (x27.p2:T.FR)
p1 * <64> *
x33 <r> (x32.p1) (*.c3=0)
p1 r13 <63> A.FRACT
x34 <r> (x32.p1) (*.c9=63)
p1 r19 <1> C
x35 SELECT (x34.p1:C)
b1.x35 BRANCH [TRUE]
x36 SR1:US (x33.p1:A.FR) (*.c1=1)
p1 r13 <63> A.FRACT
x37 +:US (x27.p3:A.EX) (*.c2=1)
p1 * <9> *
x38 <r> (x37.p1) (*.c3=0)
p1 r15 <8> A.EXP
b1.x35 ENDBR (x38.p1:A.EX) (x36.p1:A.FR)
b2.x35 BRANCH [0]
b2.x35 ENDBR (x27.p3:A.FR) (x33.p1:A.FR)
x35 ENDSEL
p1 r15 <8> A.EXP
p2 r13 <63> A.FRACT
b1.x31 ENDBR (x35.p2:A.FR) (x35.p1:A.EX) (x34.p1:C) (x17.p1:A.SI)
b2.x31 BRANCH [1]
x39 -:TC (x17.p3:A.FR) (x27.p2:T.FR)
p1 * <64> *
```

```
x40 <r> (x39.p1) (*.c3=0)
p1 r13 <63> A.FRACT
x41 <r> (x39.p1) (*.c9=63)
p1 r19 <1> C
x42 EQL:US (x40.p1:A.FR) (*.c8=0)
p1 * <1> *
x43 SELECT (x42.p1)
b1.x43 BRANCH [TRUE]
b1.x43 ENDBR (*.c3=0) (*.c12=0)
b2.x43 BRANCH [0]
b2.x43 ENDBR (x17.p1:A.SI) (x27.p3:A.EX)
x43 ENDSEL
p1 r14 <1> A.SIGN
p2 r15 <8> A.EXP
x45 NOT:US (x41.p1:C)
p1 * <1> *
x46 SELECT (x45.p1)
b1.x46 BRANCH [TRUE]
x47 --:TC (x40.p1:A.FR)
p1 * <64> *
x48 <r> (x47.p1) (*.c3=0)
p1 r13 <63> A.FRACT
x49 NOT:US (x43.p1:A.SI)
p1 r14 <1> A.SIGN
b1.x46 ENDBR (x49.p1:A.SI) (x48.p1:A.FR)
b2.x46 BRANCH [0]
b2.x46 ENDBR (x43.p1:A.SI) (x40.p1:A.FR)
x46 ENDSEL
p1 r14 <1> A.SIGN
p2 r13 <63> A.FRACT
x50 CALL @v28:NORM (x46.p2:A.FR) (x43.p2:A.EX) (x2.p1:WAIT)
(x4.p1:FP.I) (x27.p2:T.FR) (x27.p1:T.EX) (x12.p1:T.SI)
(x46.p1:A.SI) (x41.p1:C) (v24.i13:FP.E) (v24.i3:FP.R)
```

```
(v24.i5:EXP) (v24.i7:SIGN) (v24.i14:READ) (v24.i15:DATA)
(v24.i16:Z) (v24.i17:N) (v24.i1:GO)
p1 r13 <63> A.FRACT
p2 r15 <8> A.EXP
b2.x31 ENDBR (x50.p1:A.FR) (x50.p2:A.EX) (x41.p1:C) (x46.p1:A.SI)
x31 ENDSEL
p1 r13 <63> A.FRACT
p2 r15 <8> A.EXP
p3 r19 <1> C
p4 r14 <1> A.SIGN
b1.x6 ENDBR (x31.p4:A.SI) (x31.p3:C) (x31.p2:A.EX) (x31.p1:A.FR)
(x27.p2:T.FR) (x27.p1:T.EX) (x12.p1:T.SI) (v24.i13:FP.E)
(v24.i3:FP.R) (v24.i5:EXP) (v24.i7:SIGN) (v24.i15:DATA)
(x2.p1:WAIT) (v24.i14:READ)
b2.x6 BRANCH [1]
x51 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x52 [r] (v24.i3:FP.R) (x51.p1:REG)
p1 r16 <63> T.FRACT
x53 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x54 [r] (v24.i5:EXP) (x53.p1:REG)
p1 r16 <8> T.EXP
x55 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x56 [r] (v24.i7:SIGN) (x55.p1:REG)
p1 r17 <1> T.SIGN
x57 <r> (x4.p1:FP.I) (*.c11=2)
p1 * <1> MODE
x58 SELECT (x57.p1:MODE)
b1.x58 BRANCH [0]
x59 EQL:US (x52.p1:T.FR) (*.c8=0)
p1 * <1> *
```

```
x60 EQL:US (v24.i9:A.FR) (*.c8=0)
p1 * <1> *
x61 OR:US (x59.p1) (x60.p1)
p1 * <1> *
x62 SELECT (x61.p1)
b1.x62 BRANCH [TRUE]
b1.x62 ENDBR (*.c8=0)
b2.x62 BRANCH [0]
b2.x62 ENDBR (v24.i9:A.FR)
x62 ENDSEL
p1 r13 <63> A.FRACT
x64 XOR:US (v24.i11:A.SI) (x56.p1:T.SI)
p1 r14 <1> A.SIGN
x65 +:US (v24.i10:A.EX) (x54.p1:T.EX)
p1 * <9> *
x66 <r> (x65.p1) (*.c3=0)
p1 * <8> *
x67 -:TC (x66.p1) (*.c13=128)
p1 * <9> *
x68 <r> (x67.p1) (*.c3=0)
p1 r15 <8> A.EXP
x69 *:US (x62.p1:A.FR) (x52.p1:T.FR)
p1 * <126> *
x70 <r> (x69.p1) (*.c4=62)
p1 * <62> *
x71 PAD0:US (x70.p1)
p1 r13 <63> A.FRACT
x72 <r> (x71.p1:A.FR) (*.c4=62)
p1 * <1> A.FRACT
x73 EQL:US (x72.p1:A.FR) (*.c3=0)
p1 * <1> *
x74 SELECT (x73.p1)
b1.x74 BRANCH [TRUE]
```

