UNIVERSITY OF CALIFORNIA
Santa Barbara

Symbolic Data Path Analysis

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Electrical and Computer Engineering

by

Chuck Monahan

Committee in charge:

Professor Forrest D. Brewer, Chairman

Professor Malgorzata Marek-Sadowska

Professor P. Michael Melliar-Smith

Doctor Mario Nemirovsky

June 1997

The dissertation of Chuck Monahan
is approved:

Committee Chairman

June 13, 1997

June, 1997

Copyright © 1997

Chuck Monahan

All Rights Reserved

To my father.

Acknowledgments

First, I would like to thank my advisor, Professor Forrest Brewer, for his guidance and support throughout my graduate studies at University of California, Santa Barbara. While my notebooks contain entries which questioned the sanity of his ideas, time has shown that the problem stemmed not from his vision but from mine.

Also, I would like to thank the Committee members: Professor Margaret Marek-Sadowska, Professor Michael Melliar-Smith, and Dr. Mario Nemirovsky for helpful suggestions and comments helping to improve the presentation of this work.

I would like to gratefully acknowledge contributions from Dr. Andrew Seawright and Dr. Ivan Radivojević both of whom added invaluable insights to this topic and whom demonstrated that a Ph.D. in CAD was a feasible and interesting proposition. A note of deep gratitude goes out to all of those that have helped maintain and improve our C++ BDD package, HomeBrew, used extensively throughout this project. Notable among this group are Dr. Andrew Seawright and Andy Crews, and Anthony Stornetta. An additional note of appreciation goes to all of these individuals, Hien Ha, and the other occupants of room 2164 who have placed knowledge and enjoyment ahead of competition and politics.

This work was sponsored by donations from the Mentor Graphics Corporation as well as UC-MICRO program. Without their generous support and willingness to help academic research, this work would have never been realized.

I want to use this opportunity to express my thanks, one more time, to all of my teachers and colleagues at: University of California of Santa Barbara, for their contributions to my knowledge and enthusiasm over the twelve year period.

Finally, my deepest gratitude goes to my parents, Bernard and Peggy, who instilled the belief in me that I had skills to develop yet granted me the flexibility to nurture them. My support and love is extended to my sister, Joey, who always extended them in return. And a debt of gratitude, whose only rival is the national debt, is owed to Monette Stephens who convinced me to not only start but to finish great things.

VITA

Born — Pleasanton, California, U.S.A. — April 8.1967.

EDUCATION

M. S. Electrical Engineering, 1993.
Department of Electrical and Computer Engineering
University of California at Santa Barbara
Santa Barbara, CA, U.S.A.

B. S. Electrical Engineering, 1990.
Department of Electrical and Computer Engineering
University of California at Santa Barbara
Santa Barbara, CA, U.S.A.

FIELDS OF STUDY

Major Field: Computer Engineering

Specialization: System Level Computer-Aided Design
Professor Forrest Brewer

Minor Field: Computer Science

PROFESSIONAL EXPERENCE

Graduate Student Researcher, Department of Electrical and Computer Engineering, University of California, Santa Barbara — September 1993.

Consultant/Owner, Monahan Consulting, Santa Barbara, CA— November 1993.

Teaching Assistant, Department of Electrical and Computer Engineering, University of California, Santa Barbara — September 1990.

PUBLICATIONS

Conference papers:

C. Monahan and F. Brewer, "Scheduling and Binding Bounds for RT-Level Symbolic Execution", *Proc. IEEE Int. Conf. Computer-Aided Design*, San Jose, CA. Nov. 1997.

C. Monahan and F. Brewer, “Concurrent Analysis Techniques for Data Path Timing Optimization”, *33rd IEEE/ACM Design Automation Conference Proceedings*, Las Vegas, NV, June 1996.

C. Monahan and F. Brewer, “Symbolic Modeling and Evaluation of Data Paths”, *32nd IEEE/ACM Design Automation Conference Proceedings*, San Francisco, CA, June 1995.

C. Monahan and F. Brewer, “Symbolic Execution of Data Paths”, *Proceedings of 5th Great Lakes Symposium on VLSI*, Buffalo, NY, March 1995.

C. Monahan and F. Brewer, “Communication Driven Interconnection Synthesis”, *Proceedings of 6th International Workshop on High Level Synthesis*, Dana Point, CA, November. 1992.

Symbolic Data Path Analysis

by

Chuck Monahan

ABSTRACT

In ASIC construction, design changes can occur at all phases of the product development cycle. When changes occur late in the development cycle, say after data-path synthesis and verification, it can be very expensive not to maintain a significant portion of the pre-existing design. However, changes in this environment require accommodation of the limitations of the pre-existing data-path, which potentially restrict operand movement, operand storage, or control encoding.Required changes may be in the data-path structure, in the input data-flow specification or may simply be attempts to remove critical communications which are limiting the performance. A variety of problems arise from these consideration including optimal memory operand binding, optimal function unit and communication binding, and optimal data-path constrained scheduling.

This thesis presents an automata model with which to systematically explore the mapping freedom between a data-flow graph and a pre-defined data path. This technique shows great potential for accommodating last minute design changes or

creating schedules around a core structure. Our model correctly represents the limited storage capacity, restricted communications structure, and restricted control vector constraints of a real data path and can accommodate a variety of user specified constraints. An exact symbolic formulation of these constraints and of the data-path-constrained operand movement are used to ensure correctness and potentially generate optimal mappings. The systematic approach identifies all solutions which comply with these constraints and minimize the number of cycles. Various optimizations and practical heuristics are presented for both the automata and its state encoding. The automata is implemented in a compressed binary decision diagram (BDD) representation to increase the efficiency of the automata execution.

Keywords: Binary Decision Diagrams; Data Paths; High-Level Synthesis; Scheduling; Retargetable Compilers; Re-binding; Interconnection; Timing; False Paths.

Contents

Chapter 1. Introduction 1

1.1 An Example 1

1.2 The Role of Change in the Design Process 4

1.2.1 Background 4

1.2.2 Examples of change 6

1.3 Accommodating Late Changes 8

1.3.1 Mapping data-flow graphs 10

1.3.2 Evaluating performance 13

Chapter 2. Related Work 15

2.1 Synthesis Methodology 15

2.2 Compiler Methodology 20

Chapter 3. Problem Formulation 23

3.1 Modeling Data-Path Activity 24

3.2 Competing Network Topologies 27

3.3 Data-flow Alternatives 29

3.4 Incorporation of Partial Data-Flow Map 32

3.5 Benefits of Operand Modeling 32

Chapter 4. Automata Representation 34

4.1 Input Specification 34

4.1.1 Data path 35

4.1.2 Data-flow graph 40

4.2 Problem Specification 42

4.3 Representing Data Paths 43

4.3.1 Automata model 43

4.3.2 Applying the automata 46

4.4 Transform Relation 50
4.4.1 Relation construction 50
4.4.2 Memory mapping optimizations 55
4.4.3 Operand lifetime optimizations 57
4.4.4 State reduction techniques 59

4.5 Encoding 60
4.5.1 Selected codes 61
4.5.2 Re-evaluating latch transform relations 62
4.5.3 Register size constraints.. 64

4.6 Additional Restrictions 65
4.6.1 Control restrictions 66
4.6.2 Data-flow restrictions 67

Chapter 5. Single Topology Applications 68

5.1 Data-Path Routing 69
5.1.1 Memory binding optimization 70
5.1.2 Routing Results.. 71

5.2 Data-Path Binding 75
5.2.1 Converting operation schedules into bounds 77
5.2.2 Binding Results.. 79

5.3 Data-Path Scheduling 80
5.3.1 ALAP bound generation 81
5.3.2 Scheduling Results.. 84

Chapter 6. Evaluating Multiple Networks 92

6.1 Scheduling on Multiple Data Paths 92

6.2 Timing Evaluation 94
6.2.1 Timing model... 95
6.2.2 Using the automata 102
6.2.3 Experimental Results 108

Chapter 7. Discussion 115

7.1 Summary 115

7.2 Future Research Lines 116
7.2.1 Better lifetime bounds.. 116
7.2.2 Cyclic data-flow graphs 117
7.2.3 Control data-flow graphs 118

Bibliography 119

Appendix A. Binary Decision Diagrams 126

Glossary 130

List of Figures

Figure 1.1:	Example of performance trade-offs.	2
Figure 1.2:	Idealized High-Level Synthesis Methodology	5
Figure 1.3:	System overview	11
Figure 1.4:	Scheduling example.	12
Figure 3.1:	Data-path activity.	24
Figure 3.2:	Pipelined ALU modeled as a compound component.	26
Figure 3.3:	Disjoint topology alterations.	27
Figure 3.4:	Merging alternative topologies into a single data-path design.	29
Figure 3.5:	Example data-flow graph	29
Figure 3.6:	Representing alternative operations.	31
Figure 4.1:	Base component set.	37
Figure 4.2:	Representing loadable register with base components.	38
Figure 4.3:	Dedicated latch	56
Figure 4.4:	Dedicated control line example	63
Figure 5.1:	TMS32020 based data-path models	72
Figure 5.2:	Dual register data path	73
Figure 5.3:	Novel data-flow graph benchmarks.	74
Figure 5.4:	Fluctuating ALAP bounds due to operand fanout.	84
Figure 5.5:	Cycle by cycle comparison of performance	88
Figure 5.6:	Resulting schedule and operand mappings.	90
Figure 6.1:	Wire Delay Model	98
Figure 6.2:	Latch's output wire captures connection and functional behavior.	99
Figure 6.3:	Multiplexer component variation: "switching element set"	100
Figure 6.4:	Partitioned time line.	103
Figure 6.5:	TMS32010 based data-path model and floorplans	109
Figure 6.6:	Timing analysis overhead in routing.	111
Figure 6.7:	Timing analysis overhead in binding.	113
Figure 7.1:	ROBDD forms of f=AB+C using different orderings	127

List of Tables

Table 2.1: Methodology Classification 16
Table 4.1: Behavioral Constraints 37
Table 5.1: Initial and Final operand bindings 75
Table 5.2: Data-Path Routing Results 76
Table 5.3: Data-Path Binding Results 80
Table 5.4: Exact scheduling results 86
Table 5.5: Heuristic schedules results 89
Table 6.1: Five combined topology benchmarks. 93
Table 6.2: Relative efficiency of multiple topology analysis. 93
Table 6.3: Physical Parameters 109

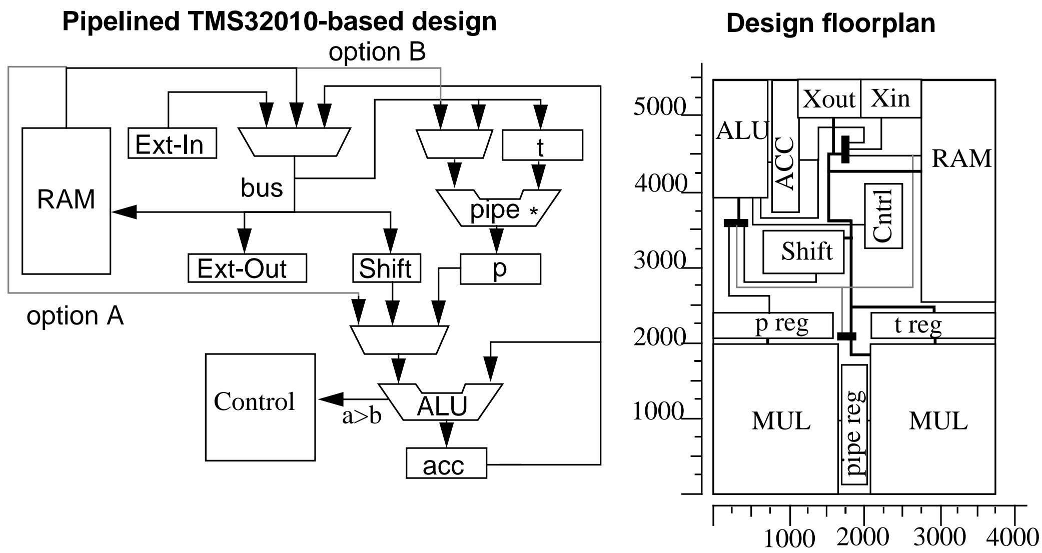
Chapter 1

Introduction

Integrated chip design and fabrication has become a fiercely competitive market. The reduction of feature sizes in combination with increased die sizes have dramatically increased the complexity of most chip designs. This complexity is in conflict with the factor that increasingly distinguishes a product: time to market. The realization that problems are inevitable in such a complex environment has resulted in significant improvements in integrated chip/digital system verification. While developments in automated synthesis have reduced the number of errors, the continued interest in verification underscores the belief that errors still exist. Equally important to the detection of errors is the correction of errors especially when these corrections can be made at low cost. This thesis proposes a number of techniques which aid the accommodation of changes, errors, or other forms of modifications that occur late in the design cycle.

1.1 An Example

Assume a designer wants to implement the differential equation (diff_eq) benchmark in a relatively short time frame. After considering the various options, they decide to utilize the pre-existing DSP core that is shown in Figure 1.1 with its



Critical Path Optimizations		(critical path)
Orig:	17 cycles * 37.4 ns = 635.8 ns	RAM -> Bus -> Shifter -> ALU -> Control
Opt A:	17 cycles * 33.4 ns = 567.8 ns	RAM -> Bus -> Multiplier -> pipe register
Opt A&B:	17 cycles * 33.1 ns = 562.7 ns	RAM -> option A -> ALU -> Control

Rescheduling Optimizations	
Orig:	17 cycles * 37.4 ns = 635.8 ns
Opt A:	16 cycles * 33.4 ns = 534.4 ns
Opt A&B:	14 cycles * 33.1 ns = 463.4 ns

Rescheduling for Timing	
Opt A&B:	14 cycles * 32.7 ns = 457.8 ns
Opt B:	14 cycles * 32.2 ns = 450.8 ns

Figure 1.1 Example of performance trade-offs.

accompanying floorplan. The problem is that the algorithm is taking too long to run on the adopted architecture. Efforts to minimize the schedule and cycle time have only obtained a 17 cycle schedule running at 37.4ns cycle. Realizing this performance problem, the designer identifies the critical path, which is running between the register file and the ALU, and introduces a bypass line (option A). With the addition of this bypass line, the critical path becomes the line between the register file and the multiplier. The addition of another bypass line (option B) causes the critical path to return to the link between the register file and ALU, but this time the path uses the original bypass line. The performance increases are detailed under the heading "Critical Path Optimizations."

At this point, the designer is wise to reinvestigate the scheduling possibilities. Because of the increased interconnection, routing path combinations are now possible which were formerly infeasible. The best schedule for each potential design is listed under the heading “Rescheduling Optimizations.” As expected, minor reductions to the schedule are possible since the data path relinquishes its dependency on the global bus. These improvements coupled with the cycle time improvements result in substantial performance benefits. Still, this rescheduling only considered the timing benefits from utilizing a predefined critical paths. The problem is that the scheduler did not have the ability to evaluate the timing options in order to improve the critical path through rerouting. If the scheduler had such ability it would be able to reduce the cycle time by rerouting operands through the multiplier instead of over the original bypass line. In retrospect, the designer would realize that “option A” is not required. This reduces the distributed RC and further reduces the clock cycle to the values shown under the heading “Rescheduling for Timing.”

This example is intended to demonstrate a number of key points. The first of these is the flexibility which is inherent in a pre-existing data path. For example, the final alteration showed that the data path was capable of routing operands through a pipelined multiplier instead of a bus without requiring additional clock cycles. This is impressive because the route through the multiplier incurs two clock cycles of delay as operands pass through the pipeline register and register p. The second point is the benefit of a quality scheduler/compiler which is capable of utilizing such routing freedom in an existing data path. The third is the unpredictable nature of alterations. The designer could have little intuitive notion that “option B” would be a superior alteration. As this example shows, it is often wise to consider various combinations of changes. The final point is the benefit of merging a rating scheme (such as the timing used in this example) with the

evaluation or re-evaluation of high level decisions. This thesis will present flexible data-path analysis techniques with which to explore such trade-offs.

1.2 The Role of Change in the Design Process

This section is intended to broaden the scope of the problems that can be addressed by these techniques.

1.2.1 Background

The design process used to create a chip may be as custom as the chip itself. Yet given that, there are a number of general statements which are applicable to most designs. First, a typical design starts from a high level specification which summarizes the intended behavior or support of the intended product. Such a description, often transformed from a executable program into the form of a data-flow graph, permits the designer to check the functional consistency of the design. After a number of synthesis steps (high-level transformations), these high level descriptions are eventually cast into an RT-level (register transfer level) description. Often the RT-level structure is in the format of a data path where physical components and subsystems may be identified. Ideally, a *data-flow map*, linking the components of the high level description with the proposed data path, is available. The data-path controller may be constructed from this data-flow map. At this point, logic level synthesis tasks optimize the various RT-level components. The complexity of these tasks often requires that the design be submitted in portions, thereby restricting their “global view” of the system architecture. Finally, these elements are merged into a single design, verified, and then submitted to a foundry for production.

This design process is depicted in Figure 1.2 as a series of sequential steps. This figure utilizes the conventional high-level synthesis view of the design

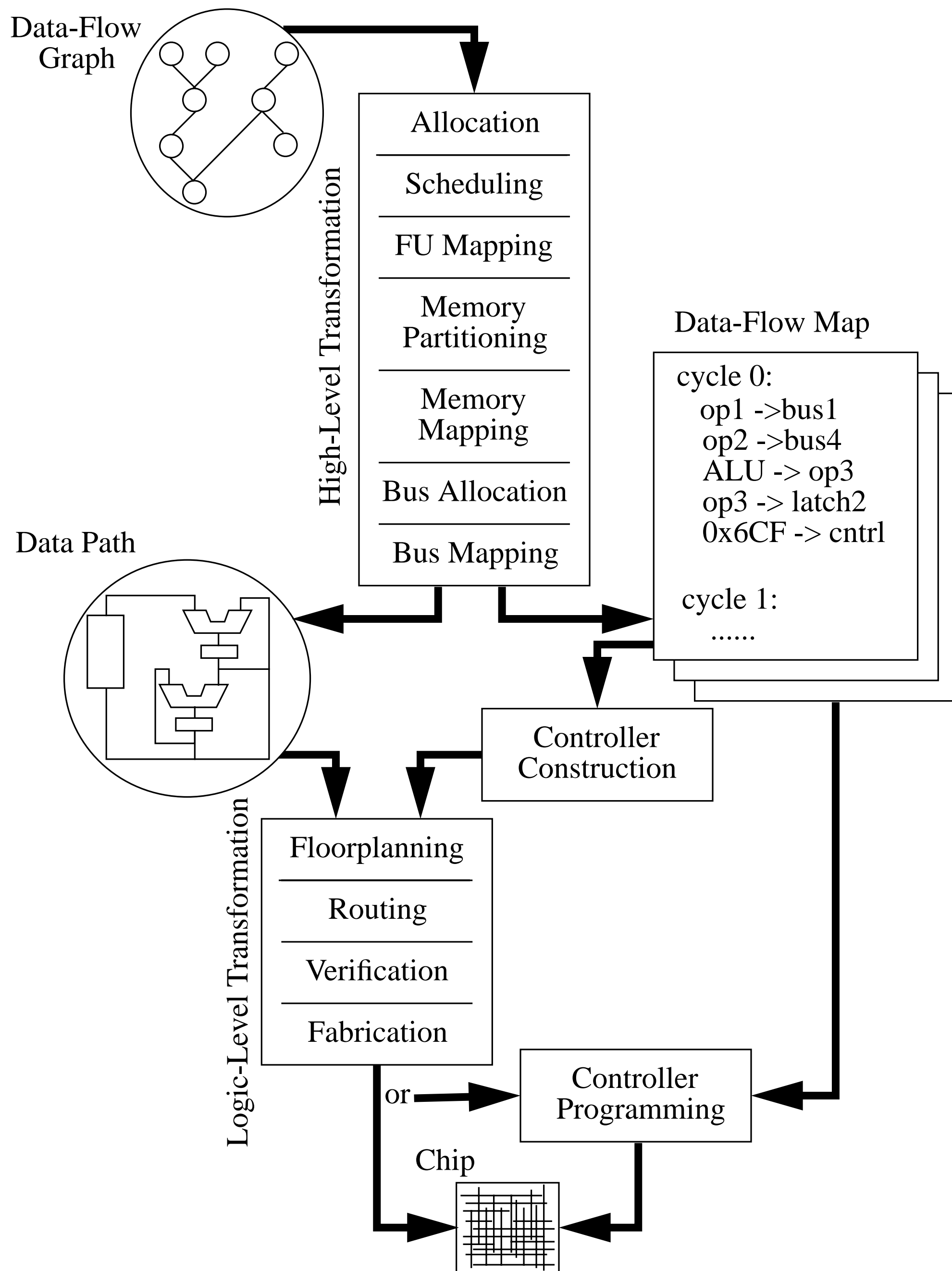


Figure 1.2 Idealized High-Level Synthesis Methodology

process, which partitions both the high-level transformation and the logic-level transformation into a series of tasks. These tasks have an ordered set of dependencies which are reflected in the figure. Unfortunately, the decisions made

at one level of abstraction affect the lower levels, such as the affect that the allocation of a bus has on routing and timing constraints. Since these effects may not be evaluated until much later in the design process, they may only be estimated when considering the options at a given abstraction level. The quality of these estimates are a critical matter for the design process. While errors stemming from underestimating can compromise a systems performance, overestimations can be equally dangerous since it can result in wasted potential. Furthermore, this specification of a sequential design process is idealized since a practicable methodology incorporates various levels of feedback.

There are many options which are missing from the process depicted in Figure 1.2, such as the use of a field programmable device or a DSP core and a retargetable compiler. One option that is depicted is the use of a programmable controller. This technique is used in a variety of products to allow the user to customize the product. This capacity is extremely enticing since it permits changes late in the design cycle without requiring any alterations to the low level design. Still, many products require the reliability and higher performance of a static, distributed controller interface.

1.2.2 Examples of change

As previously noted, feedback plays a critical role in the design process. Here, the designer attempts to accommodate inconsistencies in one level of abstraction by altering decisions which were made at higher levels. While the reasons for these changes may be quite varied, there are two main classes of changes: changes to the specification or changes required due to faulty estimations.

Changes to the specification of a design are extremely common. These alterations may result from changes in the projected market or errors discovered in

the initial high-level specification. In addition to changes resulting from outside forces, changes may result from sub-system integration. Sub-systems are used to simplify the data-path design process by partitioning the system into manageable sub-systems with well-defined interfaces. Miscommunication concerning such an interface can require substantial changes to a sub-system. Even if such miscommunications are avoided, the integration of these systems into a single framework may induce unforeseen errors related to power or propagation delays.

Even more pervasive are errors resulting from faulty estimations made at higher levels of abstraction. The precision with which the low level effects can be estimated at the highest levels of decision making is obviously limited. This limit is somewhat mitigated by steadily increasing the accuracy with each passing level of abstraction. But, the capacity to utilize these measurements is limited as the design becomes increasingly partitioned, as required by the complexity of the low-level analysis. Examples of errors resulting from such poor estimates includes: floorplanning constraints may require a smaller register bank, or timing analysis may prohibit the chaining of a certain operation or require that a bus be partitioned to reduce the propagation delay.

The use of feedback helps to accommodate these changes or errors. Many of the efforts in CAD synthesis have become multi-disciplined, making the high-level or logic-level transformation more of a single inter-related process instead of a series of sequential steps, as denoted by the groupings in Figure 1.2. While feedback within a given transformation is becoming more prevalent, the more formidable challenges lie in using feedback from the low-level in the high-level decisions. In this field, the primary interest is improving performance through high-level modifications to take advantage of the realized RT-level or low-level

structure. Modifications to the high-level structure after the RT-level or low-level structure has been created will be denoted as *late changes*.

1.3 Accommodating Late Changes

Two interesting methods for accommodating a late change are: make minor modifications to the data path and controller or modify the controller without altering the data path. A third option, completely re-initiate the design process, is typically unacceptable because of the amount of work which is wasted. This third option is only desirable if a change to the high-level specification occurs early in the design cycle. There are obvious problems with this approach for solving errors resulting from faulty estimations, since it is difficult to formulate more accurate estimations from a faulty design.

The option which maintains a majority of the data path is desirable since it limits the time spent in the design and verification processes. The ability to efficiently integrate changes into an existing system is directly related to types of alterations which are feasible. The first constraint on this feasibility is the ability of the existing design to incorporate change. Whereas an RT-level design can accommodate large amounts of alterations, a design which has completed the floorplanning and, potentially, routing portion of the design may only be able to handle minimal amounts of alterations. The second constraint is the complexity of evaluating such a modification. Selecting the ideal alteration is subject to the computational complexity and the data-path freedom.