```
x75 SLO:US (x71.p1:A.FR) (*.c1=1)
p1 r13 <63> A.FRACT
x76 -:TC (x68.p1:A.EX) (*.c2=1)
p1 * <9> *
x77 <r> (x76.p1) (*.c3=0)
p1 r15 <8> A.EXP
b1.x74 ENDBR (x77.p1:A.EX) (x75.p1:A.FR)
b2.x74 BRANCH [0]
b2.x74 ENDBR (x68.p1:A.EX) (x71.p1:A.FR)
x74 ENDSEL
p1 r15 <8> A.EXP
p2 r13 <63> A.FRACT
b1.x58 ENDBR (x74.p2:A.FR) (x74.p1:A.EX) (x64.p1:A.SI) (v24.i13:FP.E)
b2.x58 BRANCH [1]
x78 EQL:US (x52.p1:T.FR) (*.c8=0)
p1 * <1> *
x79 SELECT (x78.p1)
b1.x79 BRANCH [TRUE]
x80 <w> (v24.i13:FP.E) (*.c5=1) (*.c3=0)
p1 * <1> FP.ERROR
b1.x79 ENDBR (x80.p1:FP.E)
b2.x79 BRANCH [0]
b2.x79 ENDBR (v24.i13:FP.E)
x79 ENDSEL
p1 r4 <1> FP.ERROR
x82 EQL:US (v24.i9:A.FR) (*.c8=0)
p1 * <1> *
x83 SELECT (x82.p1)
b1.x83 BRANCH [TRUE]
b1.x83 ENDBR
b2.x83 BRANCH [0]
b2.x83 ENDBR
x83 ENDSEL
```

```
x85 XOR:US (v24.i11:A.SI) (x56.p1:T.SI)
p1 r14 <1> A.SIGN
x86 GEQ:US (v24.i9:A.FR) (x52.p1:T.FR)
p1 * <1> *
x87 SELECT (x86.p1)
b1.x87 BRANCH [TRUE]
x88 SR0:US (v24.i9:A.FR) (*.c1=1)
p1 r13 <63> A.FRACT
x89 +:US (v24.i10:A.EX) (*.c2=1)
p1 * <9> *
x90 <r> (x89.p1) (*.c3=0)
p1 r15 <8> A.EXP
b1.x87 ENDBR (x90.p1:A.EX) (x88.p1:A.FR)
b2.x87 BRANCH [0]
b2.x87 ENDBR (v24.i10:A.EX) (v24.i9:A.FR)
x87 ENDSEL
p1 r15 <8> A.EXP
p2 r13 <63> A.FRACT
x91 @ (x87.p2:A.FR) (*.c14=0)
p1 * <125> *
x92 /:US (x91.p1) (x52.p1:T.FR)
p1 * <125> *
x93 <r> (x92.p1) (*.c3=0)
p1 r13 <63> A.FRACT
x94 -:TC (x87.p1:A.EX) (x54.p1:T.EX)
p1 * <9> *
x95 <r> (x94.p1) (*.c3=0)
p1 * <8> *
x96 +:US (x95.p1) (*.c13=128)
p1 * <9> *
x97 <r> (x96.p1) (*.c3=0)
p1 r15 <8> A.EXP
b2.x58 ENDBR (x93.p1:A.FR) (x97.p1:A.EX) (x85.p1:A.SI)
```

```
(x79.p1:FP.E)
x58 ENDSEL
p1 r13 <63> A.FRACT
p2 r15 <8> A.EXP
p3 r14 <1> A.SIGN
p4 r4 <1> FP.ERROR
b2.x6 ENDBR (x58.p3:A.SI) (v24.i12:C) (x58.p2:A.EX)
(x58.p1:A.FR) (x52.p1:T.FR) (x54.p1:T.EX)
(x56.p1:T.SI) (x58.p4:FP.E) (v24.i3:FP.R)
(v24.i5:EXP) (v24.i7:SIGN) (v24.i15:DATA)
(x2.p1:WAIT) (v24.i14:READ)
b3.x6 BRANCH [2]
x98 <r> (x4.p1:FP.I) (*.c11=2)
p1 * <1> MODE
x99 SELECT (x98.p1:MODE)
b1.x99 BRANCH [0]
x100 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x101 [r] (v24.i3:FP.R) (x100.p1:REG)
p1 r13 <63> A.FRACT
x102 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x103 [r] (v24.i5:EXP) (x102.p1:REG)
p1 r15 <8> A.EXP
x104 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x105 [r] (v24.i7:SIGN) (x104.p1:REG)
p1 r14 <1> A.SIGN
b1.x99 ENDBR (x105.p1:A.SI) (x103.p1:A.EX) (x101.p1:A.FR)
(v24.i7:SIGN) (v24.i5:EXP) (v24.i3:FP.R)
b2.x99 BRANCH [1]
x106 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
```

```
x107 [w] (v24.i3:FP.R) (x106.p1:REG) (v24.i9:A.FR) (*.c3=0)
p1 r20 <63> FP.REG
x108 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x109 [w] (v24.i5:EXP) (x108.p1:REG) (v24.i10:A.EX) (*.c3=0)
p1 r21 <8> EXP.REG
x110 <r> (x4.p1:FP.I) (*.c7=0)
p1 * <2> REG.NUM
x111 [w] (v24.i7:SIGN) (x110.p1:REG) (v24.i11:A.SI) (*.c3=0)
p1 r22 <1> SIGN.REG
b2.x99 ENDBR (v24.i11:A.SI) (v24.i10:A.EX) (v24.i9:A.FR)
(x111.p1:SIGN) (x109.p1:EXP) (x107.p1:FP.R)
x99 ENDSEL
p1 r14 <1> A.SIGN
p2 r15 <8> A.EXP
p3 r13 <63> A.FRACT
p4 r22 <1> SIGN.REG
p5 r21 <8> EXP.REG
p6 r20 <63> FP.REG
b3.x6 ENDBR (x99.p1:A.SI) (v24.i12:C) (x99.p2:A.EX)
(x99.p3:A.FR) (v24.i4:T.FR) (v24.i6:T.EX)
(v24.i8:T.SI) (v24.i13:FP.E) (x99.p6:FP.R)
(x99.p5:EXP) (x99.p4:SIGN) (v24.i15:DATA)
(x2.p1:WAIT) (v24.i14:READ)
b4.x6 BRANCH [3]
x112 <r> (x4.p1:FP.I) (*.c11=2)
p1 * <1> MODE
x113 SELECT (x112.p1:MODE)
b1.x113 BRANCH [0]
x114 <r> (v24.i14:READ) (*.c3=0)
p1 r6 <1> READY
x115 WAIT (x114.p1:READ)
p1 r30 <140> WAIT
```

```
x116 <r> (v24.i15:DATA) (*.c3=0)
p1 r5 <16> DATA
x117 <r> (x116.p1:DATA) (*.c3=0)
p1 * <7> *
x118 <w> (v24.i9:A.FR) (x117.p1) (*.c15=56)
p1 r13 <63> A.FRACT
x119 <r> (x116.p1:DATA) (*.c16=7)
p1 r14 <1> A.SIGN
x120 <r> (x116.p1:DATA) (*.c17=8)
p1 r15 <8> A.EXP
x121 <w> (v24.i14:READ) (*.c3=0) (*.c3=0)
p1 * <1> READY
x122 <r> (x121.p1:READ) (*.c3=0)
p1 r6 <1> READY
x123 WAIT (x122.p1:READ)
p1 r30 <140> WAIT
x124 <r> (v24.i5:DATA) (*.c3=0)
p1 r5 <16> DATA
x125 <w> (x118.p1:A.FR) (x124.p1:DATA) (*.c18=40)
p1 r13 <63> A.FRACT
x126 <w> (x121.p1:READ) (*.c3=0) (*.c3=0)
p1 * <1> READY
x127 <r> (x126.p1:READ) (*.c3=0)
p1 r6 <1> READY
x128 WAIT (x127.p1:READ)
p1 r30 <140> WAIT
x129 <r> (v24.i15:DATA) (*.c3=0)
p1 r5 <16> DATA
x130 <w> (x125.p1:A.FR) (x129.p1:DATA) (*.c19=24)
p1 r13 <63> A.FRACT
x131 <w> (x126.p1:READ) (*.c3=0) (*.c3=0)
p1 r6 <1> READY
x132 <r> (x131.p1:READ) (*.c3=0)
```

```
p1 r6 <1> READY
x133 WAIT (x132.p1:READ)
p1 * <1> WAIT
x134 <r> (v24.i15:DATA) (*.c3=0)
p1 r5 <16> DATA
x135 <w> (x130.p1:A.FR) (x134.p1:DATA) (*.c17=8)
p1 r13 <63> A.FRACT
x136 <w> (x131.p1:READ) (*.c3=0) (*.c3=0)
p1 * <1> READY
x137 <w> (x135.p1:A.FR) (*.c12=0) (*.c3=0)
p1 * <63> A.FRACT
x138 CALL @v29:IN.N (x137.p1:A.FR) (x120.p1:A.EX)
(x4.p1:FP.I) (v24.i4:T.FR) (v24.i6:T.EX)
(v24.i8:T.SI) (x119.p1:A.SI) (v24.i12:C)
(v24.i13:FP.E) (v24.i3:FP.R) (v24.i5:EXP)
(v24.i7:SIGN) (x136.p1:READ) (v24.i15:DATA)
(v24.i16:Z) (v24.i17:N) (v24.i1:GO)
p1 r13 <63> A.FRACT
p2 r15 <8> A.EXP
b1.x113 ENDBR (x138.p2:A.EX) (x138.p1:A.FR) (x136.p1:READ)
(x133.p1:WAIT) (x119.p1:A.SI) (v24.i15:DATA)
b2.x113 BRANCH [1]
x139 <r> (v24.i14:READ) (*.c3=0)
p1 r6 <1> READY
x140 WAIT (x139.p1:READ)
p1 r30 <140> WAIT
x141 @ (v24.i10:A.EX) (v24.i11:A.SI)
p1 * <9> *
x142 <r> (v24.i9:A.FR) (*.c15=56)
p1 * <7> A.FRACT
x143 @ (x141.p1) (x142.p1:A.FR)
p1 * <16> *
x144 <w> (v24.i15:DATA) (x143.p1) (*.c3=0)
```