As powerful as alterations to the data path can be, limiting the alteration to the controller can be even more desirable since it requires minimal modifications to the existing structure. This technique tries to use the freedom inherent in the existing data-path design, whether it be modified or not, to accommodate a

potentially modified data-flow graph. If the data path is not constantly using its full utility, it presents the opportunity to incorporate an altered data-flow graph with a minimal or no performance penalty. The chief benefit of this technique is the savings of invested man hours. For some designs, this savings can outweigh the increased performance resulting from more substantial alterations.

These two techniques for accommodating change may be extended to systems which have no initial data-flow map. As seen in the introductory example, the need to map a data-flow graph on to a preexisting DSP core or embedded processor may exist. This field of applications is an equally compelling area of research which has traditionally fallen under the field of retargetable compilers.

Regardless of whether the data-flow map is being generated or modified, the mapping of a data-flow graph onto an existing structure benefits from the increased accuracy in the estimates. Whereas high-level decisions previously relied on estimations of lower level constraints, many of these constraints have been evaluated. This increased accuracy can substantially aid the decision process related to these high level trade-offs. While the quality of the estimations are compromised by alterations to the data path or controller, the quality of the estimation can only exceed those that were initially used to evaluate high level issues.

While provocative, these approaches require modifications to the data-flow map and potentially the data path. Alterations which are made by hand should be discouraged in order to minimize any unforeseen human error. A number of conventional compiler techniques are available including retargetable compilers and micro-code compaction. But these techniques traditionally characterize only a subset of the data path's functionality in order to evaluate large problems in a reasonable amount of time. Furthermore, these systems are not set up to utilize the

information present in an existing, although insufficient, data-flow map. This thesis describes a system which characterizes the complete high-level functionality of the data path in order to systematically explore the possible mappings while maintaining the predefined structural design. This system is easily extended to incorporate additional low-level information by which to analyze performance trade-offs.

1.3.1 Mapping data-flow graphs

The required level of analysis is specific to the change which is to be accommodated. An alteration to a bus may only require an alternative yet compatible communication path. A tighter register constraint might require the storage of an operand to be remapped to a different memory device accompanied by a set of accommodating communications changes. But, in general, a change may require alterations to the operation schedule which in turn requires remapping of all affected units including function unit, memory, and bus elements. Thus, this *data-path-constrained scheduling* implicitly contains the re-mapping of operations, operand storage and communication problems. This thesis will, therefore, concentrate on accommodating this problem and address subproblems where appropriate. Figure 1.3 depicts the complete system which generates a detailed mapping for a data-flow graph onto a data path compliant with a suite of optional restrictions.

Traditional scheduling techniques do not adequately address data-path scheduling constraints. Fundamentally, such techniques assume that the data path contains an unspecified, universal switching network.[1] However, the actual switching network, a network of data buses and switching elements, defines a

---

1. The throughput of such switching networks is often bounded, but the functional model acts as a cross-bar switch.

limited set of conditional paths between components over which operands can transfer. In the absence of such restrictions, operand transfers are unlimited[2] which reduces the scheduling problem to one of maintaining operation precedence and resource limits. Such constraints permit one to formulate simple bounds on the solution space, such as ASAP and ALAP bounds and their generalizations. In contrast, systems with predefined switching networks introduce a new set of constraints upon operand transfers. To demonstrate the effect of these constraints, consider the data path and data flow pair depicted in Figure 1.4. Without the data path, the ASAP for the data flow is four cycles, but with the data path, the optimal schedule is over twice as long. Clearly the constraints such as operand movement and operand recomputation which were required by this example makes the classic ASAP and ALAP bounds inappropriate. In particular, such constraints are not adequately captured by communication bounds such as “bus limits” and require a more detailed model of the interconnection structure.

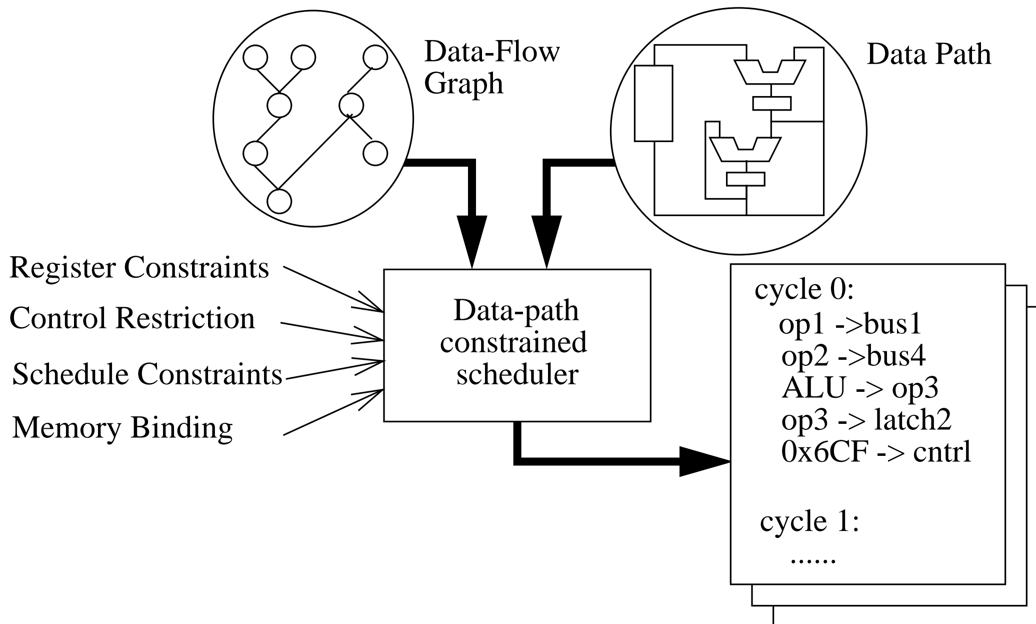
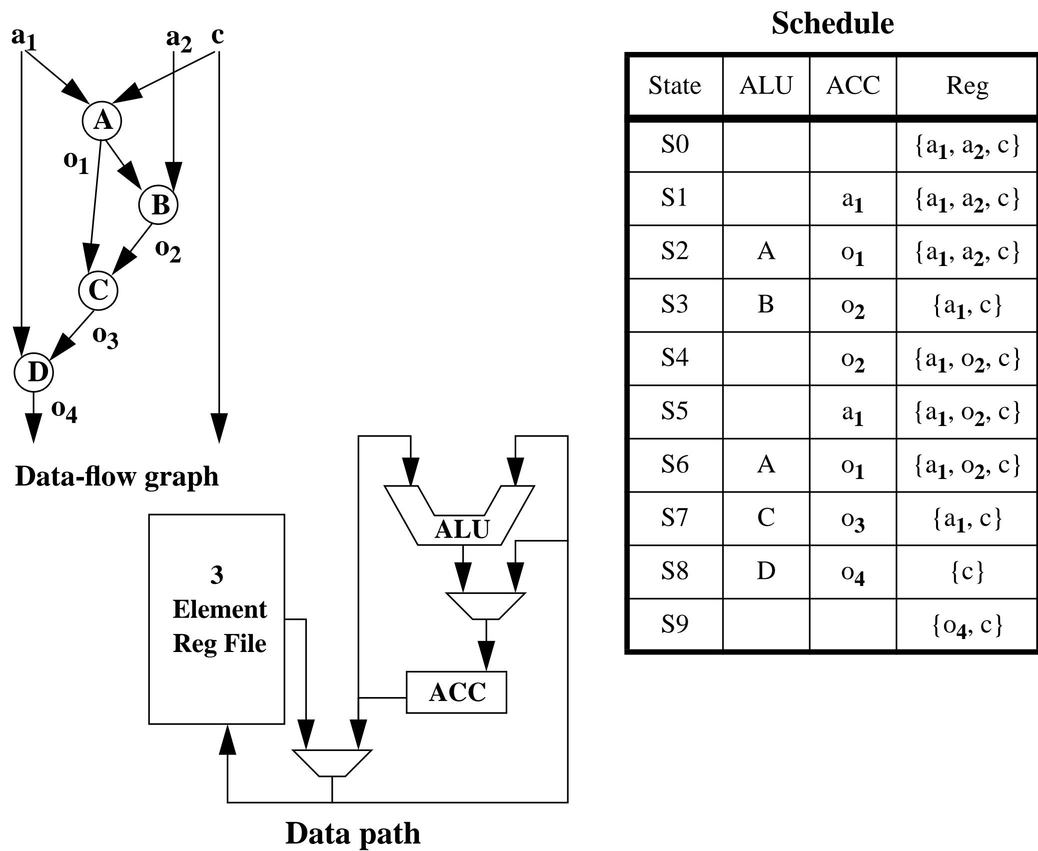


Figure 1.3 System overview

2. unlimited or at most bounded in number



State	ALU	ACC	Reg
S0			$\{a_1, a_2, c\}$
S1		a_1	$\{a_1, a_2, c\}$
S2	A	o_1	$\{a_1, a_2, c\}$
S3	B	o_2	$\{a_1, c\}$
S4		o_2	$\{a_1, o_2, c\}$
S5		a_1	$\{a_1, o_2, c\}$
S6	A	o_1	$\{a_1, o_2, c\}$
S7	C	o_3	$\{a_1, c\}$
S8	D	o_4	$\{c\}$
S9			$\{o_4, c\}$

Figure 1.4 Scheduling example.

Compiler techniques offer many benefits over scheduling techniques but are still inadequate for systematically addressing the data-path-constrained scheduling problem. Compilers' data-path models do incorporate limitations on operand transfers. A limited set of "instruction sequences", on to which the input data flow is mapped, are defined to provide the efficiency required for large scale problems. Even retargetable compilers generate an instruction set for the data path and then compile the data flow using sequences from this instruction set. This technique permits compilers to generate solutions to problems which are intractable for schedulers. However, it is impractical for the compiler to efficiently merge the tight communication and storage constraints without overlooking possible solutions. Typically, only a small subset of possible communications are used

resulting in suboptimal results. For problems of suitable size (e.g. inner loop optimization), higher quality results may be required. The techniques presented in this paper are particularly valuable in highly constrained problems.

In this thesis, I seek to bound the solution space more accurately by using a detailed symbolic model derived from the data path. This model accurately represents constraints such as:

- Limited Operand Transfer - An operand may travel between components only if the source device contains the operand, a bus exists between the devices, and the control signals enable a transfer across this bus.
- Bus Conflicts - Buses can only transmit a single operand at a time. Operations which combine two operands on the same bus are excluded.
- Register Constraints - Memory devices have a limited number of operands that they may store.
- Control Encoding Limitations - Specific combinations of control signals may be illegal.

For data-flow graphs of appropriate size, these restrictions sufficiently bound the solution space to permit the identification of optimal solutions to the data-path-constrained scheduling problem. As will be shown, the set of restrictions increase dramatically for re-mapping problems which utilize a pre-existing data-flow map enabling the optimal analysis of much larger problem instances.

1.3.2 Evaluating performance

The above technique can generate thousands of solutions. The quantity of solutions is only increased when alterations to the data path are considered. Whereas a single element may be randomly selected, there are a number of

additional criteria by which the quality of each solution may be rated. If the design is at the RT-level, the schedule length, register requirements, or control requirements may be used to differentiate the various solutions. Additionally, low-level designs may also incorporate timing and power information, which heretofore could only be estimated, as a measure to distinguish designs. The increased accuracy of these estimations results from the combinations of two factors. First, the detailed layout information helps to construct more accurate models of propagation delay or power models resulting from the interconnection structure. Equally important is the existence of a detailed data-path description. The specification of the routing options, including those paths formerly considered false-paths, binding options, and scheduling options supported by the data path permits high-level decisions to be reevaluated using the direct results as a measure. With this capacity, one may be able to not only accommodate change as required but to instigate change in order to improve the mapping of the high-level freedom on the low-level design.

Towards this end, a symbolic data-path automata is presented which models the execution freedom of several, alternative data paths. The automata can incorporate rating information by which solutions may be selected. The fundamental motivation for adopting this automata framework is its generality which permits it to be applied to a variety of applications. Furthermore, a combination of symbolic and reachable state techniques permit exact analysis of the solution space with which to identify optimal solutions. Binary decision diagrams (BDD's) are utilized to implement the proposed automata techniques.

Chapter 2

Related Work

The central idea to this thesis is the generation and evaluation of data-flow maps which enable a data path to support a data-flow graph. On the whole, this topic is prevalent in many fields including compilers and control generators. When the topic is expanded to incorporate data path alterations, additional fields such as automatic synthesis, engineering change, and timing evaluations become pertinent. While all of these fields are relevant, none match the scope of the system that is presented here. Table 2.1 gives a rough overview of how the proposed system, labeled "High-Level Appraisal", differentiates itself from the more traditional methodologies. These differences are discussed in ensuing sections. The first section addresses synthesis methodologies for high-level synthesis and for engineering change, and the second discusses compiler techniques for fixed architectures.

2.1 Synthesis Methodology

The system presented in this thesis was initially designed as a technique for synthesis. Many of the issues addressed in *high-level synthesis* [37,38,32,64,90] (namely scheduling, operation binding, operand binding, and communication

Table 2.1: Methodology Classification

Methodology	Given	Create/ Alter	Static	Objective
High-Level Synthesis	High-level specification	Data-flow map & data-path	n/a	Create optimal RT-level or low-level implementation
Engineering Change	Combinational network & new combinational specification	Combina-tional network	Data-flow map	Minimize change to original network to fulfill new specification
Compilers	Instruction set & high-level specification	Instruction sequence	Instruction set	Maximize system performance
Retargetable Compilers	Complete ASIC design & high-level specification	Instruction set & instruction sequence	Data-path & controller	Minimize program memory & Maximize system performance
High-Level Appraisal	Data-path & data-path alterations & high-level specification	Data-flow map & controller	Data-path	Maximize system performance

binding) exist in my system. But the two methodologies are very different. The essential difference is that high-level synthesis systems alter the schedule and bindings in order to create a better data path, instead of altering the data path in order to create more efficient high-level assignments. Essentially this is reversing the cause and effect portion of the synthesis design. This change in methodology

reflects the design environment of each problem. Whereas high-level synthesis assumes an environment of unlimited data-path design freedom, my system evaluates data-path cores with limited support of data-path modification. This is why the component allocation is fixed (instead of variable) in my system.

To understand the motivation for the proposed methodology, it is important to know the limitations of the top-down design methodology. High-level systems, such as CADDY/DSL [20], Cathedral [7], CHIPPE [12], CMUDA[84], HIS [20], SEHWA [78], traditionally use a top-down methodology. This methodology makes decisions for the higher level of abstraction and uses these results to guide the decisions about the lower levels. Even McFarland's BUD (bottom-up design) [64] utilized a top-down approach although it argued the importance of using detailed low-level library modules with which to evaluate the high-level decisions, such as design partitioning. While the top-down approach is effective, it has the disadvantage that earlier decisions may not be easily revised. In particular, synthesis systems rarely re-evaluate the high-level decisions when synthesizing elements below the RT-level. A notable exception are floorplaners, such as Fasolt [48], which consider rebinding communications to ease routing constraints. This inability to reevaluate high-level decisions at lower levels of design can be a major disadvantage when low-level synthesis identifies unpredicted problems in meeting the design requirements. The traditional approach with which synthesis systems handle these late inconsistencies is: "feedback and resynthesis." But, the effectiveness of this approach is inversely proportional to the scale of the considered modifications. Therefore, feedback from the low-level design to guide the high-level decisions is typically ineffective *unless* the low-level design remains relatively constant. Thus the inspiration for this thesis: modify around the low-level problems and then explore whether the high-level issues can accommodate the changes. This concept was independently proposed by Miyazaki and Ikeda

[69], but their model utilized a heuristic ASAP scheduler approach in order to analyze problems which are larger and control dominated.

This approach to accommodating design alterations late in the design process is similar to the field of *engineering change control*. In general, this research field addresses the use of controlled alterations to a preexisting design to accommodate some specification modification. The difference between engineering change and the technique described in this thesis is that engineering change tries to identify a minimal modification to the circuit to accommodate a predefined change in the high-level specification while my technique tries to modify the high-level specification to accommodate or optimize a predefined change to the data path. Because of these differing viewpoints of accommodation engineering change explores the freedom in the combinational logic level [83,91,36,10,59] and ignores the sequential freedom. While this technique is effective for its intended goal, it relies on the designer's ability to capture the intended performance of the system in a purely combinational format. By contrast, a high-level description of the problem would allow a system to explore the freedom inherent in the existing data path description and locate more effective solutions which require changes to the data-flow map and minimal or no change to the data path. An example of applying this technique is the alteration of operand transfers bindings, operation bindings, and operand bindings to improve the cycle time of a system.

The measurement/estimation of cycle time benefits from that model that this thesis describes. The main problem with timing calculations stems from the effects of false (infeasible) paths which were initially discussed in logic level combination analysis [33,21,80] but also effect RT-level models. While the use of path-sensitization has removed many false paths from the timing analysis [72], RT-level are hindered through the use of fully defined binding information. This hinderance

is becoming more prevalent as the timing models expand to incorporate propagation, switching, and control delays.[12,70,68] But these problems result from the fact that high-level decisions are traditionally evaluated in the absence of low-level information. These constraints are fortunately missing from my system enabling it to make up for the constrained data-path environment by more exact timing analysis. This subject is addressed in Section 6.2.

The internal data-path representation borrows heavily from data-path models resulting from the high-level synthesis community while also expanding these models to increase the design flexibilty. The earliest of these models were specific to register and multiplexer designs[77,79,76,41] but later expanded to incorporate register files[87] and pre-designed data path portions[34,73]. However, these systems universally rely on restricted data path models and with a few exceptions do not allow a predefined data paths. Instead, these systems construct an appropriate data path for a proposed data-flow graph. For example, although Parbus[34] and Splicer[76] allow predefined structures, both require limited interconnection networks (in order to bound the problem) which restricts the type of designs which may be modeled. Another example of restricted design description is Cathedral II[73] which requires the data path portion to be compiled from a portion of the data flow.

Before preceding to the next section, a final related research line of formal verification systems [11,17,42] should be noted. This field analyzes the execution freedom of an existing data path to ensure the correctness of the design. Although there are some obvious parallels, my model attempts to coordinate high-level components which are assumed to functionally correct. Therefore, my approach is free to introduce abstractions and simplifications which do not seriously detract from the power of the system but greatly enhance the speed.

2.2 Compiler Methodology

The automated generation of an instruction sequence to perform a specified task is traditionally though of as compiling. The freedom of each compiler is dependent upon the environment in the instructions are being generated. Speaking in general terms, the most liberal environment is in high-level synthesis which can allocate resources to aid the program execution. A slightly more constrained environment is that of retargetable compilers which may alter the set of instructions (assuming a VLIW architecture). Finally, the most restrictive environment is that of a compiler which is limited by the fixed instruction set of the target architecture.

For high-level synthesis, operation order is constructed through scheduling. Traditional scheduling determines a execution order of operands which preserves operand precedence and resource constraints in the absence of control conditions. Techniques which utilize heuristics[18,31,76,79], ILP[44], bipartite graphs[85], ROBDD[81], and reachable state analysis[92,27] have all been proposed. A common component among such techniques is the use of ASAP and ALAP bounds to limit the solution space and increase analysis speeds. In this area, Timmer demonstrates that such bounds can effectively linearize the solution space for certain problems.[85] Despite the individual merits of each scheduling technique, these techniques were developed for partially-defined data paths and do not fully model the resource constraints of a complete data-path design. Before being applied to a pre-specified data path, the assumptions including operand movement and operation recomputation and their effects on both the bounds and the scheduling techniques must be reevaluated.

An area of research which is well-suited for fully specified data paths is retargetable compilers. Retargetable compilers generate code which support a data

flow on a pre-specified ASIP or DSP architecture. This code generation is typically split into the task of identifying an "instruction set" for the architecture and then generating the code from this instruction set. While the task of generating the instruction set may be automatic, as described by Leupers[50,51] and Van Praet[88], it fails to utilize the data flow to eliminate large portions of the data path which do not concern the task at hand as we have previously demonstrated[71]. While the techniques for mapping the application data-flow graph into this instruction set vary, they are often characterized by tree pattern matching, as described in [4]. In a restricted view, such pattern matching techniques can produce optimal matches. But, the quality of the solution is limited by the quality of the instruction set. Furthermore, this matching technique assumes a static model of the target data-flow graph which is inappropriate to operand recomputation. And while the instruction set may be expanded to accommodate commutative operands, mapping associative operations is much more difficult.

Presently, the field of retargetable compilers has moved past the traditional problem of generating code are addressing a variety of specialized problems to optimize the resulting code.[52,54,55] Some of these problems are specific to a given architecture, but all are intended to give retargetable compilers an additional edge to make them practical. A survey of these problems may be found in [61]. Most of these problems are too specialized for the context of this thesis, and therefore shall not be reviewed on an individual basis. A notable exception is the work done by Liao on minimizing register or accumulator "spills"[54]. While Liao's work addresses the limited size of registers, it focuses on a single memory store and does not consider operand recomputation.

Of the various compiler techniques, those proposed by Massalin[62] and later expanded by Granlund and Kenner[39] share the closest parallel with this thesis.

These superoptimizing techniques use reachable state analysis of the instruction set to identify shorter instruction sets to perform equivalent data manipulation. While this work does generate optimal solutions, it is centered on identifying equivalent operation sets. This work requires a low-level system description and requires extensive modeling of operand values to enable pruning. Accordingly, the sequence of target instructions must be extremely compact in order to generate results. Furthermore, the input representation uses a detailed description of the instruction set instead of a direct data-path description and is therefore ill-suited for systematically analyzing data path alterations.

In closing, I would like to acknowledge the important work directed at incorporating control dominated and reactive systems. While this thesis adopts a traditional view of handling control, many researchers are expanding the capacities of schedulers[43,81,89] and retargetable compilers[24,52] to handle control data-flow graphs. While currently unsupported, these techniques must eventually be integrated into the system which is proposed in this thesis.

Chapter 3

Problem Formulation

In order to automate engineering change within a predefined structure, both the structure and the potential alterations must conform to a predetermined format. This chapter presents an overview of the various elements of the format which was selected for this work. This format is composed of four powerful techniques which allow the designer to explore a significant portion of the modification freedom. First, a uniform yet general model of data-path designs is adopted which strives to maintain a balance between specification freedom and complexity. Second, techniques for evaluating a set of data-path alterations are presented. Third, the definition of data-flow graphs is expanded to permit a fuller set of alternative yet equivalent operation sequences. Fourth, implementation details of the data-flow map, potentially stemming from previous data-flow maps, may be specified ahead of time to constrain the computational complexity. These techniques enable the utilization of user-motivated suggestions reflecting the fact that this approach is intended for controlled modification not for automated synthesis.

3.1 Modeling Data-Path Activity

Any attempt to model data-path activity is faced with the challenge of creating a symbology with which to describe the data path. The variety of data-path architectures, clocking schemes, and mixed operand types all combine to make a formidable problem. In this work, the activity on any data path is expected to fall under the framework depicted in Figure 3.1. The key to this framework is identifying a minimum set of RT-level behaviors from which an abstracted, high-end data path model can be expressed. The use of these basic behavioral types frees the system from modeling the detailed working of the individual components. It is this aspect which is crucial to the modeling of non-trivial designs.

The RT-level operation of any data path is as follows: Operands which are retrieved from either memory or the external world are passed through a common network of switching and combinational elements. The combinational logic can construct either new operands or control signals, such as the result of comparing

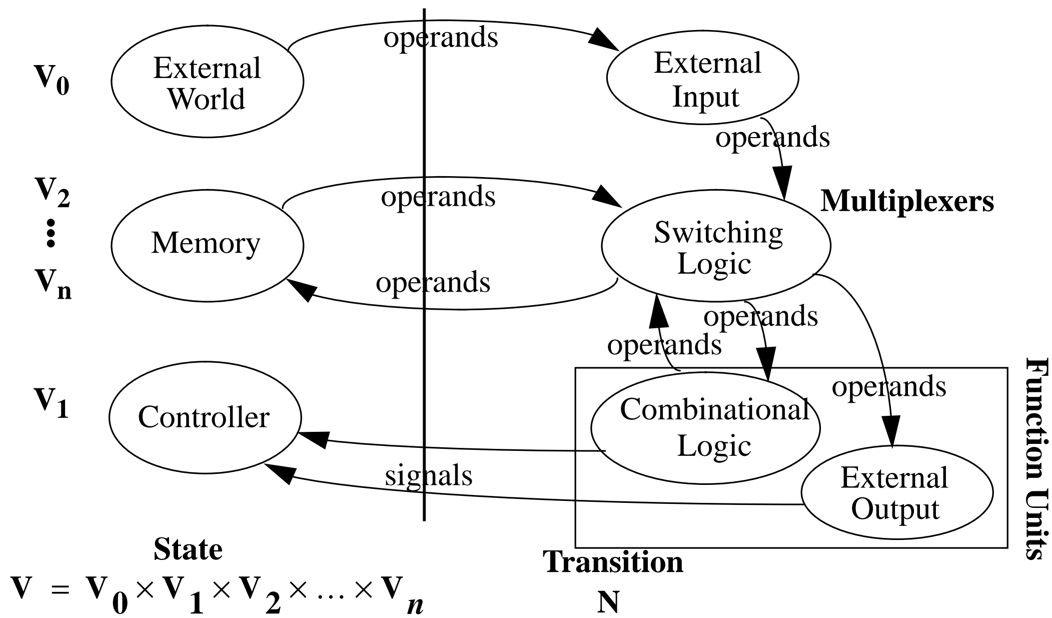


Figure 3.1 Data-path activity.

two operands. Whereas as the control signal is sent directly to the controller, the new operands return to the network of switching and combinational elements. Some of the retrieved or computed operands will be sent to memory devices or external devices from where they may affect the system on future cycles. Thus the system symbolically models operands and not operations. This permits detailed models of switching and storage units behavior while allowing conventionally operation scheduling.

A given data path is transformed into this model by mapping the various RT component of the data path using a set of specified component types. Memory devices are modeled as either latches or register files. Both external output and combinational logic can be modeled as function units. In this model, the transmission of an operand through an external output is captured as the transmission of a control signal. To prevent any data inconsistencies resulting from reading an operand twice from the external world, external inputs are modeled separately from functional units which, by contrast, may reproduce an operand as often as required. Finally, multiplexers are used as the sole model of switching logic which moves existing operands through the network. Any RT component which may not be directly mapped into one of these component types must be modeled as a *compound component*. A compound component is a component comprised of multiple base components to represent the various behavioral elements. Figure 3.2 displays an example of a compound component, a pipelined ALU, which must distinguish its switching behavior as well as its memory component from the combinational logic behavior.

This data-path model is converted into a finite state automata through the following steps. The state of the data path, V, is comprised of the current set of operands in the various memory devices plus operands from the external world

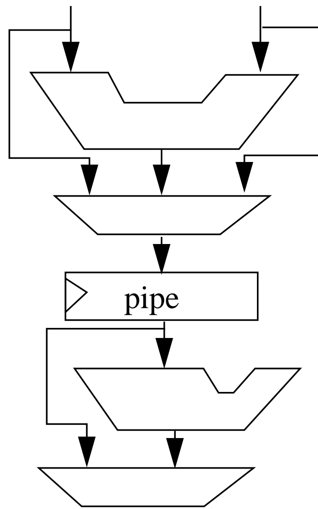


Figure 3.2 Pipelined ALU modeled as a compound component.

and signals sent to the controller. The input to the automata is the set of control signals, which are not depicted in Figure 3.1, sent from the controller to the individual data components. These signals identify the set of operands which are retrieved, routed, created, and ultimately define the next state of the data path. A single phase clocking scheme is adopted in this model which permits the synchronization of the various state elements denoted by the bar in Figure 3.1. From these restrictions, a transform relation, N, is constructed which represents this movement of operands by specifying the operating conditions under which any two states of the data path may be linked.

The problem addressed in this work is to identify a correct mapping for a given data-flow graph by exploring feasible data-path activity using reachable state analysis on the corresponding automata. The transition between these automata states will be restricted not only by the data-path limitations but by the creation of only those operands which are requested by the data-flow graph. For those problem of the appropriate size, an exact search of the reachable states from a given initial state is feasible. Such a search permits the identification of an optimal

series of data-path activity which links this initial state to a final state. Correct although possibly suboptimal mappings may be generated for problems for which exact enumeration is not feasible.

3.2 Competing Network Topologies

Alterations to the data path are modeled by multiple *network topologies*. Each network topology describes a unique interconnection of data-path components which, on its own, constitutes a valid data path. The difference between any two topologies may be as slight as altering the fanout of a bus or as complex as replacing a significant portion of the data-path design. Most alterations between two topologies may be specified as a modified wire connectivity, modified switching elements, replaced component, or a combination of such modifications. The set of alterations which are considered can be very complex. The set of topologies, identified by Υ, can be composed of multiple alterations being evaluated in concert. In this case, the set of topologies may be partitioned as $\Upsilon = \Upsilon_1 \times \Upsilon_2 \times \ldots \times \Upsilon_n$ were each Υ_i represents a set of alterations which is evaluated in a separate portion of the data path. Figure 3.3 depicts an example of two disjoint alteration sets affecting the interconnection and the component type of a single data-path design.

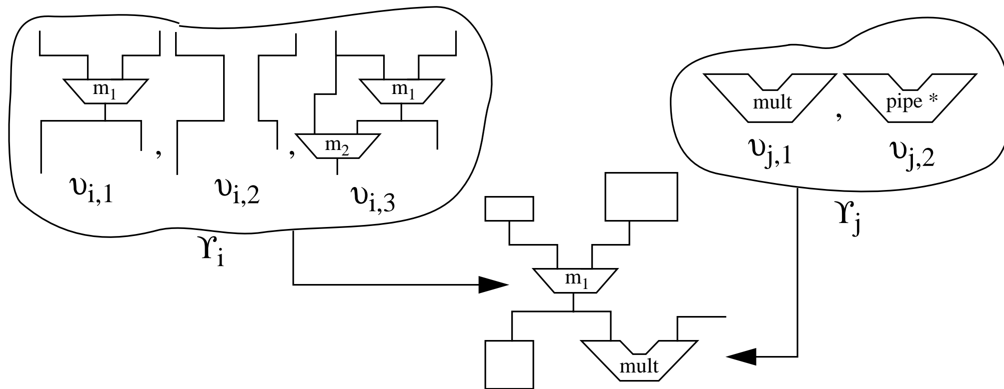


Figure 3.3 Disjoint topology alterations.

There are benefits derived from analyzing the set of different topologies concurrently instead of each topology individually. In a concurrent analysis, large portions of the analysis need not be duplicated for topologies which share some similar structure. Such similarity can result from many factors, such as: 1) the operational capacity of one topology is a subset of the capacity of an alternative topology. 2) large portions of the data path are common to all topologies. 3) the interaction between the set of disjoint topologies can create a common behavior for various sub-topologies. 4) the implied data-path mapping limits possible use of the data path. While the individual analysis may be shared through the use of a cache, the cache overhead and replacement policy can undermine their benefit and becomes a major complexity factor. This is not to suggest that concurrent analysis will always generate superior efficiency, but there are benefits when evaluating a series of data-path topologies which share a similar framework as demonstrated in Section 6.1

The specification of competing topologies merges the designs into a super structure which represents a single data path. The data-path specification utilizes topology variables to label the topology dependent elements. These labels appear only on connections between wires and component input ports, as shown in Figure 3.4. This format requires that all components appear in the data-path description even if they are only dependent upon a single topology. The use of the topology labels act as switches which limits the conditions under which operands may be passed to a given device. This permits a device which is specific to a given topology, such as the pipeline multiplier, to be ignored during those topologies for which it is not defined. However, topology labels do not act as true switches, whose settings may vary from cycle to cycle, since they must be consistent setting during every state transition. With the addition of these topology requirements,

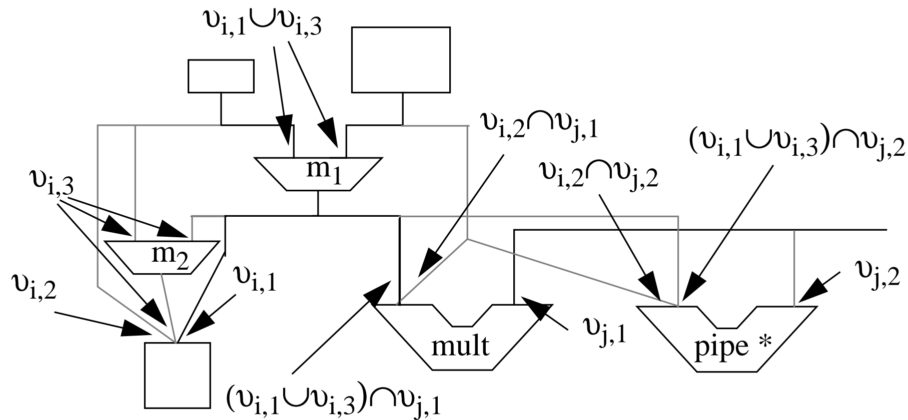


Figure 3.4 Merging alternative topologies into a single data-path design.

representation of multiple competing topologies is accommodated by the automata model.

3.3 Data-flow Alternatives

A data-flow graph is an acyclic directed graph which denotes the operation precedence inherent to the completion of a procedure. Figure 3.5 shows an example of such a graph. Starting from a set of initial operands (operands a through e), a series of operations are specified which create additional operands (operands o1 through o5). Each operation identifies a set of input operands and a pre-specified behavior with which to combine these input operands. Each input

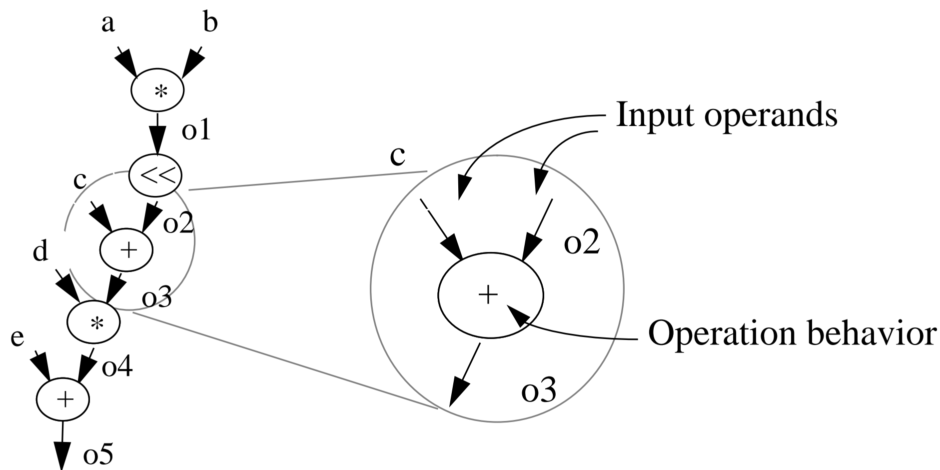


Figure 3.5 Example data-flow graph

operand is identified by a directed edge pointing from either an initial operand or a computed operand to an operation. Additionally, a set of final operands, as in operand o5, may be identified whose presence indicates the successful completion of the data-flow graph.

Not every data-path component may compute a given operation. The binding of an operation to a function unit must occur within the confines of an appropriate map which specifies the set of hardware components which may compute an operation. Schedulers traditionally use such an *operation map* to define a set of selected mathematical operations which are mapped to a set of supporting function units. Each data-flow operation then identifies a mathematical operation from this operation map. Additionally, operation maps can describe potential algebraic transforms, such as commutativity of operands, for an operation. Such transformations increase the chance that an operation may be scheduled. Despite all of this flexibility, operand maps do have some limitations in describing operations on a pre-existing data path. The fundamental problem is that the operand map is required to specify all of the alternative options for an operation. Yet, there are some alternatives which do not just require an equivalent device but use an entirely different mathematical operation, such as a shift instead of a multiply, or strength reductions in which several operations are replaced by several others.

The following approach was developed to accommodate a more complete set of alternative computations. First, each operation has a direct association with a data-path component. This technique has the additional advantage of being able to specify the exact mapping between input operands and the input ports of the selected data-path component. Second, each operation lists the operand which it creates. Third and more importantly, multiple operations may specify a common

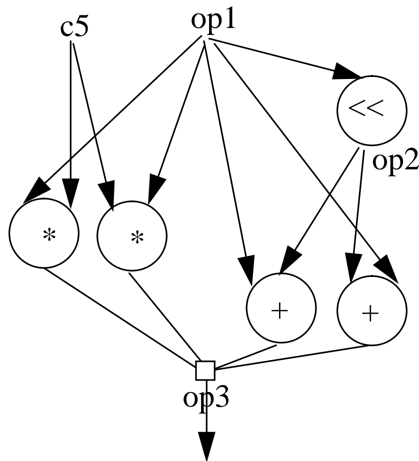


Figure 3.6 Representing alternative operations.

resulting operand. In this situation, any one of these operation sequences is sufficient to create the resulting operand. The existence of multiple operations only increases the set of alternatives with which the operand may be created. Figure 3.6 shows an example of four operations capable of computing operand op3 independently. These operations converge at a *alternative join* represented as a square point. This join is different than the joins typically associated with CDFG's (control data-flow graphs) which will permit only one of the operations to fire based upon a condition. Instead, the alternative join permits any of the *alternative operations* to occur if the data path may support it.

The ramifications of the use of alternative operations which converge at an alternative join are many. First, this specification shifts the focus from the execution of the operation to the computation of the resulting operand. This shift works well with the data-path model which models the restricted movement and computation of operands. Second, alternative means of computing an operand need not consist of a single operation but may utilize a series of operations as the shift/add alternative listed in Figure 3.6. This permits the evaluation of algebraic transformations, such as associative operations, as well as the strength reductions

shown in the example. Third, it removes the burden of representing all possible alternatives in a consistent table format of an operation map. Admittedly, the use of an operation map makes the data-flow specification much more concise. Therefore, the user must be careful to maintain the complexity of the data-flow graph by using the freedom of alternative operations wisely.

3.4 Incorporation of Partial Data-Flow Map

While the operation of the data path is limited by the data-path specification and the data flow graph which is being analyzed, additional restrictions may limit its operation. These restrictions may reflect system requirements, such as synchronization states from the external interfaces. Or, they may reflect a desire to maintain portions of a pre-existing, albeit inadequate, data-flow map to minimize the amount of resynthesis to be performed. In either case, it is important to incorporate these restrictions since they significantly reduce the search space analyzed in an exact search. In fact, the increased efficiency which results from such constraints makes it desirable to overconstrain the problem at the outset and then slowly relax the constraints on the data-path map until a feasible solution is identified.

3.5 Benefits of Operand Modeling

There are many benefits derived from the choice of modeling the operand movement through the data path. First, the movement of operands can be cast as a condition of the switch setting and the network topology, permitting the integration of multiple competing topologies into a single specification. Second, the modeling of operands allows alternative operations to be considered with greater freedom than traditionally modelled. Since an operand does not need to distinguish which operation produced it, the system is relieved from the burden of cataloging

operations encountered in operation-based systems. But all of these benefits are secondary to the chief benefit of this system: the detailed analysis of where operands are actually moving.

Instead of merely approximating the behavior of the data path, my model characterizes the operation of the data path. This permits the system to not only identify bounds of the data path's performance, such as minimal schedule length, but to evaluate the means with which these bounds are met. This characterization is critical to the support of performance analysis of the data-path operation. Here, the exact movement of operands can be labeled by their system requirements, such as time or power, and then used to make selective trade-offs to increase the system performance. Additional system requirements, such as memory usage or control requirement, may also be extracted from this type of model.

Chapter 4

Automata Representation

The backbone of the proposed technique is the ability to symbolically cast the restricted movement of operands through an existing set of data paths as an automata. Reachable state analysis of such an automata performs an exact search over the potential solution space from which a set of optimal solutions may be extracted. This chapter formalizes many of the techniques which were outlined in Chapter 3. Initially, a description of the input formats of both the data path and data-flow graph is presented. This is followed by an outline of the system objectives. Having presented these objectives, the components of the automata model and its application may be described. The remainder of the chapter addresses a number of practical issues required to make this automata model practicable. These issues are organized by: optimizations to the system, encoding issues, and the use of constrained data-flow maps.

4.1 Input Specification

In this section, the format for specifying the data path and data-flow graphs are presented. The formats were selected to permit the specification of a wide variety

of designs. Still, a series of restrictions are placed in the input format, but they are mainly designed to clarify behavior that would be otherwise ambiguous.

4.1.1 Data path

The following assumptions are made concerning the data paths to be modeled. First, the data path is assumed to be fault free. This assumption permits the data path to be modeled at the high level. Thus, the data-path model uses a RT-level description, and its values are symbolically represented as operands. The development of self-modifying circuits (most notably, circuits implemented with FPGA's) requires the second assumption to be specified: both the structure and control interface for the data path are assumed to be constant (time-independent).

The data-path model permits a limited specification of its control portion. Control signals travel from the controller to the components in order to instruct them as to how to behave. The data-path components may generate signals which are sent to the controller to specify the operation of future cycles. But the components are not permitted to generate signals which are directly sent as control inputs to other components. Such interaction between components presents many challenges that will not be addressed here. Finally, the interaction between signals coming from the data path and signals emanating from the controller is left unspecified.

The data path is modeled as a tuple (C, Ψ). Each element, c_i, of C is a data-path component defined by $(\Sigma_i, \Phi_i, \Theta_i)$. The set Σ_i defines a set of control lines, and the set Φ_i defines an ordered set of unidirectional input ports which connect to component c_i. The set Θ_i defines an unordered set of unidirectional output ports which is partitioned into $\Theta_i = \Theta_i' \cup \Theta_i''$ to distinguish the output ports which emit operands, Θ'_i, from those that emit signals, Θ''_i.[1] While two components

may share common control lines, they must always have disjoint input and output port sets. The functions $C(\phi)$ and $C(\theta)$ will be used to identify the associated component from either a input or output port specification.

A number of useful data-path attributes may be gathered from these definitions. The set Σ describes the complete set of control lines, $\{\sigma_1,\sigma_2,..,\sigma_n\}$, as defined by $\Sigma = \bigcup_i \Sigma_i$. The set Θ describes the complete set of output ports, $\{\theta_1, \theta_2, \ldots, \theta_m\}$, as defined by $\Theta = \bigcup_i \Theta_i$. Θ', and Θ'' are defined as the sets of operand and signal output ports.

Operands are transported between output and input ports over the data path's set of wires. I impose the constraint that each wire emanates from only one output port but may fanout to drive many input ports.[2] The set of associated input ports can be redefined with each changing network description. To accommodate this flexibility of the wire descriptions, a set $\Psi_i(\Theta, \Upsilon)$ is defined for each input port, ϕ_i, where each $\psi \in \Psi$ pairs an output port, θ, and a subset of network topologies, $\Upsilon' \subseteq \Upsilon$, over which the output port drives ϕ_i. These sets must be defined in such a way that an input port is never driven by two output ports for a given network description.

Component behavior

Each data-path component is assigned one of the five behavior types listed in Figure 4.1. Memory elements are represented by either latches or register files. Switching logic, used to conditionally transfer existing operands to different wires, is distinguished from combinational logic which creates new operands. In general,

1. Bidirectional ports are modeled by combinations of unidirectional ports, switching elements, and switching control restrictions.
2. Designs which drive a line from multiple sources typically utilize coordinated switching elements. Such designs are accommodated by merging these switching elements into a single switching component with a single source.

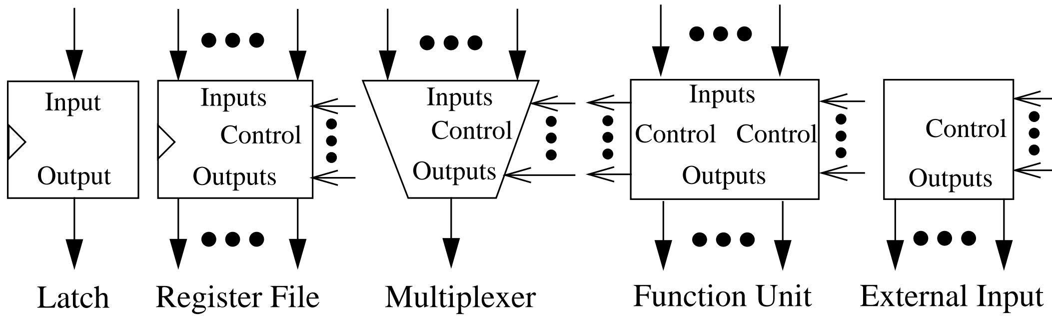


Figure 4.1 Base component set.

all switching components are modeled as multiplexers, and combinational logic blocks are referred to as function units. The external input components allow operands to be loaded onto the data path. The arrangement of these components and their connecting wires must ensure that each loop described by a consistent set of directional ports contains at least one memory device to prevent feedback races. All additional constraints are based upon the component's behavioral type and are summarized in Table 4.1.

The external input component has a unique behavior. While an external input introduce new operands to the data path much like a function unit, it must not be permitted to generate the same operand twice. While such behavior is permissible for function units, it implies an external storage device which often does not exist. For those cases where it does, additional memory devices and switching devices

Table 4.1: Behavioral Constraints

Behavior	Restrictions
Latch	$\lvert\Phi_i\rvert = 1$, $\lvert\Theta_i\rvert = 1$, $\Theta'_i = \varnothing$, and $\Sigma_i = \varnothing$
Register file	$\Theta'_i = \varnothing$ and $\lvert\Sigma_i\rvert = \lvert\Theta_i\rvert$
Multiplexer	$\lvert\Phi_i\rvert > 0$, $\lvert\Theta_i\rvert = 1$, $\Theta'_i = \varnothing$, and $\lvert\Sigma_i\rvert \geq \log_2(\lvert\Phi_i\rvert)$
Function unit	$\lvert\Phi_i\rvert > 0$ and $\lvert\Theta_i\rvert > 0$
Ext input	$\Phi_i = \varnothing$, $\lvert\Theta_i\rvert > 0$, and $\Theta'_i = \varnothing$

can be specified to model the explicit behavior. The set of output ports associated with external inputs shall be represented by the set Θ°. Note that external outputs do not require a special device behavior and are modeled as single input function units which only produce signals to the controller to note the transmission of particular operands. While a similar behavior may be achieved with the use of a register file, the function unit's ability to restrict the operand set used as input operands makes for a more efficient specification.

The behavior of many conventional data-path components will not directly correspond to one of these base behaviors. Such components are modeled as compound components by partitioning their various functional components and then connecting these components with wires. For example, a loadable register is broken into a latch and a multiplexer, as in Figure 4.2, to model the optional selection of storage. Given such techniques, the register file may appear to be a redundant entry in the set of base components since it could be modeled as a network of latches and switching elements. In fact, the register file's inclusion in the base set addresses a state representation issue instead of a functional issue. This problem with the state representation occurs when each element in an array of registers are functionally equivalent and equally accessible. Such register arrangements permit a factorial number of arrangements of the same set of operands. To prevent such explosive growth in operand/memory mapping, these register arrays are identified by the user as "register files" to permit specialized

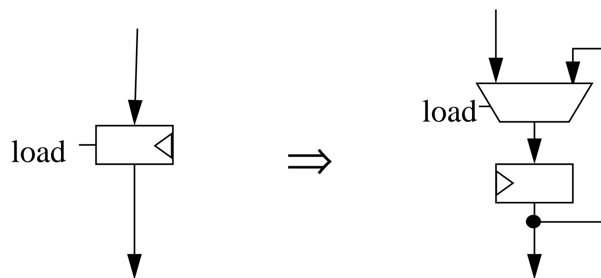


Figure 4.2 Representing loadable register with base components.

map encodings. Register files which do not comply with this description, such as those with specialized or limited access to certain elements, must be modeled as a network of switching logic, latches and/or register files.

Data-path operation

All activities of a data path are determined by its set of control lines during each clock cycle. The control is currently modeled under the assumption that the data path uses a single-phase clocking structure. As this model is not intended for timing verification, it is assumed that the control signals are well-defined and consistent over the span of a clock cycle.

The set of control lines, $\Sigma = \{\sigma_1,\sigma_2,..,\sigma_n\}$, are partitioned into two groups: those which control register file operand access and those which control all other component types. The motivation for this partition stems from the special encodings which will be used for register files. The register files store operands as an unordered set instead of placing them in specific memory locations. While this representation prevents factorial growth, it undermines the retrieval of operands from Boolean addresses. Therefore symbolic requests must be made to retrieve an operand in the absence of these addresses. The control lines to the register file transmit a symbolic value for each output port which specifies the requested operand. The notation of $\sigma_{k,\,\theta}$ shall be adopted to represent the request operand k from the output port θ of a register file. Such requests are satisfied only when the operand is an element in the register file's state encoding. The other set of control lines transmit Boolean values to multiplexers, function units, and external inputs. The specification of each multiplexer, c_i, must contain an encoding $\vec{\sigma}_i(\phi)$, defined over Σ_i, with which to select any input port $\phi \in \Phi_i$. To ensure that a unique input port can be identified for a given control setting, these encodings must be specified such that $\exists\phi_j, \phi_k \in \Phi_i | (\vec{\sigma}_i(\phi_j) \cap \vec{\sigma}_i(\phi_k) = \emptyset)$. The

requirements placed on the control lines of function units and external inputs will be detailed in the data-flow graph portion of the input specification.