```
p1 r5 <16> DATA
x145 <w> (v24.i14:READ) (*.c3=0) (*.c3=0)
p1 * <1> READY
x146 <r> (x145.p1:READ) (*.c3=0)
p1 r6 <1> READY
x147 WAIT (x146.p1:READ)
p1 r30 <140> WAIT
x148 <r> (v24.i9:A.FR) (*.c18=40)
p1 r5 <16> A.FRACT
x149 <w> (x144.p1:DATA) (x148.p1:A.FR) (*.c3=0)
p1 r5 <16> DATA
x150 <w> (x145.p1:READ) (*.c3=0) (*.c3=0)
p1 * <1> READY
x151 <r> (x150.p1:READ) (*.c3=0)
p1 r6 <1> READY
x152 WAIT (x151.p1:READ)
p1 r30 <140> WAIT
x153 <r> (v24.i9:A.FR) (*.c20=24)
p1 * <16> A.FRACT
x154 <w> (x149.p1:DATA) (x153.p1:A.FR) (*.c3=0)
p1 r5 <16> DATA
x155 <w> (x150.p1:READ) (*.c3=0) (*.c3=0)
p1 * <1> READY
x156 <r> (x155.p1:READ) (*.c3=0)
p1 r6 <1> READY
x157 WAIT (x156.p1:READ)
p1 r30 <140> WAIT
x158 <r> (v24.i9:A.FR) (*.c21=8)
p1 * <16> A.FRACT
x159 <w> (x154.p1:DATA) (x158.p1:A.FR) (*.c3=0)
p1 r5 <16> DATA
x160 <w> (x155.p1:READ) (*.c3=0) (*.c3=0)
p1 * <1> READY
```

```
b2.x113 ENDBR (v24.i10:A.EX) (v24.i9:A.FR) (x160.p1:READ)
(x157.p1:WAIT)(v24.i11:A.SI) (x159.p1:DATA)
x113 ENDSEL
p1 r15 <8> A.EXP
p2 r13 <63> A.FRACT
p3 r6 <1> READY
p4 r30 <140> WAIT
p5 r14 <1> A.SIGN
p6 r5 <16> DATA
b4.x6 ENDBR (x113.p5:A.SI) (v24.i12:C) (x113.p1:A.EX)
(x113.p2:A.FR) (v24.i4:T.FR) (v24.i6:T.EX)
(v24.i8:T.SI) (v24.i13:FP.E) (v24.i3:FP.R)
(v24.i5:EXP) (v24.i7:SIGN) (x113.p6:DATA)
(x113.p4:WAIT) (x113.p3:READ)
x6 ENDSEL
p1 r14 <1> A.SIGN
p2 r19 <1> C
p3 r15 <8> A.EXP
p4 r13 <63> A.FRACT
p5 r16 <63> T.FRACT
p6 r18 <8> T.EXP
p7 r17 <1> T.SIGN
p8 r4 <1> FP.ERROR
p9 r20 <63> FP.REG
p10 r21 <8> EXP.REG
p11 r22 <1> SIGN.REG
p12 r5 <16> DATA
p13 r30 <140> WAIT
p14 r6 <1> READY
x161 EQL:US (x6.p4:A.FR) (*.c8=0)
p1 * <1> *
x162 <w> (v24.i16:Z) (x161.p1) (*.c3=0)
p1 r7 <1> Z
```

```
x163 <w> (v24.i17:N) (x6.p1:A.SI) (*.c3=0)
p1 r8 <1> N
x164 <w> (v24.i1:GO) (*.c3=0) (*.c3=0)
p1 * <1> GO
x165 LEAVE @v24:INST (x6.p5:T.FR) (x6.p6:T.EX) (x6.p7:T.SI)
(x6.p3:A.EX) (x6.p1:A.SI) (x6.p4:A.FR) (x6.p2:C)
(x6.p8:FP.E) (x6.p9:FP.R) (x6.p10:EXP) (x6.p11:SIGN)
(x6.p14:READ) (x6.p12:DATA) (x162.p1:Z) (x163.p1:N)
(x164.p1:GO)
o1 r16 <63> T.FRACT
o2 r18 <8> T.EXP
o3 r17 <1> T.SIGN
o4 r15 <8> A.EXP
o5 r14 <1> A.SIGN
o6 r13 <63> A.FRACT
o7 r19 <1> C
o8 r4 <1> FP.ERROR
o9 r20 <63> FP.REG
o10 r21 <8> EXP.REG
o11 r22 <1> SIGN.REG
o12 r6 <1> READY
o13 r5 <16> DATA
o14 r7 <1> Z
o15 r8 <1> N
o16 r2 <1> GO
----------

v23 %s1 001001000100010000-US * * START;
i1 r2 <1> GO
i2 r3 <6> INST.IN
i3 r20 <63> FP.REG
i4 r16 <63> T.FRACT
i5 r21 <8> EXP.REG
```

```
i6 r18 <8> T.EXP
i7 r22 <1> SIGN.REG
i8 r17 <1> T.SIGN
i9 r13 <63> A.FRACT
i10 r15 <8> A.EXP
i11 r14 <1> A.SIGN
i12 r19 <1> C
i13 r4 <1> FP.ERROR
i14 r6 <1> READY
i15 r5 <16> DATA
i16 r7 <1> Z
i17 r8 <1> N
x1 CALL @v24:INST (v23.i1:GO) (v23.i2:INST) (v23.i3:FP.R)
(v23.i4:T.FR) (v23.i5:EXP) (v23.i6:T.EX) (v23.i7:SIGN)
(v23.i8:T.SI) (v23.i9:A.FR) (v23.i10:A.EX) (v23.i11:A.SI)
(v23.i12:C) (v23.i13:FP.E) (v23.i14:READ) (v23.i15:DATA)
(v23.i16:Z) (v23.i17:N)
p1 r16 <63> T.FRACT
p2 r18 <8> T.EXP
p3 r17 <1> T.SIGN
p4 r15 <8> A.EXP
p5 r14 <1> A.SIGN
p6 r13 <63> A.FRACT
p7 r19 <1> C
p8 r4 <1> FP.ERROR
p9 r20 <63> FP.REG
p10 r21 <8> EXP.REG
p11 r22 <1> SIGN.REG
p12 r6 <1> READY
p13 r5 <16> DATA
p14 r7 <1> Z
p15 r8 <1> N
p16 r2 <1> GO
```