While the set of feasible control vectors is bounded by the enumeration of the control line combinations, combinational constraints may restrict the set of permissible values to model complex interconnect or control word encoding. Such constraints exist when the control bits of the data path are heavily encoded such as in vertical micro-coded controllers. Such restrictions are modeled as additional constraints upon the set of state transitions and simplify the automata construction.

4.1.2 Data-flow graph

Data-flow graphs specify the dependencies between operands and operations. For this system, these graphs form directed, acyclic hypergraphs. A data-flow graph is a tuple (P, E) where P is a mixed set of operands and signals and E is a set of operations. The set P may be partitioned into $P' \cup P_1 \cup \{null\}$ where P' are the set of operands used by the data path, P_1 are the signals sent to the controller, and *null* is a special operand denoting "no operand." Each operation, $e \in E$, is defined as the four-tuple $(\theta, \vec{\sigma}, \Pi, p)$. The first element, θ, identifies an output port which will produce the result. The data-path component associated with the output port must be either a function unit or an external input. The operation of the this component is expressed in the control vector, $\vec{\sigma}$, which is defined over the appropriate Σ_i. The input operands to this device are specified as an ordered set of input operands, $\Pi = (\pi_1, \pi_2, \ldots, \pi_n)$, where $\pi_i \in P' \cup \{null\}$. The number of input operands must equal the number of input ports of the device to permit a matching of input operands to input ports.[3] The final element of the operation specification is the resulting operand or signal, $p \in (P' \cup P_1)$. Note, $p \in P'$ iff

3. If no operand is associated with a specific port, the null operand is used as a place holder

$\theta \in \Theta'$, otherwise $p \in P_1$. Additionally, the final subset of operands, $P_0 \subseteq P'$, are identified as those operands which may be read through external inputs, as identified by $\Pi = \emptyset$. Given this definition of operations, an operand p_1 is said to be a parent of operand p_2 and p_2 is said to be the child of p_1 iff $\exists e \in E | p_1 \in \Pi \cap p_2 = p$.

There are some non-traditional elements of this data-flow graph model. 1) I utilize a null operand to denote don't care information in the system. The null operand is used to represent either an operand that lays outside $(P' \cup P_1)$ or any operand regardless of whether it lies inside or outside of P. This first case is useful when describing any potential operand which is not explicitly defined by the data-flow graph as might be required by an initial condition. A need for the second case arises when formulating an operand constraint, such as an input operand requirement, but any operand qualifies to meet the constraint. 2) The use of alternative operations means that there are no restrictions on the number of operations which may generate any operand p_k. Each of these multiple operations provides a unique, alternative method to generate the operand. While the use of alterative operations may utilize a variety of function units, they must use a consistent set of output ports to prevent an operand from being specified as both a signal and non-signal. 3) The operation mapping explicitly lists a function unit's output port. Traditionally, this association is made by an operation map. But, the large disparity in function unit descriptions combined with the potential for highly tailored operations made such an operation table impractical. The enumeration of the associative and commutative operands as well as equivalent function unit listings, which are traditionally handled by the operation map, is accommodated through the use of alternative operations. 4) No two operands may be equivalent, where equivalency between two operands p_1 and p_2 is defined by EQ. 4.1. When

equivalent operands are detected, they should be merged into a single operand p_2; this can be done either automatically or manually.

$$\exists e_i, e_{j \neq i} \in E \mid (\theta_i = \theta_j, \vec{\sigma}_i \cap \vec{\sigma}_j \neq \emptyset, \Pi_i = \Pi_j, p_i = p_1, p_j = p_2) \qquad \text{(EQ 4.1)}$$

4.2 Problem Specification

This thesis investigates a methodology for modeling the constrained flow of operands across a set of network topologies. By modeling only the set of physical constraints, optimal mapping of a data-flow graph may be generated for a predefined architecture. The particular constraints which are modeled consist of: 1) an initial distribution of operands, 2) the limited routing capacity of the various data path topologies, 3) the limited set of operands to be constructed, as defined by the data-flow graph, and 4) any additional predefined constraints on scheduling or bindings. An automata representation of these constraints is constructed to facilitate an exact analysis of the data path freedom through reachable state analysis. This analysis is performed until a state is generated which matches an identified final state of the data path.

The novel elements of this technique are the optimal solutions which are generated and the model of the data-path activity. While the system utilizes some unique techniques such as the data path model, network topologies, and alternative operations, these techniques are only secondary issues through which the power of the system is extended. While the benefit of the optimal solutions is pellucid, the data-path-activity model is less overt. The ability to make quantitative evaluations of the data-path activity is increased by modeling the movement of operands, which are the principal cause of timing delays and power consumption, instead of the execution of operations.

4.3 Representing Data Paths

This section introduces the notation for the automata-based data-path model. This notation is intended to detail the operation of the automata model. Once these details have been described, the utilization of the automata to solve scheduling and engineering change problems can be presented.

4.3.1 Automata model

A symbolic automata is used to represent the storage of operands in memory components, the motion of operands on the switching network, and the creation of operands in function units. In its most general form, this automata is defined by the six-tuple $(V, \Upsilon, \Sigma, N, S_0, S_f)$.

V represents a finite set of states. Each state represents the status of the external inputs, the set of generated signals, and the contents of each of the data path's memory components. This set may be partitioned into the various disjoint components $V = V_0 \times V_1 \times V_2 \times \ldots \times V_n$ where V_0 lists which operands may have been loaded through external inputs, V_1 specifies the set of generated signals, and each V_i for i>1 denotes the current contents of a single memory device. In general, each V_i is defined over the set of pertinent operands, P_i. Whereas the sets external operands (P_0) and signals (P_1) have been defined in the context of the data-flow graph specification, $P_i = P'$ for i>1 to reflect the fact that any operand may be stored in a memory device. The range of each V_i is dependent on the data path portion being represented by the state space. For example, the set of signals which have been produced at a given clock cycle can potentially be any of the $2^{|P_1|}$ unordered subsets which can be constructed from P_1. This is reflected in the notation $V_1 = P_1^*$, where S^* denote the enumeration of all $2^{|S|}$ possible subsets of any random set, S. By utilizing the proper set of operands, similar formulations may be defined for the status of the external inputs and the contents of register

files. In contrast, a latch has a hard constraint on the number of operands which may be present on a given cycle: one operand. Therefore, the state space defined for a latch is defined as $V_i = P_i^1$, where S^1 denotes the enumeration of all $|S| + 1$ subsets of zero or one element from of any random set, S. While similar specification could accommodate the finite size of a register file, I choose to represent the state space of register file as $V_i = P_i^*$ and then apply a transformation constraint which ensures that the size constraint of a register file is not violated. The set V is used to represent the present state of a data path, and a second set of variables V′ are defined similarly to represent the next state.

While V is the set of possible states, the set of feasible states is constrained by the movement of operands permitted by the set of network descriptions, Υ, and the set of control lines, Σ, introduced in Section 4.1.1. State relations are defined by the transform relation N. This relation maps the set of feasible next states for each network topology, given the set of present states. Whereas N is traditionally defined over $V \rightarrow V'$, the presence of multiple network topologies requires N to operate over $\Upsilon \times V \rightarrow \Upsilon \times V'$. While the set of feasible states for a given network topology are limited by $\Sigma \times V \rightarrow V'$, Σ may be omitted from N since control line settings are not restricted by previous control lines values. I write $N(\Upsilon, V, V')$ as the symbolic representation of this state relation. While $N(\Upsilon, V, V')$ describes the transform relation for the entire machine, separate transform relations may be defined for each portion of the state space denoted by N_i as $\Upsilon \times \Sigma \times V \rightarrow \Upsilon \times V'_i$. In this case the transform relation may be rewritten as $N(\Upsilon, V, V') = \exists_{\sigma \in \Sigma} [\bigcap_i N_i(\Upsilon, \Sigma, V, V'_i)]$ to ensure that each sub-relation utilizes a compatible set of control settings. The process for constructing each element of this relation is found in Section 4.4.

Such individual transform relations are well defined for a given state, network description, and control vector because of the restrictions placed upon the input format. First, the restriction that each wire has a single source and that each input port connects to a single wire for a given network topology means that any combination of a network topology and a control vector describes a set of distinct paths through the switching network. Restricting latches to a single operand and the use of control lines to select operands from register files means that only distinct operands may appear at any path source emanating from a memory device. Restricting the data-flow graph to contain only unique operations means that only distinct operands may be produced by function units given a set of distinct operands at the inputs and a control vector. In the absence of a direct mapping between a given combination of a state, network, and control-vector and the operation for a given device, the operand produced by that device is the null operand. The absence of cyclical paths ensures that each path destination will have a distinct operand associated with the path's source.

These set of operands which appear at the set of output and input ports have the following effect on the state. Given a state v from the state V, the next state may be described in terms of the sub-states, v_0, v_1, ..., v_n, pertaining to the individual components of the state vector. If $P_{0,j}$ defines the set of operands retrieved by external inputs during the clock cycle j, then the set of external inputs grows to equal $v'_0 = v_0 \cup P_{0,j}$ after cycle j. Caution must be taken to ensure, $v_0 \cap P_{0,j} = \varnothing$. While the contents of a latch are defined as $v'_i = P_{i,j}$, all other devices utilize $v'_i = v_i \cup P_{i,j}$, where $P_{i,j}$ is the operand set associated with either Θ'' (for i = 1) or Φ_i (otherwise). A register-size constraint is violated for the ith register file when $|V_i| > RegisterSize_i$. To maintain the consistency of the design, V_i must be replaced by the $\binom{|V_i|}{RegisterSize_i}$ combination of operands.

$S_0(\Upsilon, V)$ and $S_f(\Upsilon, V')$ represent a set of initial and final states for the automata. The ability to specify sets of initial and final states gives the designer greater flexibility in determining both the proper initial and final state for the automata. Each of these state sets are defined by the user, and they must be defined such that a execution from any initial state to any final state over a consistent network description is valid since the system does not presently support conditional linkage between specific initial and final states. Moreover, each initial state must be defined in exacting detail to prevent the "invention" of operands. This includes specifying the content of V_0 and V_1 (typically set to $\varnothing$) in addition to the other V_i's. While empty register files may be set to $\varnothing$, empty latches must be initialized with a null operand. In contrast to the initial states, the final states should specify their requirements in a sparse format. These specifications will list the minimal required signal and memory bindings to complete the operation of the system. The user will place no constraint on V'_0 since this set of state variables are only used to restrict the set of allowable operations during a clock cycle.

4.3.2 Applying the automata

As applications of the automata are quite diverse, this section is divided into two major topics. The first topic is the application of reachable state analysis for the proposed automata system to the scheduling problems. The second topic details how particular solutions may be extracted.

Applications

The intended applications of this model make extensive use of symbolic reachable state analysis. While the particulars of the reachable state set is specific to the intended application, linking states from $S_0(\Upsilon, V)$ with states from $S_f(\Upsilon, V')$ is the primary interest. Therefore, I wish to compute $S_j(\Upsilon, V)$, the set

of reachable states on the jth iteration of the clock. The set of reachable states after a single clock iteration, $S_1(\Upsilon, V)$, is computed by:

$$S_1(\Upsilon, V') = \exists_{v \in V} [S_0(\Upsilon, V) \cap N(\Upsilon, V, V')] =$$
$$\exists_{v \in V} \left[\exists_{\sigma \in \Sigma} [\bigcap_i [S_0(\Upsilon, V) \cap N_i(\Upsilon, \Sigma, V, V'_i)]] \right] \qquad \text{(EQ 4.2)}$$

In general, the set of reachable states after the jth iteration is:

$$S_j(\Upsilon, V') = \exists_{v \in V} [R_j(\Upsilon, V, V')] \quad \text{, where}$$

$$R_j(\Upsilon, V, V') = \exists_{\sigma \in \Sigma} [\bigcap_i [S_{j-1}(\Upsilon, V) \cap N_i(\Upsilon, \Sigma, V, V'_i)]] \quad . \qquad \text{(EQ 4.3)}$$

Additionally, $T_j(\Upsilon, V)$ is defined as the total reachable state set, where $T_j(\Upsilon, V) = \bigcup_{i=0}^{j} S_i(\Upsilon, V)$. Such sets represents the cumulative state history after the jth iteration.

Eventually, one of the following conditions will be satisfied: $S_j(\Upsilon, V) \cap S_f(\Upsilon, V') \neq \emptyset$ or $T_j(\Upsilon, V) = T_{j-1}(\Upsilon, V)$. In the first case, a j clock cycle execution of the data-flow graph is identified. This execution is represented by the automata's use of state transitions linking an initial state and a final state. In the second case, the reachable state set has reached as steady state indicating that the exploration of the data-path freedom has been exhausted.

A *minimum-cycle scheduler* is defined as a system which identifies the set of state transitions satisfying $S_j(\Upsilon, V') \cap S_f(\Upsilon, V') \neq \emptyset$ where j is minimized. Upon $T_j(\Upsilon, V) = T_{j-1}(\Upsilon, V)$, the scheduler reports the infeasibility of the scheduling problem.

A *bounded minimum-cycle scheduler* is defined similarly but with an additional maximum cycle bound, k. Infeasibility is reported if $S_k(\Upsilon, V') \cap S_f(\Upsilon, V') = \emptyset$ thus omitting the need to maintain $T_j(\Upsilon, V)$.

In cases such as these, where the number of clock cycles are minimized, the following reduction may be applied. This reduction utilizes the fact that state sets need not be disjoint $S_j(\Upsilon, V) \cap T_{j-1}(\Upsilon, V) \neq \varnothing$. And more importantly, these common states may not lead to a minimal solution since any states reachable from this set are reachable from $T_{j-1}(\Upsilon, V)$ and must be reached at least a cycle later. Reductions to $S_j(\Upsilon, V)$ have the important benefit of reducing the complexity of the reachable state computation. Therefore, $S'_j(\Upsilon, V) = S_j(\Upsilon, V) - T_{j-1}(\Upsilon, V)$ is introduced from which the reachable state computation is modified to use:

$$R_j(\Upsilon, V, V') = \exists_{\sigma \in \Sigma} \left[\bigcap_i \left[S'_{j-1}(\Upsilon, V) \cap N_i(\Upsilon, \Sigma, V, V'_i) \right] \right] \quad .$$

A *cycle-constrained scheduler* is defined as a system which, given a cycle constraint k, identifies the set of state transitions satisfying $S_k(\Upsilon, V') \cap S_f(\Upsilon, V') \neq \varnothing$, i.e. a solution exists in k cycles. Infeasibility is reported if there are no elements of $S_k(\Upsilon, V')$ which satisfies this objective. Since this scheduler does not attempt to minimize the number of state transitions, the reduced state sets, $S'_j(\Upsilon, V)$, can not be used.

Information extraction

After the successful execution of the reachable state analysis, the set of state transitions which connect $S_0(\Upsilon, V)$ and $S_f(\Upsilon, V')$ in j cycles are determined by reviewing the set of state relations, $R_j(\Upsilon, V, V')$, generated during the reachable state analysis. This review starts by pruning the final set of reachable states by the set of final states, as in $S^\circ_j(\Upsilon, V') = S_j(\Upsilon, V') \cap S_f(\Upsilon, V')$. The set of state transitions which led to this set of final states are identified by limiting the known state transitions by the set of next states, as in $R^\circ_j(\Upsilon, V, V') = S^\circ_j(\Upsilon, V') \cap R_j(\Upsilon, V, V')$. Furthermore, the set of states from the previous cycle which are essential to this set of state transforms may be identified $S^\circ_j(\Upsilon, V) = \exists_{V'} R^\circ_{j+1}(\Upsilon, V, V')$ and used to successively generate

the preceding $R°_j$ until a pruned state relation is defined for every state transition. The resulting ordered set of state transitions represent every feasible solution found during the reachable state analysis. A single solution is represented by any series of relations $(r°_o, r°_1, \ldots, r°_f)$ where $r°_i \in R°_j(\Upsilon, V, V')$ and $r°_i \in r°_{i-1} \cap R°_i(\Upsilon, V, V')$.

Each of these relations lack the related control information which was removed in EQ. 4.3. While this removal is not essential, it dramatically reduces the complexity of representing each relation and thereby the complexity of producing the schedule set. Furthermore, the complexity of recomputing the associated control information for each state relation set drops significantly when the set of relations are utilized to prune the relation construction.

$$R_j°(\Upsilon, \Sigma, V, V') = \bigcap_i [R_j°(\Upsilon, V, V') \cap N_i(\Upsilon, \Sigma, V, V'_i)]$$

Such transform relations contain the essential system information. The states specify the memory mapping for every operand. The network descriptions identify the data-path connectivity requirements. The control information specifies the data-path functionality including operand generation (scheduling and function unit binding) and operand transfers (bus binding). With the exception of the register address lines, this control information also describes the minimal support required by the data path's controller.

Solutions can be graded in terms of their system requirements. For example, solutions which use minimal size register files, which simplify circuit verification by using a minimal number of functional units, which use a consistent set of control vectors, or whose set of operand transfers minimize cycle time may be identified by evaluating the requirements placed on the data path. Furthermore, detailed power models can be made since both the operands and their bus assignments are known for each execution cycle. While this set of evaluations is

useful for pruning solution elements, they generally require the construction of the solution set to identify the “minimal cost” before such pruning can be employed.[4] If the solution set still contains multiple elements after the set has been pruned to optimize system requirements, a representative solution may be selected at random.

4.4 Transform Relation

The individual transform relations, $N_i(\Upsilon, \Sigma, V, V'_i)$, are the key to the reachable state analysis. This section presents techniques for the construction and optimization of these transform relations. Initially, construction techniques capable of generating the transform relations directly from the input specification are presented. Unfortunately, the representation of these relations can be cumbersome and minimizing these relations is essential for processing large problems. Therefore, the following sections present a series of optimization steps to reduce the size of the transform relations. Whereas the first two techniques preserve the exact nature of the reachable state, the third employs a heuristic which may be used to address problems which would otherwise be intractable.

4.4.1 Relation construction

As noted in Section 4.3.1, any given state, network topology, and control vector will specify a well defined next state. But, constructing the transform relation by enumerating these conditions is inefficient. The transform relation contains a regular structure due to common topology elements, redundant control encodings, converging states, and a sparse operation set which results in the production of null operands by function units for most state and control vector

---

4. While some pruning can occur while generating the solution set, this ability is limited by the fact that states only encode the present state of the machine.

combinations. Therefore, a more efficient construction process which builds the relation directly from the input specification is preferred.

Instead of building the transform relation in one step, the construction process, which is presented, builds a series of sub-relations. The first relation, $\Omega_i(\Upsilon, \Sigma, \Theta)$, describes the set of feasible connection paths over the switching network for a given input port in terms of the network topology, control line settings, and output ports. Since function units cannot pass existing operands[5], this relation represents the data path's complete ability to route operands during a clock cycle. The second relation, $M_k(\Theta, \Sigma, V)$, describes the relation between state bits, control settings, and output ports required to retrieve specific operands from memory devices. The third relation, $F_k(\Theta, \Upsilon, \Sigma, V)$, is similar to the second relation, but it describes the conditions under which an operand is produced by a function unit or external input. Both the second and the third relation utilize the set of output ports to describe where the operands are retrieved or generated. The intersection of this port information with the first relation, $\Omega_i(\Upsilon, \Sigma, \Theta)$, will maintain only those ports which can drive ϕ_i and which can provide operand p_k under compatible control encodings and consistent network topologies. This intersection succinctly specifies how to get an operand to a specific location by combining the generation of the operand with the routing requirements. This separation of the generation and the routing of operands permits the operand requirements to be selectively formulated for only those locations where they are suitable. Placement of operands at the input ports of function units and memory devices shall be the principle use of this capacity.

5. Functional blocks capable of both passing existing operands and creating new operands are modeled as a combination of function units and multiplexers.

The output port relation, $\Omega_i(\Upsilon, \Sigma, \Theta)$, associates the set of output ports which can drive an input port, ϕ_i, with their required network and control settings of the switching network. Each of these relations are constructed by analyzing each of the network-dependent output ports associated with ϕ_i. If one of these output ports belongs to a multiplexer, a recursive construction is used to incorporate the reachability of the multiplexer's input port set. In such cases, each set of output ports, as defined by $\Omega_l(\Upsilon, \Sigma, \Theta)$, associated with each of the multiplexer's input port ϕ_l is subject to the input selection encodings, $\vec{\sigma}_i(\phi_l)$, from the multiplexer specification, as in:

$$\Omega_i(\Upsilon, \Sigma, \Theta) = \bigcup_{\psi_j \in \Psi_i} \left[\upsilon_j \cap \begin{cases} \bigcup_{\phi_l \in \Phi_k} \vec{\sigma}_i(\phi_l) \cap \Omega_l(\Upsilon, \Sigma, \Theta) & (c_k = C(\theta_j)) = \text{mux} \\ \theta_j & \text{otherwise} \end{cases} \right] \text{where} \quad .$$

This definition will converge because of the data-path restriction that memory devices are contained in each loop described by a consistent set of directional ports in a given network topology. Furthermore, the lack of feedback paths strictly through multiplexers permits a depth first evaluation of $\Omega_i(\Upsilon, \Sigma, \Theta)$ for every wire in a single pass of the data path.

The relation $M_k(\Theta, \Sigma, V)$ is defined for every operand $p_k \in P'$ to represent the set of conditions under which p_k is retrieved from a memory device. These conditions combine state encodings, output ports, and potentially control encodings. To aid the definition of state encoding requirements, the set of Boolean variables $v_{i,k}$ and $\bar{v}_{i,k}$ are defined to specify an operand's presence or absence in a memory device, where $v_{i,k} \leftrightarrow p_k \in V_i$ and $\bar{v}_{i,k} \leftrightarrow p_k \notin V_i$. The retrieval of an operand is dependent upon the operand's presence and, in the case of the register file, the operand requested for the output port. Any requests made by such register file control lines are specific to a particular output port of the register file, as specified in the first part of EQ. 4.4. By contrast, a latch has only a single output

port and no control lines which makes the retrieval condition a relatively simple combination of requiring the operand's presence and noting where the operand will appear, as shown in the second part of EQ. 4.4.

$$M_k(\Theta, \Sigma, V) = \bigcup_{c_i = \text{reg file}} \left(\bigcup_{\theta \in \Theta_i} \theta \cap \sigma_{k,\theta} \cap v_{i,k} \right) \cup \bigcup_{c_i = \text{latch}} (\theta_i \cap v_{i,k}) \quad \text{(EQ 4.4)}$$

The relation $F_k(\Theta, \Upsilon, \Sigma, V)$ is defined for each operand $p_k \in P$ to represent the set of conditions under which p_k is introduced to the data path as a combination of state encodings, control encodings, and output ports. In the case of external inputs, the relation must reference V_0 to ensure that p_k has not been previously loaded. As opposed to the previous relations which were defined for each data-path component, $F_k(\Theta, \Upsilon, \Sigma, V)$ is defined over the set of operations from the data-flow graph. The corresponding data-path components are derived from the operation specification, as in:

$$F_k(\Theta, \Upsilon, \Sigma, V) = \bigcup_{e_i | p_i = p_k} \left[F'_{\theta_i, \vec{\sigma}_i, \Pi_i, p_k, (\Phi_j | c_j = C(\theta_i))} (\Theta, \Upsilon, \Sigma, V) \right]$$

where each $F'_{\theta, \vec{\sigma}, \Pi, p, \Phi}(\Theta, \Upsilon, \Sigma, V)$ defines a relation for each operation, e. This operation based construction allows the system to disregard components which are inappropriate for a given operand.

The relation, $F'_{\theta, \vec{\sigma}, \Pi, p, \Phi}(\Theta, \Upsilon, \Sigma, V)$, has distinctly different formats for function units and external inputs. Function units ($\Phi \neq \emptyset$) require that all of the proper input operands appear at the correct input ports. This accounts for the main portion of the relation specification where the generation of the input operands by either function units, external input, or memory devices is intersected with the routing requirements to the specific input port. After this intersection is taken, only the control and network topology is of interest. Therefore, the port requirements of the input operand may be removed. This intersection may be skipped when null is specified as the input operand.[6] While operations using external inputs do not

require input operands, they require that the operand p_k was not previously loaded by checking $p_k \notin V_0$. In addition to the specification of either input operand or external input requirements, $F'_{\theta, \vec{\sigma}, \Pi, p, \Phi}(\Theta, \Upsilon, \Sigma, V)$ adds the requirements on the control vectors and specifies the new output port, as in:

$$\begin{cases} \theta \cap \vec{\sigma} \cap \left(\bigcap_{\pi \in \Pi, \phi \in \Phi} \exists_{\theta \in \Theta} [(F_\pi(\Theta, \Upsilon, \Sigma, V) \cup M_\pi(\Theta, \Sigma, V)) \cap \Omega_\phi(\Upsilon, \Sigma, \Theta)] \right) & \Phi \neq \emptyset \\ [\theta \cap \vec{\sigma} \cap \bar{v}_{0,k}] & \text{where } \Phi = \emptyset \end{cases}$$

In order to represent chaining of operations, the definition of each $F_k(\Theta, \Upsilon, \Sigma, V)$ is potentially dependent upon other $F_l(\Theta, \Upsilon, \Sigma, V)$'s. But the acyclical nature of the data-flow graph ensures that such dependencies are not self referential. While $F_k(\Theta, \Upsilon, \Sigma, V)$ can depend on $M_k(\Theta, \Sigma, V)$ and the set of $\Omega_i(\Upsilon, \Sigma, \Theta)$ relations, neither of these relations depends upon on $F_k(\Theta, \Upsilon, \Sigma, V)$. These facts permit a depth first construction of this set of relations.

The individual transform relations, $N_i(\Upsilon, \Sigma, V, V')$, are defined by the system's ability to load and maintain operands. If $N'_{i,k}(\Upsilon, \Sigma, V)$ denotes the ability to load or maintain operand p_k, then the individual transform relations may be constructed from their set of associated operands using:

$$N_i(\Upsilon, \Sigma, V, V'_i) = \bigcap_{p_k \in P_i} N_{i,k}(\Upsilon, \Sigma, V, V'_i) \text{, where} \qquad \text{(EQ 4.5)}$$

$$N_{i,k}(\Upsilon, \Sigma, V, V'_i) = (v'_{i,k} \cap N'_{i,k}(\Upsilon, \Sigma, V)) \cup (\bar{v}'_{i,k} \cap ((\Upsilon \times \Sigma \times V) - N'_{i,k}(\Upsilon, \Sigma, V)))$$

which ensures that the status of every operand is defined for any combinatorial of state, control, and network combinations. The definition of $N'_{i,k}(\Upsilon, \Sigma, V)$ depends on the type of device being considered. For example, both V_0 and V_1 must check whether the operand was previously generated or if it was created during this cycle. While the relation, $F_k(\Theta, \Upsilon, \Sigma, V)$, specifies the conditions under which an operand is created, $N'_{0,k}$ must be careful to use only those output ports

---

6. Alternatively, $F_{null}(\Theta, \Sigma, V, V'_e) \cup M_{null}(\Theta, \Sigma, V)$ may be defined as Θ'.

associated with the external input devices which it is supposed to be monitoring. Therefore,

$$N'_{0,k}(\Upsilon, \Sigma, V) = \{v_{i,k} \cup \exists_{\theta \in \Theta}[F_k(\Theta, \Upsilon, \Sigma, V) \cap \Theta^\circ]\} \text{ and}$$

$$N'_{1,k}(\Upsilon, \Sigma, V) = \{v_{i,k} \cup \exists_{\theta \in \Theta}[F_k(\Theta, \Upsilon, \Sigma, V)]\}.$$

By contrast, the operands stored in memory devices may originate from another memory device, as well as being created, and must be routed to the proper input port. While these two constraints are common for all memory types, the transform relations for register files (EQ. 4.6) are more complex than those for latches (EQ. 4.7). These simplifications stem from a latch's single input port as well as its inability to maintain pre-existing operands.

$$v_{i,k} \cup \bigcup_{\phi \in \Phi_i} \exists_{\theta \in \Theta}[(F_k(\Theta, \Upsilon, \Sigma, V) \cup M_k(\Theta, \Sigma, V)) \cap \Omega_\phi(\Upsilon, \Sigma, \Theta)] \quad \text{(EQ 4.6)}$$

$$\exists_{\theta \in \Theta}[(F_k(\Theta, \Upsilon, \Sigma, V) \cup M_k(\Theta, \Sigma, V)) \cap \Omega_\phi(\Upsilon, \Sigma, \Theta)] \quad \text{(EQ 4.7)}$$

4.4.2 Memory mapping optimizations

The formulation of $F_k(\Theta, \Upsilon, \Sigma, V)$ exploited the observation that an operand may be produced by only a subset of the data path's function units and external inputs. A similar argument could be made that an operand need not be stored in every memory device. For example, Figure 4.3 depicts a component arrangement where operands in latch, l, must feed directly into the multiplier. Since use of the multiplier is only defined for a specific subset of operations, only those operands which would be defined as an input operand for such an operation need be stored in that latch. Therefore the set of operands, P_i, used by a memory device, c_i, are redefined to contain only these essential operands. Since $P_i \subseteq P'$ instead of $P_i = P'$, the complexity of each $N_i(\Upsilon, \Sigma, V, V'_i)$ as well as the set of unique feasible states are reduced. Furthermore, the quality of the results of the reachable state analysis are maintained since these excluded operands can not effect that data

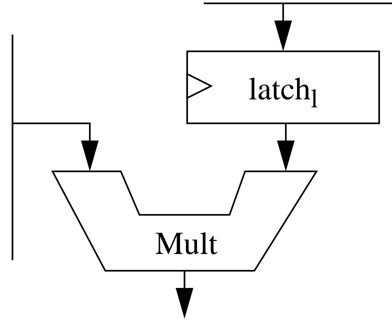


Figure 4.3 Dedicated latch

path's ability to create operands. Whereas the restrictions on $F_k(\Theta, \Upsilon, \Sigma, V)$ came directly from the data-flow operations, these restrictions stem from the data path's ability to support such operations.

To accurately limit the operand bindings for a memory device, requires detailed knowledge of the connectivity restrictions over multiple clock cycles. This connectivity analysis is compiled into a series of sets, $\tau_{i,x}$, where each set indicates the output ports which can be connected to input port ϕ_i by traversing no function units, x memory devices, and as many multiplexers as required. The construction of $\tau_{i,x}$ utilizes each output port and its associate component type which may connect to the specified input port as shown in EQ. 4.8. The network topology associated with each potential output port is not utilized, since these sets define the behavior for all topologies. A series of sets is defined for each input port for values of x between and including 0 to the number of memory devices.

$$\tau_{i,x} = \bigcup_{\psi_j \in \Psi_i} \left[\begin{cases} \bigcup_{\phi_l \in \Phi_k} \tau_{\phi_l, x} & c_k = \text{multiplexer} \\ \theta_j \cup \bigcup_{\phi_l \in \Phi_k} \tau_{W(\phi_l), x-1} & \text{where } c_k = \text{latch or register file} \\ \theta_j & \text{otherwise} \end{cases} \right] \quad \text{(EQ 4.8)}$$

To identify the set of memory devices which are applicable to an operand, the following observations are made. First, the data flow graph indicates the set of function unit input ports that an operand can appear. Second, the final state

requirements specifies the set of memory unit input ports at which an operand can appear. Third, operands must travel to these input ports by a combination of wires, memory device and multiplexers. The possible routes which could be taken may be represented as a collection of output ports which can reach a given input port ϕ_i regardless of the number of required clock cycles. Such a set of outputs ports is defined for each input port, ϕ_i as the set, $\tau'_i = \bigcup_x \tau_{i,x}$. Finally, given the list of input ports defined by the first two conservations, all of the output ports which can route the operand p_k to any of the locations it may want to reach can be determined.

The set of output ports of interest to operand p_k are defined as $\Omega'(p_k)$. Each port set is defined by both the final state specification and set of operations which use p_k, as in:

$$\Omega'(p_k) = \{ \bigcup_{v_{i,k} \in S_f(V)} \left[\Theta'_i \cup \bigcup_{\phi \in \Phi_i} (\tau'_\phi) \right] \} \cup \{ \bigcup_{e_j | p_k \in \Pi_j} \left[\tau'_{\phi_{e_j, p_k}} \right] \}$$

where ϕ_{e_i, p_k} identifies the input port associated with operand p_k. When considering the requirements placed on the final state, it is important to include the output ports of memory device specified in the final state as well as the ports which can reach this device. This prevents the operand from being excluded from the essential memory device. Form this set of output ports, the operand list, $P_i = \bigcup_{p_k \in P'} p_k \big| \Theta_i \subseteq \Omega'(p_k)$, is generated for each memory device with which to simplify EQ. 4.5. Additionally, the $M_k(\Theta, \Sigma, V)$ relations may be simplified to only consider components c_i where $\Theta_i \subseteq \Omega'(p_k)$.

4.4.3 Operand lifetime optimizations

The set of relations may be significantly reduced by noting that on any clock cycle j, each transform relation need only represent the set of states reachable from $S_j(\Upsilon, V)$. It is sufficient if the relation is defined over any $S_j''(\Upsilon, V)$ where

$S_j(\Upsilon, V) \subseteq S_j''(\Upsilon, V)$. Therefore, the set of relations $N^\circ_{i,j}(\Upsilon, \Sigma, V, V'_i)$ are utilized instead of $N_i(\Upsilon, \Sigma, V, V'_i)$, where $N^\circ_{i,j}(\Upsilon, \Sigma, V, V'_i)$ represents the state relation $\Sigma \times S_j''(\Upsilon, V) \rightarrow \Upsilon \times V_i'$. This new set of relations are generated dynamically for each iteration of the reachable state analysis.

The dynamic construction of transform relations is performed as follows. On each clock cycle, the set of operands, P′, is partitioned into a set of dead and a set of potentially active operands. The optimal set of dead operands, D_j, is defined to include operands whose presence in the next state at clock cycle j does not affect the reachable state analysis. The set of active states, A_j, is defined as $A_j \equiv P - D_j$. Given this partition, the individual transform relations may be represented as:

$$N^\circ_{i,j}(\Upsilon, \Sigma, V, V'_i) = \left[\bigcap_{p_k \in (A_j \cap P_i)} N^\circ_{i,j,k}(\Upsilon, \Sigma, V, V'_i) \right] \text{, where} \qquad \text{(EQ 4.9)}$$

$$N^\circ_{i,j,k}(\Upsilon, \Sigma, V, V'_i) = (v'_{i,k} \cap N'_{i,k}(\Upsilon, \Sigma, V)) \cup (\bar{v}'_{i,k} \cap ((\Upsilon \times \Sigma \times S_j(\Upsilon, V)) - N'_{i,k}(\Upsilon, \Sigma, V)))$$

While this representation is much smaller than the general transform, it requires operands to be selected for the set D_j. The following observations are made about constructing such a set. If D_j is defined as the optimal set of dead operands, a suboptimal set D_j' will be either $D_j' \subseteq D_j$ or $D_j' \not\subset D_j$. In the first case, the functional behavior of $N^\circ_{i,j}(\Upsilon, \Sigma, V, V'_i)$ is maintained. The only sub-optimality is that $N^\circ_{i,j}(\Upsilon, \Sigma, V, V'_i)$ will be more complicated than it needs to be. In the second case, $N^\circ_{i,j}(\Upsilon, \Sigma, V, V'_i)$ does not represent the complete set of reachable states. While this may cause the reachable state analysis to overlook reachable states, it will not generate any non-reachable states. Therefore, the sub-optimality of the second set classification is that it may generate sub-optimal but correct results.

Each D_j set is constructed by analyzing the lifetimes of the operand set. An operand’s lifetime is defined as the first cycle on which it may be scheduled (birth)

and the last cycle on which one of its children may be scheduled (death). Such a lifetime constitutes the cycles during which an operand must be present in the set of active operands. Operand p_k should remain in the set of dead operands until the following test is passed.

$$(F_k(\Theta, \Upsilon, \Sigma, V) \cap S_j(\Upsilon, V)) \neq \varnothing \quad \text{(EQ 4.10)}$$

But unfortunately, no such test exists to determine when p_k should return to the dead list. Subsequent sections will detail the best methods for bounding the death of an operand, but even these may be insufficient. For these cases, an operand's death condition may be estimated as the cycle j if every child of the operand is present in every state of $S_j(\Upsilon, V)$. This estimate may cause the set of dead variables to fall under the classification $D_j' \not\subset D_j$ hence this can be viewed as a heuristic speedup.

4.4.4 State reduction techniques

For problems of sufficient size, exact analysis is not practical. Such problems require the use of heuristics such as the "operand death approximation" discussed in Section 4.4.3. This technique pruned the transform relation, $N_i(\Upsilon, \Sigma, V, V'_i)$. An alternative approach involves pruning the set of states, $S_j(\Upsilon, V)$. Here, the elements of the state set are evaluated based on the number of operands which have been computed. Towards this goal a new set of variables, V'', are introduced. While V and V' describe the existence of operands in terms of specific locations, V'' describes operands present in the entire system as well of those that have completed their lifetime. Whereas V and V' represented a set of states, it is simpler to describe V'' as a set of bits, $\{p_0, p_1, \ldots, p_n\}$, where each $p_i \in P'$ and represents that the corresponding operand p either exists in the data path or has produced all of its children. The evaluation of these conditions is preformed by traversing the data-flow graph in reverse order, evaluating a operand only after all

of its children have been evaluated. Each evaluation sets p_k iff $(\bigcup_i p_k \in V_i) \vee (B_p \subseteq V'')$ where B_p is the set of p's children.

I define $S_j(\Upsilon, V, V'')$ as the state set, $S_j(\Upsilon, V)$, after the set of V'' have been evaluated. This permits each of the state elements to be evaluated in terms of the number of operands that it has produced. At this point one may prune based on the existence of specific operands or on the basis of the quantity of operands. This type of heuristic will be referred to as a "maximum utilization" heuristic.

The intersection of the functions, $\binom{|V''|}{m}$[7], and $S_j(\Upsilon, V, V'')$ identifies the subset of states which have at least m operands present or expired. The maximum number of operands present in a state set may be identified by iteratively increasing m's value. Requirements on the number of operands may then be enforced based this upper bound and a variety of user specified options. Such techniques enable the system to prune those elements which have grown stagnant or are not producing sufficient operands to have a practical impact on the solution set.

4.5 Encoding

I chose to represent the symbolic variables of this model with Boolean encodings. Boolean encodings have an advantage over other types of representation in that they can utilize conventional binary decision diagrams (BDD's). While these encodings are efficient, the encodings selected for the latches require that their associated transform relations be reevaluated. Additionally, the encoding selected for the register files, permit the specification of the register constraint to be expressed in a more exacting format.

---

7. The $\binom{|V''|}{m}$ function may be concisely represented in a graphical, Boolean format with only $m \cdot |V''|$ nodes using binary decision diagrams.

4.5.1 Selected codes

The set of symbols to be encoded include V, V′, Σ, ϒ, and Θ. The encodings for the first three of these sets are based on the decomposed sets $\{V_0, V_1, V_2, ..., V_n\}$, $\{V'_0, V'_1, V'_2, ..., V'_n\}$, and $\{\sigma_1, \sigma_2, .., \sigma_n\}$. The encodings selected for V_0 and V_1 are one hot encodings defined over $|P_0|$ and $|P_1|$ bits suitable for defining any subset of loaded external operand or signals. The encodings selected for V_i for i>1 depend upon the associated device type. Since register files may contain any subset of P′, each operand is assigned a one-hot encoding to represent any unordered operand set in $|P'|$ bits. But a one-hot encoding would be inefficient for a latch since latches contain one operand at most. Therefore, a set of unique binary encodings are used to represent the elements of the set $P' \cup \{null\}$ requiring $\lceil \lg(|P'|) + 1 \rceil$ bits. I choose to add an additional bit, the “null bit”, to identify the null operand from the other operands. The null operand is added to this set because a set of binary encodings can only denote the presence of an operand and is inappropriate to denote an empty set. The set of encodings for V′ are selected to be identical to those of V. The bits of V and V′ are interleaved to reduce the size of the BDD’s representing the various relations.

The encodings selected for σ_i depends on the type of control line. As noted in Section 4.1.1, control lines transmit either Boolean or symbolic values. Representation of the Boolean controls simply mirrors the Boolean encodings specified in the data path and data-flow graph descriptions. Since symbolic control lines are utilized by registers for specifying a single operand from the set P′, the content of these lines are encoded with the representation employed for encoding latch contents.

The set of network topologies, ϒ, use a binary encoding scheme since each design topology is disjoint. As noted in Section 3.2, the topology alterations can

often be partitioned into a set of partition sets $\Upsilon = \Upsilon_1 \times \Upsilon_2 \times \ldots \times \Upsilon_n$, where each Υ_i constitutes a network change for an individual wire which may be considered independent to any other Υ_j. In such cases, greater efficiency results by encoding each Υ_i over a separate set of binary encoding to promote sharing during common topology alterations.

Binary encodings are utilized to encode each output port in Θ. This is possible since any term from any of the relations which operate over the set of output ports always identify a single output port.

4.5.2 Re-evaluating latch transform relations

These encodings force us to revisit the construction of the transform $N^{\circ}_{i,j}(\Upsilon, \Sigma, V, V'_i)$. While register files dedicate a Boolean value for each operand with which to represent both existence and absence, latches share the bits with all the operands. This sharing requires us to only represent an operand's existence since representing its absence would interfere with the representation of other operands. Therefore, $N^{\circ}_{i,j,k}(\Upsilon, \Sigma, V, V'_i)$ is redefined for latches from its original specification in EQ. 4.9 to:

$$N^{\circ}_{i,j,k}(\Upsilon, \Sigma, V, V'_i) = (v'_{i,k} \cap N'_{i,k}(\Upsilon, \Sigma, V)) .$$

We are assured that any pair of $N'_{i,k}(\Upsilon, \Sigma, V)$ and $N'_{i,l}(\Upsilon, \Sigma, V)$ will be disjoint since each combination of a state, network topology, and a control line setting will result in at most one operand appearing at the input of a latch. Therefore the individual transform relation for latches can be redefined as:

$$N^{\circ}_{i,j}(\Upsilon, \Sigma, V, V'_i) = N^{\circ}_{i,j,\text{null}}(\Upsilon, \Sigma, V, V'_i) \cup \bigcup_{p_k \in (A_j \cap P_i)} N^{\circ}_{i,j,k}(\Upsilon, \Sigma, V, V'_i)$$

where $N^{\circ}_{i,j,\text{null}}(\Upsilon, \Sigma, V, V'_i) = v'_{i,\text{null}} \cap \{ [\Sigma \times S_j(\Upsilon, V)] - \left[\bigcup_{p_k \in (A_j \cap P_i)} N'_{i,k}(\Upsilon, \Sigma, V) \right] \}$.

The use of $N^{\circ}_{i,j,\text{null}}(\Upsilon, \Sigma, V, V'_i)$ is essential to make sure that $N^{\circ}_{i,j}(\Upsilon, \Sigma, V, V'_i)$ is defined for all control and present state combinations,

$\Sigma \times S_j(\Upsilon, V) \cdot N^{\circ}_{i, j, \text{null}}(\Upsilon, \Sigma, V, V'_i)$ represents the cases when latch i contains an undefined operand.

This introduction of null variables into the relation may cause useless states to appear in the set of reachable states. Let $S_j(\Upsilon, V)|_{v_{i,k}}$ be the cofactor of the jth set of states with respect to the ith memory device containing operand p_k and $S_j(\Upsilon, V)|_{\bar{v}_{i,k}}$ be the set of states where the operand, p_k, is absent. Any state defined by $S_j(\Upsilon, V)|_{v_{i,\text{null}}} \cap S_j(\Upsilon, V)|_{\bar{v}_{i,\text{null}}}$ is a state set whose elements have either a null or an operand in memory device, i, but are otherwise equivalent. When solving the data-path constrained scheduling problem, the set of states reachable from a state containing a null are also reachable from an equivalent states which contains an operand. Therefore, this subset of states of $S_j(\Upsilon, V)|_{v_{i,\text{null}}}$ which intersect with $S_j(\Upsilon, V)|_{\bar{v}_{i,\text{null}}}$ may be removed from the set of states $S_j(\Upsilon, V)$.

Given this observation, $N^{\circ}_{i,j}(\Upsilon, \Sigma, V, V'_i)$ may be altered to produce fewer states $S_j(V')$ which meet the above criteria. First it should be noted that certain control lines only influence the next state information for a specific V_i'. Figure 4.4 depicts an example circuit where the latch, l, has a dedicated control line, σ_k. The problem is that once the latch contains a null operand, the evaluation of every control combinations will result in the latch maintaining the null operand. But for

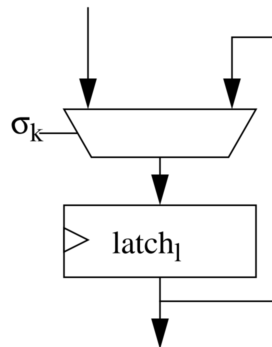


Figure 4.4 Dedicated control line example

this example, the next state analysis need only analyze the variable σ_k when evaluating $N^\circ_{l,j}(\Upsilon, \Sigma, V, V'_i)$ as in:

$$R_j(\Upsilon, V, V') = \exists \sigma_{\substack{m \\ \sigma_m \neq \sigma_k}} [S_{j-1}(\Upsilon, V) \cap \exists \sigma_k (N^\circ_{l,j}(\Upsilon, \Sigma, V, V'_i)) \cap \bigcap_{i \neq l} [S_{j-1}(\Upsilon, V) \cap N^\circ_{i,j}(\Upsilon, \Sigma, V, V'_i)]]$$

Since $\exists \sigma_k N^\circ_{l,j}(\Upsilon, \Sigma, V, V'_i) = \bigcup_{p_k \in (A_j \cap P_i)} \exists \sigma_k (N^\circ_{i,j,k}(\Upsilon, \Sigma, V, V'_i))$, $N^\circ_{l,j}(\Upsilon, \Sigma, V, V'_l)$ can be defined over $((\Sigma - \sigma_k) \times S_j(\Upsilon, V))$. Smoothing out this extra variable increases the chance that some $N^\circ_{i,j,k}(\Upsilon, \Sigma, V, V'_i)$ will be defined for those cases were the latch needlessly maintains the null operand. Furthermore, this reduction may be applied to all other transform relations since they are not defined over σ_k. The set of dedicated control lines may be identified through the analysis of the data-path structure. Once identified, a new set of control lines, Σ', may be utilized instead of Σ, where Σ' is the set of control lines which affect either register files or multiple memory components.

4.5.3 Register size constraints

Ensuring the constraints placed on a register's size is complicated by the encoding selected for register files. As noted in Section 4.3.1, the contents of each of the register files is described as $v_i' = v_i \cup P_{i,j}$. But this model causes the quantity of operands to steadily increase. To accommodate register constraints, register files must allow for operands to be lost or overwritten. Therefore, the set of present states are expanded to accommodate the loss of any operand, k, in any register file, j, as in: $S_i(\Upsilon, V)|_{v_{j,k}} \cap (v_{j,k} \cup \bar{v}_{j,k})$. This last step is effective for an ROBDD structure, where the representation of $(v_{j,k} \cup \bar{v}_{j,k})$ is simpler than the representation of $v_{j,k}$. In fact, the simplicity of this expanded representation can be exploited to reduce the complexity of the transform relation $N_i(\Upsilon, \Sigma, V, V'_i)$.

The following constraints are used to remove states which exceed the specific register file size from this expanded state set. A constraint is built for each register, i, over the set of Boolean variables belonging to V_i in the format: $\chi'_i = \bigcup_{j=0}^{\text{RegisterSize}_i} \binom{|P'|}{j}$.[8] The intersection of the state set and the set of register constraints maintains only those states which comply with the register size bound.

4.6 Additional Restrictions

Until now, the automata has executed under the assumption that every element of the data-flow mapping was unknown. There are many cases where a portion of the mapping is known or fixed by an external requirement. Consider examples where 1) operations interfacing with external devices are pre-scheduled, 2) large portions of the schedule are pre-determined but need final scheduling, 3) a memory mapping is required which implements a specified schedule, or 4) the compatibility of a completely defined data-flow map and a given data path must be tested.