```
x2 LEAVE @v23:STAR (x1.p8:FP.E) (x1.p12:READY) (x1.p13:DATA)
(x1.p14:Z) (x1.p15:N) (x1.p16:GO)
o1 r4 <1> FP.ERROR
o2 r6 <1> READY
o3 r5 <16> DATA
o4 r7 <1> Z
o5 r8 <1> N
o6 r2 <1> GO
----------
```

G.2 Floating Point Coprocessor Park Normal Form Description

```
#
#  Description automatically generated by vt2pnf
#
nn114 dummy 0
nn117 dummy 0
nn118 dummy 0
nn123 dummy 0
nn124 dummy 0
nn127 dummy 0
nn157 dummy 0
nn159 dummy 0
nn164 dummy 0
nn165 dummy 0
nn168 dummy 0
nn169 dummy 0
nn174 dummy 0
nn179 dummy 0
nn182 dummy 0
```

```
nn185 dummy 0
nn187 dummy 0
nn19 dummy 0
nn191 dummy 0
nn221 dummy 0
nn34 dummy 0
nn50 dummy 0
nn51 dummy 0
nn57 dummy 0
nn62 dummy 0
nn64 dummy 0
outport dummy 0
root dummy 0
v23_225 dummy 63
v23_226 dummy 63
v23_227 dummy 8
v23_228 dummy 8
v23_229 dummy 1
v23_230 dummy 1
v23_231 dummy 63
v23_232 dummy 8
v23_233 dummy 1
v23_234 dummy 1
v23_235 dummy 1
v23_236 dummy 1
v23_237 dummy 16
v23_240 bit-read 1
v23_241 dummy 1
v23_242 bit-read 6
v23_243 bit-read 6
v23_244 bit-read 6
v23_245 dist 0
v23_246 join 0
```

```
v23_249 bit-read 6
v23_250 read 63
v23_251 bit-read 6
v23_252 read 8
v23_253 bit-read 6
v23_254 read 6
v23_255 cmp 63
v23_256 dist 0
v23_257 join 0
v23_262 cmp 63
v23_263 dist 0
v23_264 join 0
v23_267 concat 8
v23_268 concat 63
v23_269 bit-read 63
v23_270 bit-read 63
v23_271 bit-read 63
v23_272 bit-read 6
v23_273 dist 0
v23_274 join 0
v23_277 inv 63
v23_282 dummy 8
v23_283 dummy 8
v23_285 cmp 8
v23_286 dist 0
v23_287 join 0
v23_290 cmp 8
v23_291 dist 0
v23_292 join 0
v23_298 add 8
v23_299 bit-read 8
v23_300 r-shift 63
v23_301 cmp 8
```

```
v23_302 dist 0
v23_303 join 0
v23_315 add 8
v23_316 bit-read 8
v23_317 r-shift 63
v23_318 cmp 8
v23_319 dist 0
v23_320 join 0
v23_329 dummy 8
v23_330 dummy 63
v23_332 bit-read 6
v23_333 xor 6
v23_334 xor 6
v23_335 dist 0
v23_336 join 0
v23_339 add 63
v23_340 bit-read 63
v23_341 bit-read 63
v23_342 dist 0
v23_343 join 0
v23_346 r-shift1 63
v23_347 add 8
v23_348 bit-read 8
v23_353 sub 63
v23_354 bit-read 63
v23_355 bit-read 63
v23_356 cmp 63
v23_357 dist 0
v23_358 join 0
v23_363 inv 63
v23_364 dist 0
v23_365 join 0
v23_368 neg 63
```

```
v23_369 bit-read 63
v23_370 inv 0
v23_391 bit-read 63
v23_392 cmp 63
v23_393 dist 0
v23_394 join 0
v23_399 l-shift 63
v23_400 sub 8
v23_401 bit-read 8
v23_406 bit-read 6
v23_407 read 63
v23_408 bit-read 6
v23_409 read 8
v23_410 bit-read 6
v23_411 read 6
v23_412 bit-read 6
v23_413 dist 0
v23_414 join 0
v23_417 cmp 63
v23_418 cmp 63
v23_419 or 63
v23_420 dist 0
v23_421 join 0
v23_426 xor 6
v23_427 add 8
v23_428 bit-read 8
v23_429 sub 8
v23_430 bit-read 8
v23_431 mul 63
v23_432 bit-read 63
v23_434 bit-read 63
v23_435 cmp 63
v23_436 dist 0
```

```
v23_437 join 0
v23_440 l-shift 63
v23_441 sub 8
v23_442 bit-read 8
v23_447 cmp 63
v23_448 dist 0
v23_449 join 0
v23_452 bit-write 1
v23_455 cmp 63
v23_456 dist 0
v23_457 join 0
v23_462 xor 6
v23_463 cmp 63
v23_464 dist 0
v23_465 join 0
v23_468 r-shift 63
v23_469 add 8
v23_470 bit-read 8
v23_473 concat 0
v23_474 div 63
v23_475 bit-read 63
v23_476 sub 8
v23_477 bit-read 8
v23_478 add 8
v23_479 bit-read 8
v23_482 bit-read 6
v23_483 dist 0
v23_484 join 0
v23_487 bit-read 6
v23_488 read 63
v23_489 bit-read 6
v23_490 read 8
v23_491 bit-read 6
```