Meeting these specifications restricts the behavior of the data flow graph, providing an opportunity to reduce the solution space. The formulation of these restrictions is highly dependent upon the nature of the data-flow map specification. Some constraints are easily incorporated into the transform relation, while others are more suitable as a constraint on the set of next states. These restrictions shall be specified as an ordered series of relations $(\chi_0, \chi_1, \ldots, \chi_n)$ where each $\chi_j(\Upsilon, \Sigma, V)$ can restrict either the input vector, the set of next states, or both, making EQ. 4.3 become:

$$R_j(\Upsilon, V, V') = \exists_{\sigma \in \Sigma} [\bigcap_i [S_{j-1}(\Upsilon, V) \cap \chi_j(\Upsilon, \Sigma, V') \cap N^{\circ}_{i,j}(\Upsilon, \Sigma, V, V'_i)]]$$

8. The number of BDD nodes required for the representation of χ'_i for any BDD ordering equals $(|P| \cdot \text{RegisterSize}_i)$

The transformation of a given constraint into a set of such restrictions will be presented as follows. Initially, the representation of control encodings and data flow graph restrictions shall be presented in this section. These restrictions are completely general to any application. But, the increased efficiency derived from operand storage and operand scheduling restrictions have demonstrated sufficient merit that they are presented as specific applications which appear in Chapter 5.

4.6.1 Control restrictions

Any restrictions placed on the control encoding may be specified as a set of allowable encodings, $\chi_j(\Sigma)$. Control restrictions model several properties. They can restrict combinational constraints placed on the control lines. Such constraints may result either from heavily encoded control bits, as found in vertical micro-coded controllers, or from component dependencies which maintain power limitations by restricting simultaneously active components. Additionally, control constraints provide an easy means to impose restrictions on the data-path specification. This technique would act as an alternative to editing the input specification in order to either minimize errors introduced into the specification or permit cycle dependent restrictions on the data path. Finally, cycle-dependent restrictions may be used to ensure the use of a component, such as an external input port. But, restrictions on the production of operands are best cast as scheduling restrictions.

When such restrictions are cycle independent, the construction of $M_k(\Theta, \Sigma, V)$ and $F_k(\Theta, \Upsilon, \Sigma, V)$ may easily incorporate of the restrictions placed on the control setting. But, such alterations are cumbersome in the case of cycle-dependent restrictions. In this case, it is often more efficient to use $\chi_j(\Sigma)$ when constructing $N^\circ_{i,j}(\Upsilon, \Sigma, V, V'_i)$.

4.6.2 Data-flow restrictions

Typical data-flow graph restrictions are posed as constraints on the set of operations. These restrictions prevent certain operands from being created, they are typically used to specify a subset of operations to use. This technique can be utilized to specify function unit bindings for the creation of an operand by retaining only the operation(s) associated with the selected device(s). In the absence of directly modifying the data-flow-graph specification, these restrictions may be incorporated into the formulation of $F_k(\Theta, \Upsilon, \Sigma, V)$ (if cycle independent) or the formulation of $N^{\circ}_{i,j}(\Upsilon, \Sigma, V, V'_i)$ (otherwise).

Chapter 5

Single Topology Applications

Both this chapter and the succeeding chapter demonstrate the applicability of the automata model to a variety of problems. While Chapter 6 addresses issues specific to multiple network topologies, this chapter presents issues pertinent to both single and multiple topologies. These issues pertain to constrained operation of a data flow. Whether accommodating an error late in the design cycle or porting the data flow to a core data-path, the designer will be faced with a number of constraints in addition to those imposed by the data path. Large amounts of testing and verification may be maintained if previously analyzed schedules and memory bindings are maintained. These constraints may be defined for the entire system or for only critical portions such as external interfaces. While these additional constraints may be modeled as additional burdens to the system, they should be seen as opportunities to restrict the operational freedom of the data path and the resulting automata model. By incorporating these constraints wisely, efficiency of the reachable state analysis is increased permitting the designer to analyze alternatives with greater accuracy.

This chapter will demonstrate how the reachable state analysis can incorporate a variety of external constraints. Initially, the most restrictive constraint, memory

binding constraints, will be analyzed. This section is followed by scheduling constraints. The final section shows that constraints derived purely from the data path itself may be construct in the absence of external constraints. Each of these sections will conclude with a result section which demonstrates how the techniques reviewed improved the system performance when analyzing a consistent set of benchmarks. The results will always focus on mapping a data-flow graph onto a data path with a single network topology in order to omit any misleading results stemming from multiple topologies. The effect that multiple topologies has on the reachable state analysis will be analyzed in Section 6.1.

5.1 Data-Path Routing

The principle challenge to executing a data-flow graph under the restriction of predefined operand mapping is insuring that the data path can support the routing and the creation of operands required to meet this constraint. Circumstances which are of particular interest to this data-path routing are the following: 1) designers may verify the feasibility of a fully specified data flow map, 2) implementation faults may be avoided by finding alternative routes, or 3) a partially defined data flow map may be completed by finding links for either undefined states or undefined potions of the data path. In any of these cases, the designer may specify the known constraints as a series of constraints on the next state of the machine. These constraints shall be represent as $(\chi_0(V'), \chi_1(V'), \ldots, \chi_n(V'))$, where each $\chi_j(V')$ represents either a single or multiple states of the data path for a given cycle. It is important to compose the constraint in terms of V' to ensure that any state pruning occurs as early as possible. While specifications for the operation set leading to this state may be as detailed as possible (denoting when operations will occur or identifying which of operation alternatives were used), this section will assume that no operation information has been provided.

5.1.1 Memory binding optimization

The availability of these $\chi_j(V')$'s dramatically redefines the reachable state analysis since the reachable states are known a priori; just the feasibility of the state transitions are left unspecified. This description gives the misleading indication that reachable states need not be applied in this environment. But the system can only be reduced to the verification of state transition if every element of the data path is defined for every clock cycle and if every state in $\chi_j(V')$ can be reached from a state in $\chi_{j-1}(V')$. In general, it is simpler to maintain the reachable state approach since it provides a minimum amount of overhead and is able to quickly accommodate alterations in the problem specification. While each state constraint may be applied directly to the reachable state, as in:

$$R_j(V, V') = \exists_{\sigma \in \Sigma} [\bigcap_i [S_{j-1}(V) \cap \chi_j(V') \cap N^\circ_{i,j}(\Sigma, V, V'_i)]] \quad ,$$

this would be inefficient since each $N_{i,j}^\circ$ is defined for many operands which will not be allowed to be stored in device c_i. By incorporating the restriction into the following dynamic construction of $N_{i,j}^\circ$, any illegal operand/memory pair is omitted.

$$N^\circ_{i,j}(\Sigma, V, V'_i) = \bigcap_{p_k \in A_i} [\chi_j(V') \cap N^\circ_{i,j,k}(\Sigma, V, V'_i)] .$$

This formulation of the memory constraints incorporates both the flexibility and efficiency desired by the user. The use of reachable state analysis allows the user to always identify solutions even when portions of the memory bindings are omitted. Additionally, the constraints from the defined memory bindings reduce the complexity of each reachable state computation, and the limited states constrain the set of reachable states to consider.

5.1.2 Routing Results

The feasibility of this technique is demonstrated by analyzing a series of benchmarks. Each benchmark pairs a data path and data-flow graph from the set of selected data paths and data flows discussed in the ensuing sections. The application program which performs this analysis was written in C++ and utilizes an in-house BDD package. All executions were run on a 141MHz SPARC Ultra with 416MB of memory.

Data path Benchmarks

Five different data paths, each with unique challenges, were selected to be used in the evaluation of the automata system. Four of these data paths are variations of the high level description of Texas Instruments' TMS32010 DSP processor. The first of these four design mirrors the TMS32010's data-path portion, the second design incorporates a second global bus to investigate the effect of added connectivity. The third and fourth designs are similar to the first two designs except that a two-cycle pipelined multiplier replaces the single cycle multiplier. Each of these four designs were coded utilizing the base component set introduced in Section 4.1.1 and are depicted as a composite in Figure 5.1. The dashed lines represent a second bus which was added for the two-bus examples. The addition of the second bus permitted the A multiplexer and the t latch to be replaced by a single multiplexer, B. For those design with the pipeline multiplier, a latch and second pipeline stage are inserted between the multiplier and multiplexer C.

The fifth benchmark demonstrates the applicability of the automata model to designs with multiple, dedicated register files. While such architectures are accommodated by a variety of compiler techniques, there are few such benchmarks in the literature. Therefore, I introduce a "Dual Register" data path which is used as our fifth benchmark and is depicted in Figure 5.2.

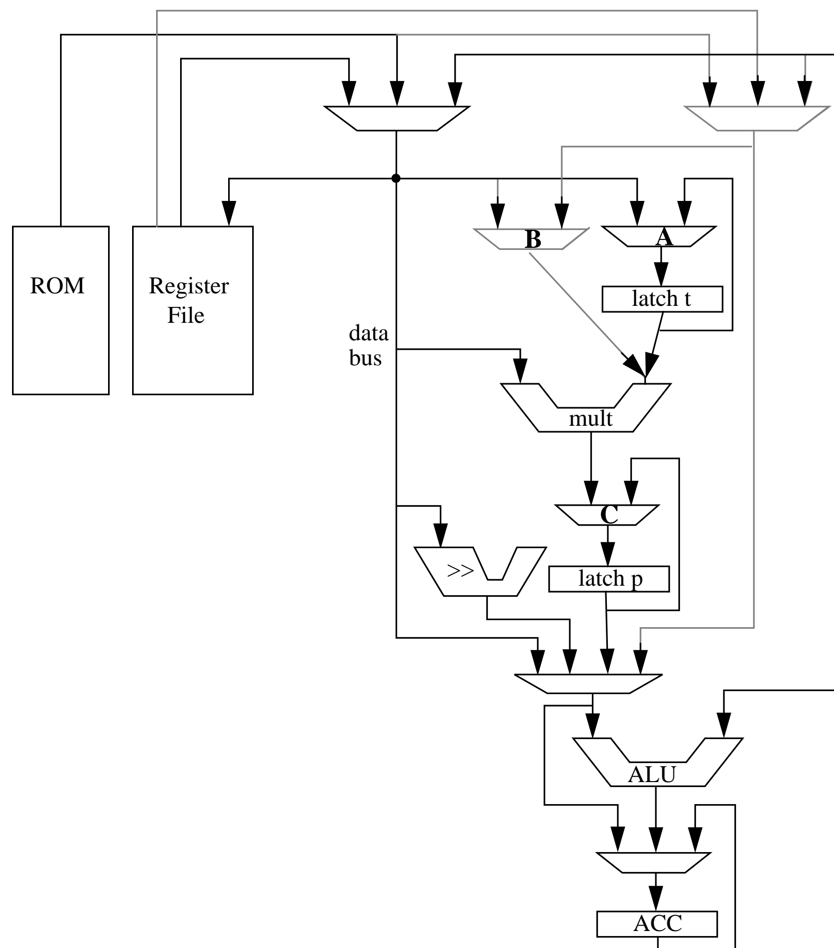


Figure 5.1 TMS32020 based data-path models

Data flow Benchmarks

The scheduling benchmarks utilized four data-flow graphs: differential equation (diff_eq), 3x3 determinant (3x3_det), differential heat release computation (dhrc), and elliptic wave filter (ewf). The determinant benchmark is a new benchmark and is specified in Figure 5.3a. Additionally, the dhrc benchmark contained many operations specific to memory index operations which were inappropriate for the control model of this thesis. Therefore, the modified dhrc benchmark shown in Figure 5.3b was utilized. Although not depicted, each of these data flows specify commutativity for each operand pair under the assumption

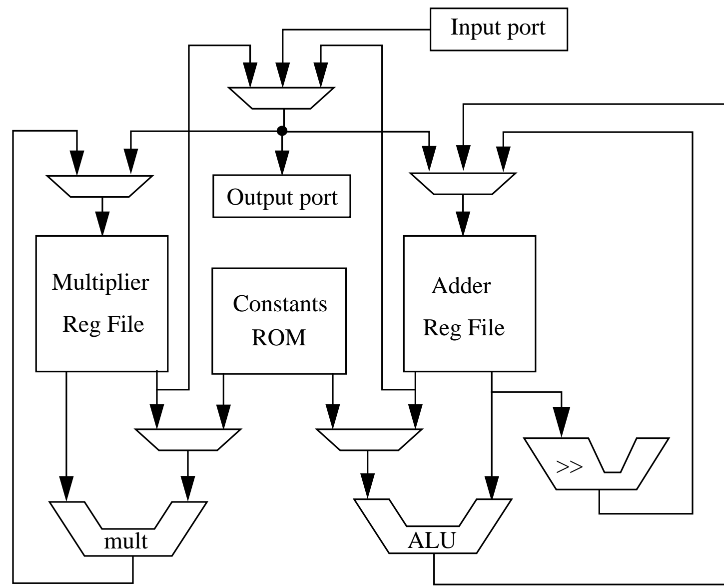
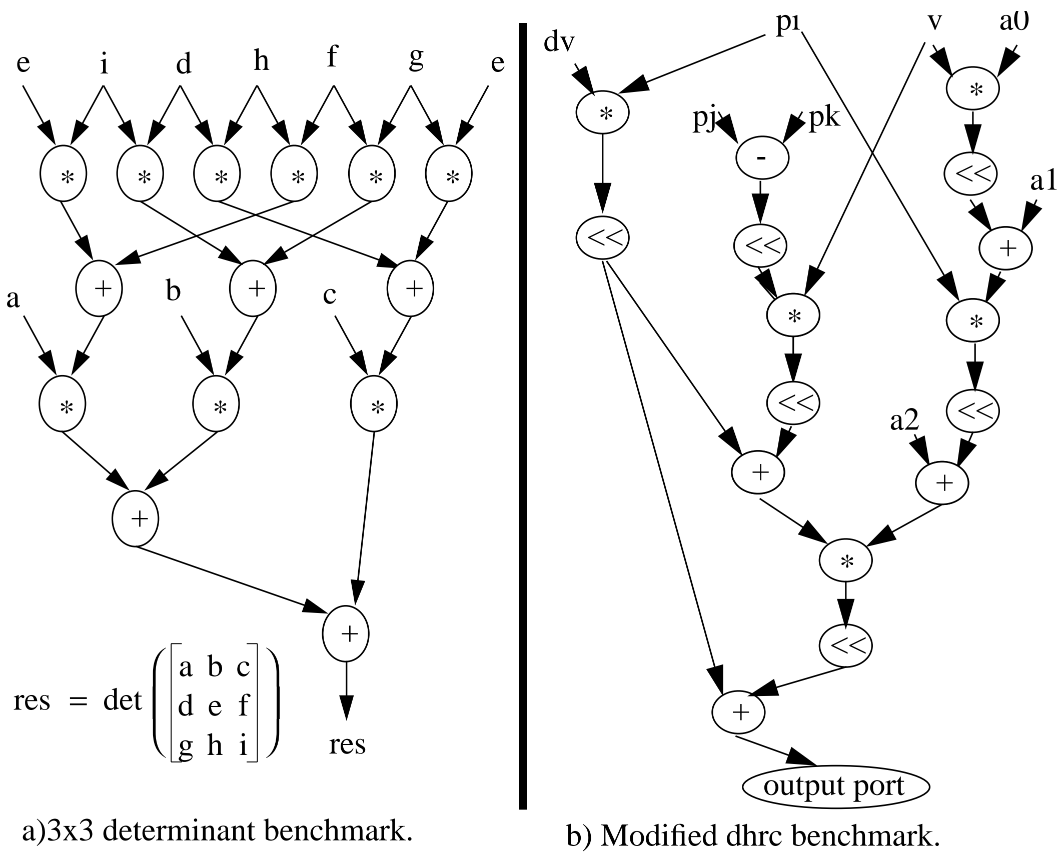


Figure 5.2 Dual register data path

that the ALU supports both forms of subtraction. Finally, each of these data-flow graphs were checked for redundant operations (as defined by EQ. 4.1) and automatically merged such operands during each execution.

The mapping of these data-flow graphs on to the specified set of data paths is dependent upon the initial and final state conditions. Table 5.1 specifies the initial and final bindings used in the benchmark suite. The table details not only the memory contents but the also the conditions on the external variables and control signals. The initial and final states selected for the diff_eq and ewf benchmarks facilitate the inner-loop nature of the data flow. By contrast, the 3x3_det and dhrc benchmarks compute a result which is returned to the register file or exported. Initial state entries which correspond to latches and contain "∅" utilize the null operand to indicate that no variable is present. The final states contain a number of entries with hyphens, indicating that no requirements are placed on the content of these states.



a)3x3 determinant benchmark.   b) Modified dhrc benchmark.

Figure 5.3 Novel data-flow graph benchmarks.

Run time complexity

The set of benchmarks were executed on a reachable state analysis machine using a series of memory bindings. These memory bindings indicated a single set of operand bindings for each cycle of the data path execution. Each operand binding was potentially incomplete since the content of empty latches were not specified.

Table 5.2 depicts the resulting run times for running the reachable state analysis for this routing problem on the benchmark set. The benchmarks are organized by the data-flow graphs and the data path utilized. The number of cycles associated with the execution of each benchmark is listed in the “# Cycles” column. In addition to verifying the existence of appropriate state transition, each

Table 5.1: Initial and Final operand bindings

Data Flow	State	tms32010-based designs					dual register file		
		reg	t	p	acc	ext./sig	mult reg	adder reg	ext./sig.
diff eq	initial	x, y, z	∅	∅	∅	∅	x, z	y	∅
	final	x’, y’, z’	-	-	-	y’ < a	x’, z’	y’	y’ < a
3x3 det	initial	a, b, c, d, e, f, g, h, i	∅	∅	∅	∅	a, b, c, d, e, f, g, h, i	-	∅
	final	res	-	-	-	-	-	res	-
dhrc	initial	pi, pj, pk, dv, v	∅	∅	∅	∅	pi, dv, v	pj, pk	∅
	final	-	-	-	-	res	-	-	res
ewf	initial	t2, t13, t18, t26, t33, t38, t39	∅	∅	∅	in	∅	t2, t13, t18, t26, t33, t38, t39	in
	final	t2’, t13’, t18’, t26’, t33’, t38’, t39’	-	-	-	-	-	t2’, t13’, t18’, t26’, t33’, t38’, t39’	-

execution verifies that the memory bindings complied with the specified register constraint denoted under the column “Register Size”. Since our system permits the binding for specific memory devices to be omitted, this register constraint is imposed on each S(V), instead on each χ_j. Because of the efficiency resulting from the binding restrictions, an exact reachable state analysis was performed. The execution times for constructing the set of relation sets and performing this reachable state analysis including the enforcement of register constraints are shown in the final column. The increase in execution times reflect the increasing complexity of the data-flow benchmarks.

5.2 Data-Path Binding

There are numerous cases were a designer wishes to schedule operations on specific cycles when mapping a data-flow onto a data path. For example, the data

Table 5.2: Data-Path Routing Results

Data Flow	Data Path			# Cycles	Register Size	Run Time (sec)
diff_eq	tms32010	single cycle mult.	1 bus	17	3	0.62
			2 bus	12	4	0.76
		pipeline mult.	1 bus	17	3	0.83
			2 bus	12	3	1.06
	dual register file			12	3, 3	0.36
3x3_det	tms32010	single cycle mult.	1 bus	20	9	0.86
			2 bus	13	9	1.03
		pipeline mult.	1 bus	20	9	1.00
			2 bus	13	9	1.49
	dual register file			22	10, 2	0.80
dhrc	tms32010	single cycle mult.	1 bus	22	5	0.88
			2 bus	19	5	1.20
		pipeline mult.	1 bus	23	5	1.22
			2 bus	21	5	1.79
	dual register file			19	3, 3	0.81
ewf	tms32010	single cycle mult.	1 bus	60	10	3.88
			2 bus	41	9	3.67
		pipeline mult.	1 bus	60	10	4.58
			2 bus	41	9	4.90
	dual register file			43	2, 9	4.66

path may have established interface constraints which must be met. Therefore, the execution of the operand loading and signal generation corresponding to external synchronizations must occur on predefined cycles. Alternatively, a designer may wish to reevaluate the data-flow map generated while synthesizing an RT-level design. While the RT-level structure may have been constructed from a schedule, the synthesis might have underestimated some penalties. In this case, the designer

may wish to reevaluate the binding and routing options to determine a better utilization of the existing system

In the presence of a set of scheduling bounds, the mapping of a data-flow graph onto a data path becomes a problem of binding. Three binding problems must be addressed: 1) where are the operands going to be stored, 2) which function units will perform the operations, and 3) which buses will transmit the operands to the proper memory and function units. As opposed to the data-path routing problem, the data-path binding problem lacks a set of predefined states with which to bound the reachable state analysis. While this can greatly enhance the complexity of the reachable state search, there are a number of bounds which may be extracted from the scheduling information to limit this resulting complexity.

5.2.1 Converting operation schedules into bounds

Scheduling constraints may be specified in a variety of ways. The constraint may specify which operation must occur on a given cycle or may just identify the resulting operand without identifying the operation that creates it. Additionally, the scheduling constraints may specify when an operand is created, when it is not created, or both. Finally the constraints may be specified for every operand or for only a subset of the operands. The following discussion will define the assumptions required for each constraint generated.

For any case where operand p_k must be created on cycle j, two constraints, χ_j and χ_{j-1}, may be generated. The first restriction, χ_j, ensures that p_k was created by restricting the control vectors, $\chi_j(\Upsilon, \Sigma) = \exists_{v \in V} \exists_{\theta \in \Theta} [F_k(\Theta, \Upsilon, \Sigma, V)]$. If the scheduling constraint lists an operation, e, instead of merely, an resulting operand, the control setting of the operations may be incorporated into the constraint, as: $\chi_j(\Upsilon, \Sigma) = \vec{\sigma} \cap \{ \exists_{v \in V} \exists_{\theta \in \Theta} [F_k(\Theta, \Upsilon, \Sigma, V)] \}$. This constraint may be

expanded to ensure the resulting operand is stored, $\chi_j(V') = \bigcup_i (v'_{i,k})$, if the data path cannot chain the operand into another operation. Such chaining restrictions can occur because of either scheduling constraints on the child operands or a lack of switching paths between the appropriate function units. Furthermore, $F_k(\Theta, \Upsilon, \Sigma, V)$ is only satisfied for the combination of topologies and present states defined by

$$\chi_{j-1}(\Upsilon, V) = \exists_{\theta \in \Theta} \exists_{\sigma \in \Sigma} [F_k(\Theta, \Upsilon, \Sigma, V)] \quad .$$

This constraint may be applied much more effectively when restated over the set of next state variables as $\chi_{j-1}(\Upsilon, V')$ to preserves only those states which permit p_k to be created on the following cycle.

Two constraints may be employed when the scheduling constraints specify the only times during which an operand must be created. First, the operand's actual birth cycle and its inclusion in the set of active operands may be easily determined from the scheduling constraints. This bound is much more efficient than the "birth test" introduced in Section 4.4.3 since it delays the introduction of an operand until it is actually needed. Unfortunately, this information is insufficient to generate any bounds on the death of an operand. Second, the information concerning the computation of an operand may be used to formulate more concise $N'_{i,k}(\Upsilon, \Sigma, V)$'s. These relations, which denote the conditions under which an operand is maintained or loaded, can utilize the knowledge of when an operand may or may not be created or maintained. This restriction is simplest stated by substituting either $M_k(\Theta, \Upsilon, \Sigma, V)$ or $F_k(\Theta, \Upsilon, \Sigma, V)$ for $M_k(\Theta, \Upsilon, \Sigma, V) \cup F_k(\Theta, \Upsilon, \Sigma, V)$.

The final set of constraints require scheduling constraints for multiple operands. Whereas the schedule for p_k defined its birth cycle, the schedule of p_k's

children define when p_k will die. This constraint is still just an upper bound since alternative operations may permit one of p_k's children be created without the use of p_k. Still this bound provides an important measure for managing the complexity of the reachable state analysis. Additionally, constraints may be derived for the gap between an operand's creation and the operand's use. Since an operand cannot be created without its parent operands, a series of constraints can be formulated to ensure that at least one set of the parent operand exist on the cycles preceding the scheduled operation. But this constraint must be formulated carefully to ensure that it only addresses parent operands which may no longer be scheduled and that it captures the exclusive nature of alternative operations are modeled. Therefore each operand has a set of alternative constraints, one for each alternative operation, which are combined to ensure that a set of parent operands exist or can be created, as in:

$$\chi_j(\Sigma, V') = \bigcup_{e_l | p_l = p_k} \left(\bigcap_{p_m \in \Pi_l} \left(\begin{cases} \bigcup_i v'_{i,m} & \text{where } j > lastbirth(p_m) \\ \text{tautology} & \text{otherwise} \end{cases} \right) \right)$$

5.2.2 Binding Results

Table 5.3 depicts the resulting run times for exact exploration of the binding freedom in the suite of benchmarks. Each of these executions utilized a fully defined schedule and employed all of the techniques presented in this section. Similar to the "routing" run times reported in Table 5.2, these "binding" run times reflect an exact search over the reachable states as well as the enforcement of register constraints. It is quite interesting to note the modest growth in the run times in comparison to those generated when consider the routing options. This growth is restrained as a result of the efficiency of the constraints derived from the schedules.

Table 5.3: Data-Path Binding Results

Data Flow	Data Path			# Cycles	Register Size	Run Time (sec)
diff_eq	tms32010	single cycle mult.	1 bus	17	4	0.76
			2 bus	12	4	0.92
		pipeline mult.	1 bus	17	3	1.02
			2 bus	12	4	1.24
	dual register file			12	3, 3	0.51
3x3_det	tms32010	single cycle mult.	1 bus	20	9	0.85
			2 bus	13	9	1.00
		pipeline mult.	1 bus	20	9	1.17
			2 bus	13	9	1.51
	dual register file			22	10, 2	0.88
dhrc	tms32010	single cycle mult.	1 bus	22	5	1.22
			2 bus	19	5	1.51
		pipeline mult.	1 bus	23	5	1.59
			2 bus	21	5	2.09
	dual register file			19	3, 3	0.95
ewf	tms32010	single cycle mult.	1 bus	60	10	5.31
			2 bus	41	9	4.80
		pipeline mult.	1 bus	60	10	5.75
			2 bus	41	9	5.96
	dual register file			43	2, 9	6.66

5.3 Data-Path Scheduling

When designers port a data-flow graph onto a core data path, they do not have the luxury of a preexisting schedule or predefined bindings. In this environment, the designer must schedule a data-flow graph within the constraints of the data path. For problems of sufficient size, optimal solutions may be generated. The essential difference which distinguishes problems which may be solved exactly

from those that can’t, is the clarity with which the lifetime of each operand may be bounded. Therefore, this section initially presents a technique for bounding the lifetime of an operand. The merits of this technique and other optimizations steps will be contrasted in the concluding result section. Additionally, the benefits of employing heuristic to approximate an operand’s lifetime will be presented.

5.3.1 ALAP bound generation

While lifetimes of an operand can not be determined exactly, they may be bounded relative to some cycle limit. Such cycle limits are provided with the bounded minimal-cycle scheduler and the cycle-constrained scheduler defined in Section 4.3.2. From such a limit, *ALAP* (as late as possible) bounds can be derived for the operation set based on the routing restrictions imposed by the data path. In general, an ALAP bound specifies the last feasible cycle on which a operation may be performed and still effect the final state. The limitations are derived from a combination of the operation dependencies and the data path resources which permit these operations to be performed. From these ALAP bounds which are computed for each operation, an upper bound on the death of an operand may be determined. Since an operand is no longer required after all of its children have been produced, the operand p_k can be considered dead once the cycle equals $\max_{e_i \in E | p_k \in \Pi_i} [\mathrm{ALAP}(e_i)]$.

The requirements placed upon an operands position, as defined in the data-flow graph and the set of final states, are utilized to improve the quality of the ALAP bounds. For example, each operation specifies an output port for each resulting operand. Also, an operation specifies a set of operand and input port pairs which must be satisfied. Each final state encoding specifies where an operand must be stored. The set of input ports corresponding to these final storage locations constitute a set of destinations for the set of final operands. The transfer of

operands from these output ports to the required input ports will often incur a sizable delay with which to bound the operating speed.

The data path can provide many obstacles to the movement of operands. First, the switching network limits the amount of connectivity. Second, operands cannot traverse function units since acyclical data-flow graphs prevent an operation which recreates the same operand. Third, while operands can traverse memory devices, they will suffer the delay of a cycle. These limitations combined with the constraints derived from the simultaneous transfer of multiple operands account for a majority of the cycles in a schedule. Useful scheduling bounds may be derived by formalizing the minimal delay of a single operand traversing from specific locations. The minimal delay between a given output port, θ, and input port, ϕ, shall be denoted as $\tau(\phi, \theta)$. These delays may be extracted from the $\tau_{\phi, x}(\theta)$ sets introduced in Section 4.4.2 by identifying the minimal x value where $\theta \in \tau_{\phi, x}(\theta)$.

Each operation is assigned an ALAP bound based on the time it takes to route the resulting operand to its specified position. These bounds start with the set of operations which produce control signals ($\Pi = \emptyset$). The ALAP bound for these operands is set to the cycle limit, “last cycle”, to reflect the fact that any signal may constitute the final operation. Next, those operations whose resulting operand appears in the final state specification are evaluated. The bound for these operations is computed from the delay required to store the resulting operand, p_k, into the memory devices specified in the final state set. Since the user can specify multiple, alternative final states, each state is evaluated separately to determine which provides the smallest delay. Still, the evaluation of each final state must ensure that the operand is sent to all of that state’s specified memory devices, but

the operand is allowed to utilize the input port which provides quickest route to that memory device, as in:

$$ALAP(e) = \text{last cycle} - \left\{ \min_{S_j \in S_f(V)} \left[\max_{v_{i,k} \in S_j} \left(\min_{\phi \in \Phi_i} \tau(\phi, \theta) \right) \right] \right\}$$

The ALAP bound for the remaining set of operations determines that last cycle on which the operation result could be used as an input operand. This is formulated as:

$$ALAP(e_1) = \max_{e_2 \in E | p_1 \in \Pi} [ALAP(e_2) - \tau(\phi_{e_2, p_1}, \theta)]$$

where the notation ϕ_{e_i, p_k} identifies the input port of c_j (where $c_j = C(\theta_i)$) associated with operand p_k.

These ALAP bounds differ dramatically from the computation of traditional ALAP bounds. One difference is the fact that this proposed method utilizes resource constraints in terms of routing restrictions instead the number functional unit types. But, the central difference stems from the accommodation of the recomputation of an operand. Because of this capacity, the ALAP bound above must maximize the ALAP bound instead of the minimizing it. Use of the minimal bound indicates the last cycle on which an operation could fulfill the timing requirements of all of the subsequent operations. Since there is no guarantee that this operation result will be used for all of the subsequent operations, the ALAP bound is based on the result which can fulfill the requirements for one of the children (subsequent) operations. This policy means that operations may have an ALAP bound which is later than the ALAP bound of a child operation as shown in Figure 5.4. Fortunately, this policy accommodates the fact that not every operation must be executed since any operation may have an alternative.

While powerful, these bounds suffer from the fact that they do not model the constraints resulting from multiple communications acting in concert. More exact

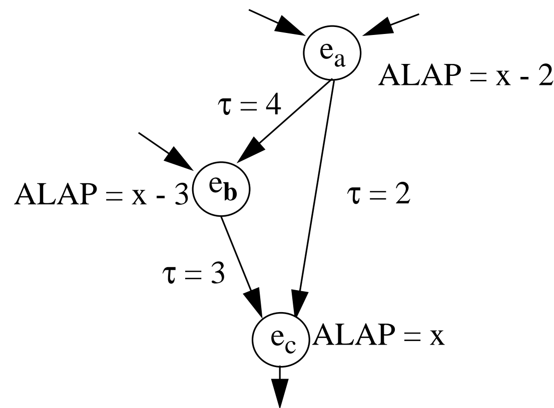


Figure 5.4 Fluctuating ALAP bounds due to operand fanout.

bounds could be derived by partitioning the data flow graph and generating ALAP bound for each portion. Data-flow graphs which produce multiple final operands are of particular interest. If the computation of each of the final operands were described as a separate data-flow graph, the interaction of the intermediate operands could be studied more closely. By executing a reachable state search and then reviewing the generated states, a set of ALAP bounds may be derived for each of these new data-flow graphs. This review of the reachable states would determine the last cycle on which each operand actually aided the completion of the data-flow graph and specify this cycle as a tentative ALAP bound. These tentative bounds are then reconfigured to accommodate the final cycle for the data-flow portions working in concert. While promising, this technique poses a number of challenges in order to extract an operand's death from a review of a reachable state search. Since these challenges have not been presently addressed, this topic is a suitable candidate for future research.

5.3.2 Scheduling Results

A minimum-cycle scheduler as described in Section 4.3.2 was constructed. This implementation provided a series of command line options to explore different characteristics of the solution set. This permitted a series of questions to

be asked of each data-flow/data-path pair: 1) what is the minimal number of cycles required for executing the data-flow regardless to register size constraints, 2) what are the minimal register sizes which permit this cycle bound to be met, 3) how may heuristics be used to reduce the execution time, yet still find a solution complying with the register constraint and cycle bound, and 4) extract a detailed schedule report using this last technique. If the problem specification proved too large for exact analysis, heuristics were employed to approximate a minimal number of cycles followed by steps 2 through 4 to minimize register constraints and extract a solution. For these later cases, both the "death approximation" and "maximum utility" heuristics from Section 4.4.3 and Section 4.4.4 were employed.

Minimal schedule identification

Table 5.4 lists the results for finding the optimal, minimal-cycle schedules as determined through the use of a bounded minimum-cycle scheduler. Surprisingly, the use of the pipeline multiplier in the tms32010-based designs did not have a negative effect on the schedule length for either of the 3x3_det benchmarks or the diff_eq/single-bus benchmark. Exact results for the suite of ewf benchmarks were not produced because our BDD package began swapping. In addition to its increased complexity, the structure of ewf data-flow graph results in a number of fluctuating ALAP bounds. While this problem may be overcome by the use of breadth-first BDD algorithms, it provides an estimate for the limitations for this proposed system.

The execution times listed in Table 5.4 demonstrate the relative merit of the memory mapping optimization (Section 4.4.2) and the ALAP bounds (Section 5.3.1) in terms of each benchmark.[1] The quality of the resulting schedules are not modified by either of these techniques since no heuristic pruning is involved. Still,

Table 5.4: Exact scheduling results

Data Flow	Data Path			# Cycles	Run Time (sec)				
					Neither	Mem	F-ALAP	D-ALAP	Both
diff eq	tms32010	single cycle mult.	1 bus	17	250.2	125.5	240.4	195.2	96.7
			2 bus	12	22.9	22.8	23.0	18.6	18.5
		pipeline mult	1 bus	17	565.7	350.2	539.6	301.8	191.7
			2 bus	13	109.4	109.1	97.2	51.2	36.7
	dual register file			12	15.2	15.1	16.4	16.1	16.2
3x3 det	tms32010	single cycle mult.	1 bus	20	4,745	1,487	3,627	2,099	687
			2 bus	13	267	279	188	93	94
		pipeline mult	1 bus	20	11,215	5,412	8,109	2,625	1,396
			2 bus	13	798	813	475	77	78
	dual register file			22	415	420	412	383	398
dhrc	tms32010	single cycle mult.	1 bus	22	2,508	486	2,259	790	168
			2 bus	19	1,051	1,053	664	334	277
		pipeline mult	1 bus	23	16,045	2,353	13,242	1,180	274
			2 bus	21	1,534	1,561	1,032	325	327
	dual register file			19	103	106	107	38	38

the complexity of the reachable state analysis and the resulting run times are very dependent upon the pruning techniques employed.

The first column, "Neither" lists the run times resulting from executing the reachable state analysis utilizing every optimization except for memory mapping and ALAP bounds. The "Mem" column lists the run times when memory mappings were optimized. Substantial benefits are visible in data paths which contained memory devices dedicated to a function unit input such as the "t latch" in the single bus tms32010 designs. The benefit for each of these single bus benchmarks is relatively uniform for each data-flow graph. This result is expected since the reduction to the state space is dependent upon the data-flow graph's

1. Efforts to run the automata without the other optimizations, such as dynamic relation construction (Section 4.4.3) or latch relation optimization (Section 4.5.2), quickly cause the reachable states analysis to become intractable.

operation set. The occasional increase in execution times associated with the other benchmarks reflects the overhead due to the computation of the τ sets.

The results for the proposed ALAP bounds are compared against two sets of run times. In addition, to the run times listed under "Neither", a set of run times are listed corresponding to bounding the reachable state analysis with ALAP derived solely from the function unit resources, "F-ALAP". The results from using this traditional bound are mixed. Examples containing the dual-register file data path or the diff_eq data-flow graph show only slight improvements in run times, if any. By contrast, the ALAP bounds derived from the complete set of data path resources, "D-ALAP", demonstrate a consistent set of improvements.

Figure 5.5 demonstrates how these benefits are realized for a particular example. Here we see the constant growth in the set of reachable states, until the set is intersected with the set of final states. By employing the ALAP bounds, the size of the reachable state set is reduced as the analysis approaches the anticipated final clock cycle as elements are removed which have no impact on the solution set. Finally, when these techniques are combined with the memory mapping (corresponding run times are in listed in column "Both") the size over all cycles is limited by reducing the set of states from which the reachable state analysis must consider.

Scheduling and register constraints

Table 5.5 is a compilation of the best known schedules subject to register constraints. The additional scheduling results were derived by loosely constrained heuristics. These results are contrasted with those derived from traditional scheduling using data-path estimates. No published results are available for the

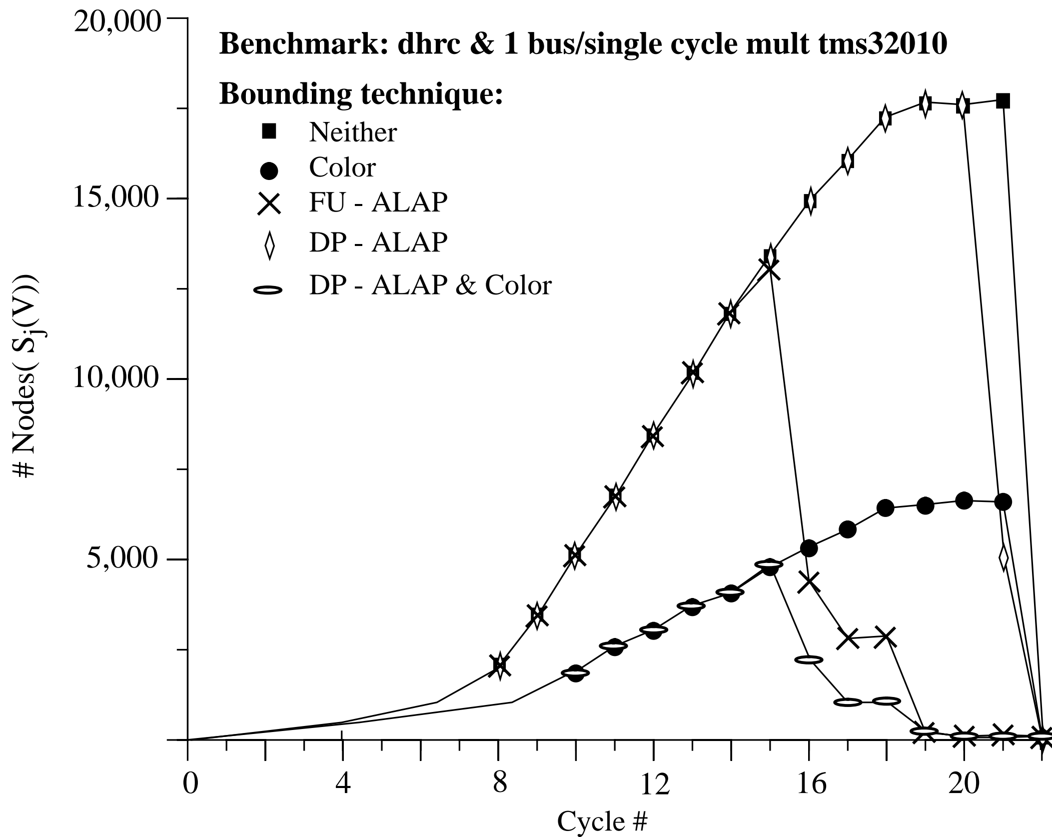


Figure 5.5 Cycle by cycle comparison of performance

dual register file design. A comparison of the available results underscores the additional delay mandated by practical, pre-existing designs.

Each value listed in the column "Register Size" corresponds to the minimal register size which met the minimal cycle constraint. In the case of the dual-register-file data path, the size of the multiplier register file proceeds the adder register file. Asterisks indicate when the register size matches the register requirements of either the initial or final state specification. Multiple factors combine to determine whether a schedule may fit within such a minimal register size including: the data path, data-flow graph, and the availability of extra latches, such as those found in the set of tms-based data paths. For example, the additional pipeline latch reduces the register requirements for the "diff_eq" benchmark.

The run times listed in the right column of Table 5.5 correspond to running the application with individually tuned heuristics. The interaction of the data path, data flow, register constraint, and selected heuristics cause a high variance in the run times. Most important, is the dramatic rise in run times for the benchmarks

Table 5.5: Heuristic schedules results

Data Flow	Data Path			# Cycles: Traditional Scheduling[a]	# Cycles: Data-Path Constrained	Register Size	Run Time (sec)
diff eq	tms32010	single cycle mult.	1 bus	7	17	4	16.44
			2 bus		12	4	3.41
		pipeline mult	1 bus	8	17	3*	23.54
			2 bus		13	4	11.80
	dual register file			-	12	3, 3	15.69
3x3 det	tms32010	single cycle mult.	1 bus	10	20	9*	142.49
			2 bus		13	9*	107.79
		pipeline mult	1 bus	12	20	9*	221.72
			2 bus		13	9*	86.45
	dual register file			-	22	10, 2	99.68
dhrc	tms32010	single cycle mult.	1 bus	10	22	5*	24.00
			2 bus		19	5*	5.78
		pipeline mult	1 bus	12	23	5*	17.34
			2 bus		21	5*	10.51
	dual register file			-	19	3, 3	4.52
ewf	tms32010	single cycle mult.	1 bus	27	60	10	918.92
			2 bus		41	9	410.24
		pipeline mult	1 bus	28	60	10	766.05
			2 bus		41	9	397.48
	dual register file			-	43	2, 9	4,594.63

a. Scheduled with no bus constraint.

using the ewf data-flow graphs. This rise is due to the poor ALAP bounds resulting from the data-flow graph structure as well as the increased complexity of the operation set.

Figure 5.6 displays a representative solution produced by this scheduling technique. This example displays the results of mapping the diff_eq benchmark on the single-bus, single-cycle multiplier tms32010-based data path with a register constraint of three. While the data flow which is listed in the figure has already

Differential Equation mapping

Differential Equation specification

op1 = dx * x;
op2 = c5 * y;
op3 = c3 * z;
op4 = op1 * op2;
op5 = dx * op3;
op6 = x - op4;
x’ = op6 - op5;
y’ = dx + y;
z’ = op1 + z;
cntrl <- y’ < a;

State	t	p	Acc	Reg	ROM	y’ < a
S0				{x y z}	{a dx c3 c5}	
S1	c5	c5	c5	{x y z}	{a dx c3 c5}	
S2	y	op2	y	{x y z}	{a dx c3 c5}	
S3	dx	op2	y’	{x y z}	{a dx c3 c5}	
S4	dx	op2	y’	{x y z}	{a dx c3 c5}	X
S5	dx	y’	op2	{x z y’}	{a dx c3 c5}	
S6	x	op1	op2	{z y’}	{a dx c3 c5}	
S7	op2	x	op1	{z y’}	{a dx c3 c5}	
S8	op2	op4	x	{z y’ op1}	{a dx c3 c5}	
S9	c3	op4	x	{z y’ op1}	{a dx c3 c5}	
S10	c3	op3	op10	{z y’ op1}	{a dx c3 c5}	
S11	op10	op10	op3	{z y’ op1}	{a dx c3 c5}	
S12	op3	op10	op10	{z y’ op1}	{a dx c3 c5}	
S13	op3	op7	op10	{z y’ op1}	{a dx c3 c5}	
S14	op3	z	x’	{z y’ op1}	{a dx c3 c5}	
S15	x’	x’	z	{x’ y’ op1}	{a dx c3 c5}	
S16	x’	x’	z’	{x’ y’}	{a dx c3 c5}	
S17	x‘	x‘	x‘	{x’ y’ z’}	{a dx c3 c5}	

Figure 5.6 Resulting schedule and operand mappings.

merged two redundant operations into a single operation resulting in op1, this merger was left out of the benchmark specification requiring the application to perform the merger. The ability to create such schedules and memory binding is essential for providing the constraints used in the previous sections.

Chapter 6

Evaluating Multiple Networks

The capacity to specify multiple network topologies is extraordinary beneficial when comparing competing designs. In this case, alternative but similar alterations to a data path are evaluated. The capacity of each data path are modeled as exclusive topologies, but all topologies may be modeled concurrently. This concurrent analysis is very effective when topologies share a common functionality differing only in terms of delay or power. Topologies which contain a limited amount of functional difference can receive a substantial benefit from concurrent analysis. Initially, this chapter will review the penalties for such functional differences. Later, the incorporation and evaluation of timing information are presented.

6.1 Scheduling on Multiple Data Paths

The ability to evaluate multiple topologies is a very tempting proposal, but as the ensuing results will show, the efficiency derived from this technique is subject to many factors. Table 6.1 lists five data-path benchmarks which are constructed from the set of tms32010-based designs presented in Section 5.1.2. Each benchmark specifies the set of data-path topologies which are included in the data-

Table 6.1: Five combined topology benchmarks.

Data-Path Topologies	1 bus & 1-cyc. mult.	1 bus & pipe. mult.	2 bus & 1-cyc. mult.	2 bus & pipe. mult.
1 bus	X	X		
1-cyc. mult.	X		X	
2 bus			X	X
pipe. mult.		X		X
all	X	X	X	X

path specification. The benchmarks are designed to incorporate a common theme, such as the pipeline multiplier, or as in the case of the last benchmark incorporate all four of the topologies. The run times resulting from a exact reachable state analysis on each of the benchmarks are specified in Table 6.2. Whereas the run times listed in Table 5.4 resulted from scheduling on a single data path, these run times are the result of analyzing two or more data paths concurrently. The time required to schedule each of the data-flow graphs is listed under the appropriate columns labeled “run time”. The ratio of these run times over the sum of the run times from individual analysis are listed in the column “efficiency”. Efficiency

Table 6.2: Relative efficiency of multiple topology analysis.

Data-Path Topologies	diff_eq		3x3_det		dhrc	
	run time	efficiency	run time	efficiency	run time	efficiency
1 bus	393.52	136.5%	338.18	16.2%	654.47	166.2%
1-cyc. mult.	54.51	47.3%	181.49	23.2%	676.31	151.7%
2 bus	88.45	160.2%	335.83	194.8%	492.95	81.6%
pipe. mult.	170.83	74.8%	187.90	12.7%	1,208.69	201.4%
all	125.99	36.7%	466.19	20.6%	278.83	26.7%

values which are less than 100% identifies those cases where concurrent analysis achieved superior results.

The relative efficiency varies greatly from example to example. Many times, a great savings results from the disparity of topology designs. While multiple topologies are analyzed, the results will identify the best topologies. If the topologies differ significantly, the performance of one topology can lower the maximum cycle bound dramatically. The ALAP bounds generated from this lower cycle bound help the performance of the system. This is why the 3x3_det benchmark was scheduled faster on the “1 bus” and “pipe. mult.” examples. Additionally, the diluting of performance which results from combining a large number of topologies helped to make the data path which combined all four topologies generate consistently superior results. At the same time, the mixture of topologies is not a panacea. For example, the quality of ALAP bounds suffer from the mixing of topologies, since the topology from which the cycle bound was derived might not be the same topology which provides the minimal operand transfers. Regardless of the ALAP bounds, the exact state sets which become merged and the efficiency of the system to merge them may incur substantial overhead. While one expects a BDD structure to merge state sets efficiently, the results show that many topology pairs defied this expectation.

6.2 Timing Evaluation

The logical extension to the modeling of operand movement is the modeling of operand transmission delay. Such delay information permits the quantization of timing penalties and cycle time requirements. Such analysis, as shown in the initial example presented in Section 1.1, gives the designer a critical advantage when reviewing changes to the data path or data-flow map. This section will detail how such clock cycle modeling may be incorporated into the automata model which

has been presented. After a discussion of timing model utilized for measuring the cycle time, the expansion of the automata model is discussed. The section concludes with a review of some experimental results.