```
v23_492 read 6
v23_495 bit-read 6
v23_496 write 63
v23_497 bit-read 6
v23_498 write 8
v23_499 bit-read 6
v23_500 write 6
v23_503 bit-read 6
v23_504 dist 0
v23_505 join 0
v23_508 bit-read 1
v23_510 bit-read 16
v23_511 bit-read 16
v23_512 bit-write 63
v23_513 bit-read 16
v23_514 bit-read 16
v23_515 bit-write 1
v23_516 bit-read 1
v23_518 bit-read 8
v23_519 bit-write 63
v23_520 bit-write 1
v23_521 bit-read 1
v23_523 bit-read 16
v23_524 bit-write 63
v23_525 bit-write 1
v23_526 bit-read 1
v23_528 bit-read 16
v23_529 bit-write 63
v23_530 bit-write 1
v23_531 bit-write 63
v23_549 bit-read 63
v23_550 cmp 63
v23_551 dist 0
```

```
v23_552 join 0
v23_557 l-shift 63
v23_558 sub 8
v23_559 bit-read 8
v23_564 bit-read 1
v23_566 concat 8
v23_567 bit-read 63
v23_568 concat 63
v23_569 bit-write 63
v23_570 bit-write 1
v23_571 bit-read 1
v23_573 bit-read 63
v23_574 bit-write 63
v23_575 bit-write 1
v23_576 bit-read 1
v23_578 bit-read 63
v23_579 bit-write 63
v23_580 bit-write 1
v23_581 bit-read 1
v23_583 bit-read 63
v23_584 bit-write 63
v23_585 bit-write 1
v23_586 cmp 0
v23_587 bit-write 1
v23_588 bit-write 1
v23_589 bit-write 1
v23_i1 dummy 1
v23_i16 dummy 1
v23_i17 dummy 1

v23_240 v23_241 1
v23_242 v23_243 6
v23_243 v23_244 6
```

```
v23_244 v23_245 6
v23_245 nn51 0
v23_243 v23_249 6
v23_225 v23_250 63
v23_249 v23_250 6
v23_243 v23_251 6
v23_227 v23_252 8
v23_251 v23_252 6
v23_243 v23_253 6
v23_229 v23_254 1
v23_253 v23_254 6
v23_250 v23_255 63
v23_255 v23_256 63
v23_231 v23_262 63
v23_262 v23_263 63
v23_263 nn57 0
v23_252 v23_267 8
v23_254 v23_267 6
v23_267 v23_268 8
v23_250 v23_268 63
v23_268 v23_269 63
v23_268 v23_270 63
v23_268 v23_271 63
v23_243 v23_272 6
v23_272 v23_273 6
v23_270 v23_277 63
nn62 v23_274 0
v23_270 nn62 63
nn57 v23_267 0
nn57 v23_272 0
nn64 v23_264 0
v23_233 nn64 1
v23_232 nn64 8
```

```
v23_231 nn64 63
v23_282 v23_285 8
v23_283 v23_285 8
v23_285 v23_286 8
v23_282 v23_290 8
v23_283 v23_290 8
v23_290 v23_291 8
v23_291 nn19 0
v23_298 v23_299 8
v23_299 v23_301 8
v23_301 v23_302 8
v23_291 nn34 0
v23_315 v23_316 8
v23_316 v23_318 8
v23_318 v23_319 8
nn50 v23_287 0
v23_283 nn50 8
v23_282 nn50 8
v23_287 v23_329 0
v23_287 v23_330 0
v23_264 v23_282 8
v23_252 v23_283 8
v23_243 v23_332 6
v23_264 v23_333 0
v23_254 v23_333 6
v23_332 v23_334 6
v23_333 v23_334 6
v23_334 v23_335 6
v23_264 v23_339 0
v23_330 v23_339 63
v23_339 v23_340 63
v23_339 v23_341 63
v23_341 v23_342 63
```

```
v23_342 nn114 0
v23_340 v23_346 63
v23_347 v23_348 8
nn114 v23_346 0
nn114 v23_347 0
nn117 v23_343 0
v23_340 nn117 63
v23_335 nn118 0
v23_264 v23_353 0
v23_330 v23_353 63
v23_353 v23_354 63
v23_353 v23_355 63
v23_354 v23_356 63
v23_356 v23_357 63
nn123 v23_358 0
v23_264 nn123 0
v23_355 v23_363 63
v23_363 v23_364 63
v23_364 nn124 0
v23_354 v23_368 63
v23_368 v23_369 63
v23_358 v23_370 0
nn124 v23_368 0
nn124 v23_370 0
nn127 v23_365 0
v23_358 nn127 0
v23_354 nn127 63
v23_391 v23_392 63
v23_392 v23_393 63
v23_400 v23_401 8
nn118 v23_353 0
nn51 v23_249 0
nn51 v23_251 0
```

```
nn51 v23_253 0
nn51 v23_262 0
nn51 v23_332 0
v23_245 nn157 0
v23_243 v23_406 6
v23_225 v23_407 63
v23_406 v23_407 6
v23_243 v23_408 6
v23_227 v23_409 8
v23_408 v23_409 6
v23_243 v23_410 6
v23_229 v23_411 1
v23_410 v23_411 6
v23_243 v23_412 6
v23_412 v23_413 6
v23_413 nn159 0
v23_407 v23_417 63
v23_231 v23_418 63
v23_417 v23_419 63
v23_418 v23_419 63
v23_419 v23_420 63
nn164 v23_421 0
v23_231 nn164 63
v23_233 v23_426 1
v23_411 v23_426 6
v23_232 v23_427 8
v23_409 v23_427 8
v23_427 v23_428 8
v23_428 v23_429 8
v23_429 v23_430 8
v23_421 v23_431 0
v23_407 v23_431 63
v23_431 v23_432 63
```

```
v23_434 v23_435 63
v23_435 v23_436 63
v23_436 nn165 0
v23_430 v23_441 8
v23_441 v23_442 8
nn165 v23_440 0
nn165 v23_441 0
nn168 v23_437 0
v23_430 nn168 8
nn159 v23_417 0
nn159 v23_418 0
nn159 v23_426 0
nn159 v23_427 0
v23_413 nn169 0
v23_407 v23_447 63
v23_447 v23_448 63
v23_235 v23_452 1
nn174 v23_449 0
v23_235 nn174 1
v23_231 v23_455 63
v23_455 v23_456 63
v23_233 v23_462 1
v23_411 v23_462 6
v23_231 v23_463 63
v23_407 v23_463 63
v23_463 v23_464 63
v23_464 nn179 0
v23_231 v23_468 63
v23_232 v23_469 8
v23_469 v23_470 8
nn179 v23_468 0
nn179 v23_469 0
nn182 v23_465 0
```

nn185 v23_491 0

v23_483 nn187 0

v23_243 v23_495 6

v23_225 v23_496 63

v23_495 v23_496 6

v23_231 v23_496 63

v23_243 v23_497 6

v23_227 v23_498 8

v23_497 v23_498 6

v23_232 v23_498 8

v23_243 v23_499 6

v23_229 v23_500 1

v23_499 v23_500 6

v23_233 v23_500 1

nn187 v23_495 0

nn187 v23_497 0

nn187 v23_499 0

v23_243 v23_503 6

v23_503 v23_504 6

v23_504 nn191 0

v23_236 v23_508 1

v23_237 v23_510 16

v23_510 v23_511 16

v23_231 v23_512 63

v23_511 v23_512 16

v23_510 v23_513 16

v23_510 v23_514 16

v23_236 v23_515 1

v23_515 v23_516 1

v23_227 v23_518 8

v23_512 v23_519 63

v23_518 v23_519 8

v23_515 v23_520 1

```
v23_520 v23_521 1
v23_237 v23_523 16
v23_519 v23_524 63
v23_523 v23_524 16
v23_520 v23_525 1
v23_525 v23_526 1
v23_237 v23_528 16
v23_524 v23_529 63
v23_528 v23_529 16
v23_525 v23_530 1
v23_529 v23_531 63
v23_549 v23_550 63
v23_550 v23_551 63
v23_558 v23_559 8
nn191 v23_508 0
nn191 v23_510 0
nn191 v23_515 0
nn191 v23_518 0
nn191 v23_523 0
nn191 v23_528 0
v23_504 nn221 0
v23_236 v23_564 1
v23_232 v23_566 8
v23_233 v23_566 1
v23_231 v23_567 63
v23_566 v23_568 8
v23_567 v23_568 63
v23_237 v23_569 16
v23_568 v23_569 63
v23_236 v23_570 1
v23_570 v23_571 1
v23_231 v23_573 63
v23_569 v23_574 63
```

```
v23_573 v23_574 63
v23_570 v23_575 1
v23_575 v23_576 1
v23_231 v23_578 63
v23_574 v23_579 63
v23_578 v23_579 63
v23_575 v23_580 1
v23_580 v23_581 1
v23_231 v23_583 63
v23_579 v23_584 63
v23_583 v23_584 63
v23_580 v23_585 1
nn221 v23_564 0
nn221 v23_566 0
nn221 v23_567 0
nn221 v23_570 0
nn221 v23_573 0
nn221 v23_578 0
nn221 v23_583 0
v23_246 v23_586 0
v23_586 v23_587 0
v23_246 v23_588 0
root v23_i1 1
root v23_i16 1
root v23_i17 1
v23_335 v23_339 0
v23_343 v23_336 0
v23_343 v23_336 0
v23_341 v23_336 63
v23_264 v23_336 0
v23_348 v23_343 8
v23_346 v23_343 63
v23_342 nn117 0
```

```
v23_355 v23_336 63
v23_365 v23_336 0
v23_393 v23_394 0
v23_357 v23_358 0
v23_357 nn123 0
v23_370 v23_365 0
v23_369 v23_365 63
v23_364 nn127 0
v23_551 v23_552 0
v23_414 v23_246 0
v23_234 v23_246 1
v23_414 v23_246 0
v23_414 v23_246 0
v23_407 v23_246 63
v23_409 v23_246 8
v23_411 v23_246 6
v23_414 v23_246 0
v23_225 v23_246 63
v23_227 v23_246 8
v23_229 v23_246 1
v23_237 v23_246 16
v23_241 v23_246 1
v23_236 v23_246 1
v23_551 v23_552 0
v23_437 v23_414 0
v23_437 v23_414 0
v23_426 v23_414 6
v23_235 v23_414 1
v23_420 v23_421 0
v23_420 nn164 0
v23_442 v23_437 8
v23_440 v23_437 63
v23_436 nn168 0
```

```
v23_286 v23_290 0
v23_475 v23_414 63
v23_479 v23_414 8
v23_462 v23_414 6
v23_449 v23_414 0
v23_457 v23_414 0
v23_448 v23_452 0
v23_452 v23_449 1
v23_448 nn174 0
v23_456 v23_457 0
v23_456 v23_457 0
v23_292 v23_287 0
v23_292 v23_287 0
v23_292 v23_287 0
v23_470 v23_465 8
v23_468 v23_465 63
v23_464 nn182 0
v23_245 v23_482 0
v23_484 v23_246 0
v23_234 v23_246 1
v23_484 v23_246 0
v23_484 v23_246 0
v23_226 v23_246 63
v23_228 v23_246 8
v23_230 v23_246 1
v23_235 v23_246 1
v23_484 v23_246 0
v23_484 v23_246 0
v23_484 v23_246 0
v23_237 v23_246 16
v23_241 v23_246 1
v23_236 v23_246 1
v23_492 v23_484 6
```

```
v23_490 v23_484 8
v23_488 v23_484 63
v23_229 v23_484 1
v23_227 v23_484 8
v23_225 v23_484 63
v23_233 v23_484 1
v23_232 v23_484 8
v23_231 v23_484 63
v23_500 v23_484 6
v23_498 v23_484 8
v23_496 v23_484 63
v23_245 v23_503 0
v23_505 v23_246 0
v23_234 v23_246 1
v23_505 v23_246 0
v23_505 v23_246 0
v23_226 v23_246 63
v23_228 v23_246 8
v23_230 v23_246 1
v23_235 v23_246 1
v23_225 v23_246 63
v23_227 v23_246 8
v23_229 v23_246 1
v23_505 v23_246 0
v23_505 v23_246 0
v23_505 v23_246 0
v23_530 v23_505 1
v23_513 v23_505 16
v23_237 v23_505 16
v23_302 v23_303 0
v23_283 v23_292 8
v23_232 v23_505 8
v23_231 v23_505 63
```

```
v23_585 v23_505 1
v23_233 v23_505 1
v23_584 v23_505 63
v23_282 v23_292 8
v23_302 v23_303 0
v23_286 nn50 0
v23_336 v23_246 0
v23_336 v23_246 0
v23_336 v23_246 0
v23_336 v23_246 0
v23_330 v23_246 63
v23_329 v23_246 8
v23_254 v23_246 6
v23_235 v23_246 1
v23_225 v23_246 63
v23_227 v23_246 8
v23_229 v23_246 1
v23_237 v23_246 16
v23_241 v23_246 1
v23_236 v23_246 1
v23_257 v23_246 0
v23_256 v23_257 0
v23_256 v23_257 0
v23_274 v23_264 0
v23_271 v23_264 63
v23_269 v23_264 63
v23_273 v23_277 0
v23_319 v23_320 0
v23_277 v23_274 63
v23_273 nn62 0
v23_263 nn64 0
v23_319 v23_320 0
v23_393 v23_394 0
```

```
v23_i1 v23_240 1
v23_i1 v23_589 1
v23_i16 v23_587 1
v23_i17 v23_588 1
v23_250 nn50 63
v23_282 v23_298 8
nn19 v23_298 0
nn19 v23_300 0
v23_250 v23_300 63
v23_283 v23_301 8
nn19 v23_301 0
v23_299 v23_292 8
v23_303 v23_292 0
v23_300 v23_292 63
v23_283 v23_315 8
nn34 v23_315 0
nn34 v23_317 0
v23_250 v23_317 63
v23_282 v23_318 8
nn34 v23_318 0
v23_316 v23_292 8
v23_320 v23_292 0
v23_317 v23_292 63
v23_287 v23_347 0
v23_287 nn117 0
v23_287 nn123 0
v23_365 v23_391 63
v23_365 v23_399 63
v23_358 v23_400 8
v23_399 v23_336 63
v23_394 v23_336 0
v23_241 v23_336 140
nn118 v23_336 0
```

```
v23_243 v23_336 5
nn118 v23_336 0
v23_330 v23_336 63
nn118 v23_336 0
v23_329 v23_336 8
nn118 v23_336 0
v23_254 v23_336 1
nn118 v23_336 0
v23_365 v23_336 1
v23_355 v23_336 1
v23_235 v23_336 1
nn118 v23_336 0
v23_225 v23_336 63
nn118 v23_336 0
v23_227 v23_336 8
nn118 v23_336 0
v23_229 v23_336 1
nn118 v23_336 0
v23_236 v23_336 1
nn118 v23_336 0
v23_237 v23_336 16
nn118 v23_336 0
nn118 v23_336 0
v23_i16 v23_336 1
nn118 v23_336 0
v23_i17 v23_336 1
nn118 v23_336 0
v23_i1 v23_336 1
v23_401 v23_336 8
v23_432 v23_434 63
v23_432 v23_440 63
v23_432 nn168 63
v23_508 v23_505 1
```

```
v23_516 v23_505 1
v23_521 v23_505 1
v23_526 v23_505 1
v23_531 v23_549 63
v23_531 v23_557 63
v23_514 v23_558 8
v23_557 v23_505 63
v23_552 v23_505 0
v23_243 v23_505 5
nn191 v23_505 0
v23_226 v23_505 63
nn191 v23_505 0
v23_228 v23_505 8
nn191 v23_505 0
v23_230 v23_505 1
nn191 v23_505 0
v23_513 v23_505 1
v23_234 v23_505 1
nn191 v23_505 0
v23_235 v23_505 1
nn191 v23_505 0
v23_225 v23_505 63
nn191 v23_505 0
v23_227 v23_505 8
nn191 v23_505 0
v23_229 v23_505 1
nn191 v23_505 0
v23_530 v23_505 1
v23_237 v23_505 16
nn191 v23_505 0
nn191 v23_505 0
v23_i16 v23_505 1
nn191 v23_505 0
```

```
v23_i17 v23_505 1
nn191 v23_505 0
v23_i1 v23_505 1
v23_559 v23_505 8
v23_564 v23_505 1
v23_571 v23_505 1
v23_576 v23_505 1
v23_581 v23_505 1
root v23_232 8
root v23_233 1
root v23_234 1
root v23_235 1
root v23_236 1
root v23_237 16
root v23_242 6
root v23_225 63
root v23_226 63
root v23_227 8
root v23_228 8
root v23_229 1
root v23_230 1
root v23_231 63
v23_246 outport 0
v23_246 outport 0
v23_246 outport 0
v23_246 outport 0
v23_246 outport 0
v23_587 outport 1
v23_588 outport 1
v23_589 outport 1
v23_246 outport 0
v23_246 outport 0
v23_246 outport 0
```

```
v23_246 outport 0
v23_246 outport 0
v23_246 outport 0
v23_246 outport 0
v23_246 outport 0
```