6.2.1 Timing model

Delays through a data path are incurred as operands traverses wires, switching elements, combinational logic, until finally reaching a destination: a storage element or the controller. The term *operand path* will be used to define any path over a consistent network topology which starts at either an external input port or memory unit, traverses any number of function units and multiplexers, and ends at the input port of a memory unit or a function unit's signal output port. Given a data path, an upper bound may be placed on the maximum data-path delay of a network topology by enumerating each operand path and computing the associated delay. While the cycle time of the data path need not exceed the maximum of these delays, it could be considerably less. This is because the set of paths through a data path may contain a number of *false paths* which will never be utilized. The cycle time can be set unnecessarily high if it is based on one of these false paths. False paths can occur for a variety of reasons: an operand path may require conflicting control settings making it infeasible, faster redundant paths through the data path may alleviate the need to use a more costly path, or operands are never transferred along a specific path even though it is feasible and unique. After detailing the delay models, I shall demonstrate how these false paths are avoided.

Component Delays

The data-path specification for each component c_i is expanded to include timing delays. The format for expressing these delays is based on the behavior of the component being modeled. A set of *read delays* is defined for each operand generating component (external inputs and memory devices) where each delay is

dependent upon the output port and control settings. Similarly a set of *throughput delay*s is defined for components which process operands (multiplexers and function units), but these delays are dependent upon the input port, output port, and control settings. Finally, a set of *load delays* are defined for memory devices which are solely defined by the input port. In general, the delay for a component c_i may be retrieved by the function $\Delta_i(\vec{\sigma}, \phi, \theta)$. These delays may contain a special delay of "$-\infty$" to denote that an input port is not used for some output and control combinations. Additionally, the appropriate passing parameters of Δ_i are set to null when retrieving read or load delays.

While the control lines feeding the data-path components are assumed to be defined for the entire active clock cycle, the delay of transporting the generated signals to the controller must be modeled. Presently, the delay required to produce a signal and route it to the controller is lumped into a single throughput delay. If this model is insufficient, experience has shown that the interconnection delay for signals can be adequately modeled after some minor modifications to the data-path specification. This can be achieved by modeling the signals as operands which are connected to special *controller* components.

Wire delays

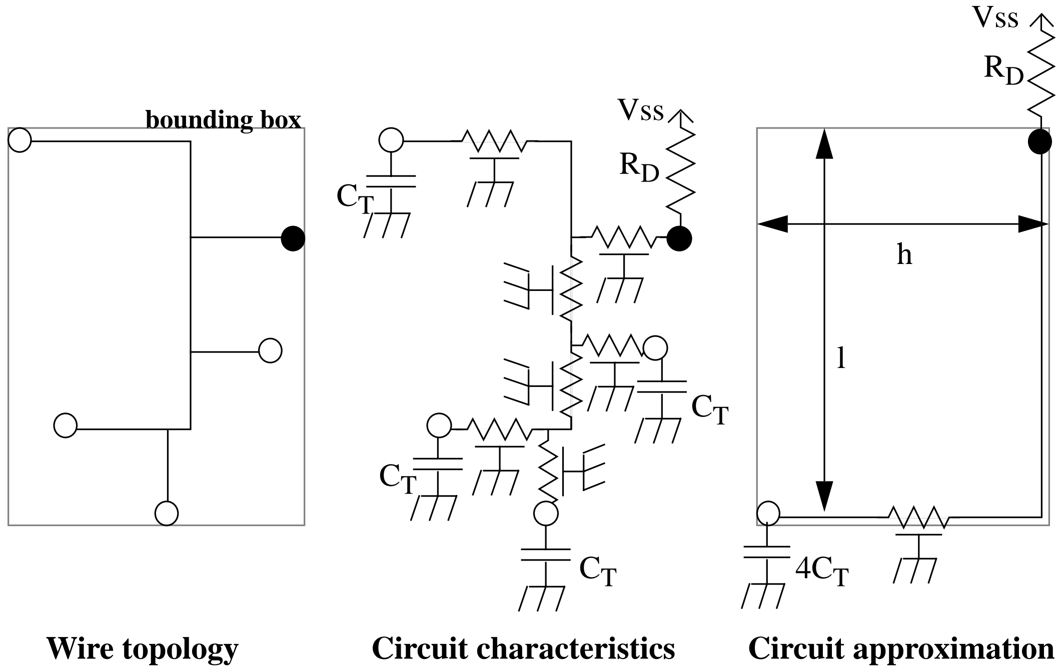
Propagation delays occur as operands are transferred through wires. These delays are dependent upon the physical routing of the wire as well as the portion of the wire which the operand traverses. Presently, my system supports propagation delays which are either computed dynamically or submitted in a pre-computed format. The availability of the dynamic computation allows the user to preview the effect of a network modifications without undergoing lengthy interconnection modelling, but the thoroughness of these calculations are limited by the input specification. Whenever more detailed analysis is required, the propagation delays

can be computed by the proper application and then submitted with the data-path input.

These dynamic propagation delay estimations are derived from a wire's physical properties as described in each network topology. While the data-path specification does not currently handle the specification of routing information, it does allow layout information to be associated with the set of input and output ports. This positional information is utilized to estimate the physical layout of each wire. This layout information is combined with the wire's circuit properties to compute the propagation delay. But, only a single conservative delay is computed for each wire topology instead of a delay for each path across the wire. While extensions to the propagation delay model are feasible, it is still preferred to precompute the delay models externally by tools capable of modeling the actual routing information.

The transmission delay is dependent upon the distributed RC (resistive capacitance) of the wire, the resistance of the driver, and the cumulative capacitance of the taps. Wire layout is approximated by the bounding box which contains the wire's set of drivers and taps. The wire length and its associated distributed RC are estimated as a function of half the perimeter of the Manhattan bounding box. The driver resistance is assumed to be constant. The cumulative tap capacitance is a function of the number of connecting points of the wire. The 50% rise time, appropriate for static CMOS circuits, is computed from these three values using the approximation in Bakoglu [5] p.204. Figure 6.1 displays an example wire configuration, its actual physical properties, the estimated model, and its timing equation.

Alterations to a network topology effect the wire delay by removing and inserting elements from a wire's set of connection points. Proper modeling



$$R_w = (l+h)\,R_{unit},\ \ C_w = (l+h)\,C_{unit}\ ,\ \ C_L = (\sum_{taps} C_T)$$
$$t_{50\%} = 0.4R_wC_w + 0.7\,(2.3)\ (R_DC_w + R_DC_L + R_wC_L)$$

Figure 6.1 Wire Delay Model

requires instantiating each set of viable connection points to determine the range of propagation delays for each wire. Unfortunately, these sets of connection points can not be easily extracted from the data-path specification format. The assembling of such a list must be performed by analyzing the topology conditions under which any input port can connect to an output port, θ_i, such as:

$$\bigcap_{\phi_j \in \Phi} \left[\exists_{\psi_{j,k} \in \Psi_j | \theta_{j,k} = \theta_i} [\,(\Upsilon_{j,k} \cap \phi_j) \cup ((\Upsilon - \Upsilon_{j,k}) \cap \bar{\phi}_j)\,] \right]$$

A delay can be subsequently computed for each unique set of input ports, and then each delay and its associated network topologies is stored with each input port. The function $\Delta_i(\upsilon)$ is utilized to retrieve the delay of input port ϕ_i for a given network topology.

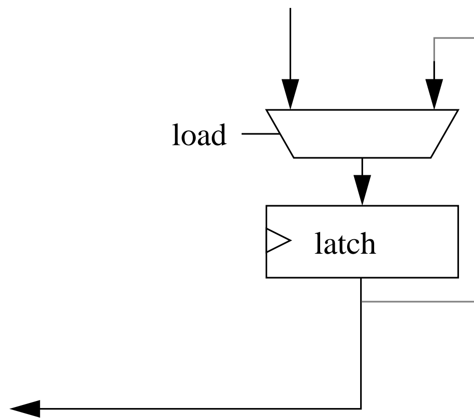


Figure 6.2 Latch's output wire captures interconnection and functional behavior.

Revisiting restricted component set

It is useful to review the compatibility of a timing model and set of component behavioral types from Section 4.1.1. The set behavioral types was restricted in order to define a minimal set of base components. While the complexity of the automata construction is reduced dramatically by adopting this minimal component set, this set increases the complexity for the timing specification. Therefore, the following modifications are made:

The propagation delay along a wire to an input port may be pre-specified as zero even when propagation delays are dynamically computed. This alteration is required since wires are used for two different purposes. While they are primarily used to model the interconnection structure of the data path, they are also used to join basic components when modeling compound components. It is important that wires used for compound components do not complicate the timing model by introducing nonexistent delays. Therefore wires which are used solely for this purpose can be identified and assigned a zero delay for all input ports. Additionally, any wire, such as the one in Figure 6.2, may fan out to describe items both in the interconnection structure and compound components. These wires will have timing delays, but the dynamic computations of these delays must distinguish

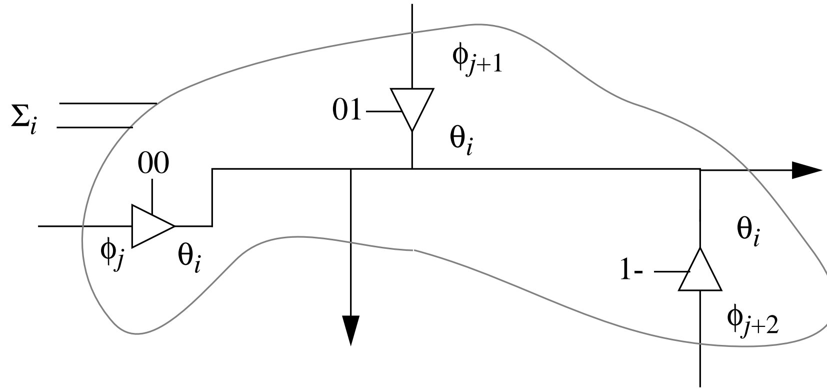


Figure 6.3 Multiplexer component variation: "switching element set"

which connection points should be omitted from the delay computation. Furthermore, the input ports which are not used to model the interconnect structure should have a zero propagation delay. Therefore, accommodations have been made to identify these special connections.

Additionally, the multiplexer description is expanded to model the switching elements used to drive buses. While functionally equivalent to a multiplexer, a set of switching elements have multiple output ports from which a single wire may be driven. Each of these output ports will be directly associated with an appropriate input port and will share a common name, as shown in Figure 6.3. This association with an input port permits the system to identify the set of topologies under which an input, ϕ_i, and its output are not required, $\Upsilon_i = \bigcup_{\psi_{i,j} \in \Psi_i} (\Upsilon_{i,j})$. These conditions are then used to determine the minimal set of drivers which must exist for a given network topology, from which the positional and capacitive effect can be added to the computation of $\Delta_i(\upsilon)$. This formulation shall be written in terms of the associated input ports, as in:

$$\bigcap_{\phi_j \in \Phi_i} [(\Upsilon_j \cap \phi_j) \cup ((\Upsilon - \Upsilon_j) \cap \bar{\phi}_j)]$$

since all of the output ports in a set of switching elements share a common name. This utilization of a common output port name alleviates any changes to the Ψ_i

specification to accommodate an input port being potentially driven by multiple output ports for a given network topology. The deficiency of this nomenclature is the difficulty it presents in specifying delays associated with a given output port. But, this can only be of interest to applications which use predefined timing delays since dynamically computed ones are based solely on the network topology. In the case of these predefined timing delays, the output-port-dependent delays may be restates in terms of the throughput delay of the switching element.

Computing delays through the network

The computation of delays through the network utilizes techniques standard to propagating delays through a graph. This analysis cumulates successive delays that emanate from primary inputs (external inputs or memory devices) until reaching a primary output (memory devices or output port for signals). When multiple delays merge at a single node, the maximum of the compatible delays is selected. This straight forward task is complicated by the fact that multiple delays are propagated along a single path. The multiple network topologies means that a single path may contain many network-dependent delays. Additionally, delays which converge at a single device can use disjoint control settings, as in a multiplexer, which introduces control-dependent delays. Therefore, the delay at a given point in the data path is dependent upon both the network and control settings. This relation between a delay and the network and control dependencies is maintained as an ordered pair. I choose create a set of these pairs Δ'_i for each input and output port where each element, $\delta'_{i,j}$ is expressed as $(t_{i,j}, \Lambda_{i,j}(\Upsilon, \Sigma))$. Additionally, those false paths arising from conflicting control settings shall be automatically eliminated as a resulting of maintaining the control requirements. The delay for an input, ϕ_i, is defined by the output port, θ, to which it is connected and the

propagation delay. As formulated below, if there is no output port associated with the input port, the delay is infinite.

$$\Delta'_{\phi_i}(\sigma, \upsilon) = \begin{cases} \infty & \text{where } \varnothing = \theta \\ (\Delta_{\phi_i}(\phi, \theta, \upsilon) + \Delta'_{\theta}(\sigma, \upsilon)) & \text{otherwise} \end{cases}$$

The notation $\Delta_i(\phi, \theta, \sigma) + \Delta'_j(\sigma, \upsilon)$ represents the summation of the timing values for common topology and control settings. This notation allows the delay for an output port to be defined in terms of the slowest path, as in:

$$\Delta'_{\theta}(\sigma, \upsilon) = \left(\max_{\phi \in \Phi_i} [\Delta_i(\phi, \theta, \sigma) + \Delta'_{\phi}(\sigma, \upsilon)] \right)$$

Recall, that an input port which is not used for a given control setting will have a delay of $-\infty$ with which to counteract any input-port delays of ∞.

6.2.2 Using the automata

Those false paths which are infeasible due to control conflicts are removed during the computation of delays for each operand path. Still, a large number of feasible operand paths exist which may never be used to complete the execution of a data-flow graph. The execution of the automata $N(\Upsilon, V, V')$ may be used to distinguish the paths which are used from the false paths. Furthermore, the relation may be expanded to incorporate timing information, denoted as $N(\Upsilon, \Sigma, V, V', \Delta°)$, with which to grade the solution set and identify the data-flow map which requires the minimal clock-cycle time. This section will demonstrate how the timing values are introduced to the automata model and used.

Constructing the automata

After computing the delays for the set of operand paths, delays are defined for the set of input ports to the memory devices and the output ports dedicated to the transmission of control signals, Θ''. In order to distinguish the false paths of this system, the timing information for each operand transfer is integrated into the

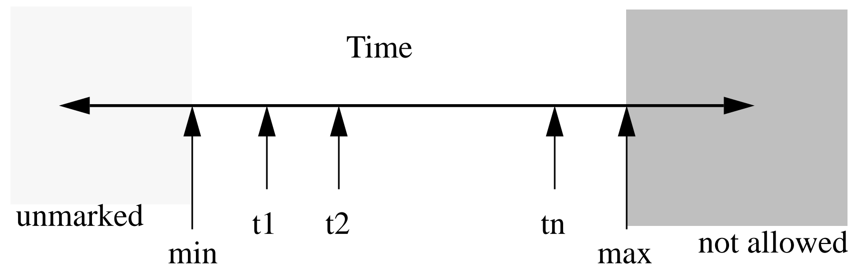


Figure 6.4 Partitioned time line.

automata. But this technique presents a problem: how to integrate timing values composed over the set of real numbers with an automata composed of discreet components?

The solution lies in transforming the set of timing values into a set of discreet values. While one may generate a discrete value for each unique timing value, this set could be quite large. Instead, a set of user defined timing ranges are utilized to cluster the timing values. In addition to such timing ranges, the user may define a maximum and minimum timing bound with which to limit the timing analysis, as shown in Figure 6.4. Any delays exceeding the maximum bound are defined as illegal resulting in an unsuccessful transfer of an operand. All delays which are less than the lower bound are considered legal but simply do not require timing analysis. In the absence of such user defined bounds, the bounds are automatically defined by the maximum and minimum delays computed for the data path.

The range of timing values between these timing bounds may be partitioned into a number of timing ranges. If the user simply wants to verify that a clock cycle exists below some specified time, they may specify equivalent timing bounds to indicate that no timing analysis beyond the exclusion of operand transfers is required. Otherwise, $\Delta^\circ = \delta^\circ_1 \times \delta^\circ_2 \times \ldots \times \delta^\circ_n$ is defined as the set of timing combinations where each Boolean δ°_i symbolically represent one of the timing ranges. Moreover, the use of disjoint timing variables allows the description of

multiple timing. This feature is useful when describing the set of timing delays associated with a set of operand transfers which occur in a single cycle. More importantly, it permits all of the timing delays stemming from all operand transfers over all cycles to be represented in single format. This format permits the user to identify the clock cycle time by just reviewing the set of final states instead of reviewing the history of state transitions.

The relation $\Lambda^\circ_i(\Delta^\circ, \Sigma, \Upsilon)$ is defined for each input port to a memory device and every output port used to transmit control signals. The relation, denotes the set of permissible control and network setting under which an operand may reach the input port. This relation is constructed directly from the set of $(t_{i,j}, \Lambda_{i,j}(\Upsilon, \Sigma))$ pairs, where $t_{i,j} < t_{max}$ and each $t_{i,j}$ is mapped into the set of times which all have the appropriate timing range, δ°_i, specified. This set of timing ranges are represented as $\Delta^\circ \cap \delta^\circ_i$.[1] The important idea is that the requirements on all of the timing ranges are left unspecified except for the timing range which is known to occur.

The alterations to the automata are device dependent. While the relation for external inputs is not effected by the introduction of the timing model, each $N_i(\Upsilon, \Sigma, V, V'_i)$ for i>0 is. For the group of control signals, the control setting which are used to transmit a signal out of an output port are labeled with timing information. This labeling is performed on an operand by operand basis. In order to match the pertinent output ports is handled by labeling each $\Lambda^\circ_i(\Delta^\circ, \Sigma, \Upsilon)$ with the proper output port as shown below:

$$N'_{1,k}(\Sigma, V) = v_{1,k} \cup \exists_{\Theta}\left[F_k(\Theta, \Sigma, V) \cap \bigcap_{\theta \in \Theta''}(\theta \cap \Lambda^\circ_\theta(\Delta^\circ, \Sigma, \Upsilon))\right]$$

1. If $t_{i,j}$ < minimum bound, Δ° is used in the relation.

For memory devices, the timing delays is added in terms of each $N'_{i,k}(\Sigma, V)$, since the delays are incurred only as operands are loaded into memory devices. This labeling is performed by replacing

$$[(F_k(\Theta, \Sigma, V) \cup M_k(\Theta, \Sigma, V)) \cap \Omega_\phi(\Theta)]$$

in EQ. 4.6 and EQ. 4.7 with

$$[(F_k(\Theta, \Sigma, V) \cup M_k(\Theta, \Sigma, V)) \cap \Omega_\phi(\Theta) \cap \Lambda^\circ{}_\phi(\Delta^\circ, \Sigma, \Upsilon)].$$

While this alteration leaves a zero timing delay associated with the act of maintaining an operand in a register file, it is assumed that this oversight will not effect the clock cycle.

Expanding operand non-transfers

The construction of each $N_{i,k}(\Upsilon, \Sigma, V, V'_i, \Delta^\circ)$ requires more than just the addition of timing information. The introduction of timing penalties creates a new problem which must be addressed. Previously, the omission of an operand from the next state occurred only if the operand could not be routed to the correct position, as described by $N'_{i,k}(\Upsilon, \Sigma, V)$. But with the introduction of timing restrictions, it may make sense not to load an operand even when it is feasible in order to forego the timing penalty. In case of the general relation, this change means that $\bar{v}'_{i,k}$ must be considered for all combinations and that it should be accompanied by the complete set of timing patterns to avoid restricting the timing constraints. This redefines EQ. 4.5 as:

$$N_{i,j,k}(\Upsilon, \Sigma, V, V'_i, \Delta^\circ) = (v'_{i,k} \cap N'_{i,k}(\Upsilon, \Sigma, V, \Delta^\circ)) \cup (\bar{v}'_{i,k} \cap (\Upsilon \times \Sigma \times S_j(\Upsilon, V) \times \Delta^\circ))$$

Because of the encoding scheme selected for latches, this restriction effects the latch transform equation by redefining the conditions under which the null variable may be loaded as all feasible conditions, as in:

$$N^\circ{}_{i,j,\mathrm{null}}(\Upsilon, \Sigma, V, V'_i, \Delta^\circ) = v'_{i,\mathrm{null}} \cap [\Sigma \times S_j(\Upsilon, V) \times \Delta^\circ]$$

This constraint causes problems for the optimization introduced in Section 4.5.2 which reduced the conditions under which the null operand was considered. The loss of this reduction technique is a serious set back since it is a powerful tool with which to make the reachable state analysis practical. But this can be overcome by the following observation: the set of solutions that this analysis overlooks are characterized by an increased use of the null operand. The benefits of both techniques may be found by running the reachable state analysis in two phases. In the first phase, the automata is run in the absence of timing information utilizing all of the reduction techniques and the reviewed to generate the series of states, $S^{\circ}_{j}(\Upsilon, V')$, essential to finding the solution set, as shown in Section 4.3.2. Each of these state sets are then transformed into state restrictions $\chi_j(\Upsilon, V')$ to be applied on the second phase of the reachable state analysis which includes timing analysis. This reduces this second phase to a data-path routing problem. The use of these state restrictions significantly reduces the set of feasible states resulting from combinations of non-null operands (P' and P'') enough to mitigate the effect of the increased presence of the null operands. To accommodate this increase of null operands, each $\chi_j(\Upsilon, V')$ is constructed by introducing the null operand into each possible latch encoding from which it is missing. For a data path with a single latch, such a change would appear as:

$$\chi_j(\Upsilon, V') = S^{\circ}_{j}(\Upsilon, V') \cup \left[S^{\circ}_{j}(\Upsilon, V') \big|_{\bar{v}_{i,\,\mathrm{null}}} \cap v_{i,\,\mathrm{null}} \right]$$

The complications of converting the state encodings for multiple latches is alleviated by the choice of operand encodings for the latch. By cofactoring the “null bits” for each latch encoding, the set of states are expanded to accommodate null operands in any location for which they were previously forbidden in a single BDD operation whose complexity is bounded by the size of $S^{\circ}_{j}(\Upsilon, V')$.

Analyzing the results

Having defined how to generate a new set of relations, $N_i(\Upsilon, \Sigma, V, V'_i, \Delta°)$ the execution and use of these relations may now be presented. The initial state may be redefined over this set of timing combinations as $S_0(\Upsilon, V, \Delta°) = S_0(\Upsilon, V) \times \Delta°$ to indicate that any combination of timing values is valid. Restrictions are then placed on which timing variables must be present with each succeeding state transitions. When the set of states intersects the set of final states, the set of timing restrictions associated with each possible solution may be formulated as $\exists_{\upsilon \in \Upsilon} \exists_{v \in V} [S_j(\Upsilon, V, \Delta°) \cap S_f(\Upsilon, V)]$. These timing constraints may, in turn, be applied to additional data-flow graphs to determine how their timing constraints interact. Regardless as to the number of data-flow graphs which are analyzed, the system will wish to identify the minimum cycle time. As opposed to the use of network topologies where the relative value of a topology is difficult to formulate, timing ranges have an intrinsic preferential order. This order is utilized to automatically identify the minimum clock cycle time and generate the associated data-flow maps. By successively restricting the timing ranges which are allowed, the set of timing constraints may be pruned until only no constraints are feasible. The minimum clock cycle time is defined as the maximum timing range which causes the timing restriction to become invalid. Assuming that the timing partitions are arranged in the order of their timing ranges, the timing requirements may be reduced by one timing range at a time, as in

$$\bigcap_{i=n}^{1} \left[\overline{\delta°}_i \cap \exists_{\upsilon \in \Upsilon} \exists_{v \in V} [S_j(\Upsilon, V, \Delta°) \cap S_f(\Upsilon, V)] \right]$$

Each successive $\overline{\delta°}_i$ omits the set of timing constraints which use an operand transfer from the range $\delta°_i$. The $\overline{\delta°}_i$ which causes this set to become null is the minimal timing range, $\delta°_{min}$, required for the completion of the automata. A mini-

mal timing restriction may be generated from this timing range and applied to the solution set as:

$$\exists_{\delta^\circ \in \Delta^\circ}\left[S_j(\Upsilon, V, \Delta^\circ) \cap \bigcap_{i = \min + 1}^{n} \overline{\delta^\circ}_i\right] \cap S_f(\Upsilon, V)$$

This set of remaining solutions define the feasible network topologies and their associated final state which meet this minimal cycle time bound and complete in j cycles.

6.2.3 Experimental Results

The efficiency of such timing techniques is demonstrated by analyzing a set of benchmarks. All of the benchmarks utilize the data path represented in Figure 6.5. This data path, shown here in terms of compound components, is functionally equivalent to the tms32010-based data path with a pipelined multiplier and a single global bus presented in Section 5.1.2. The single cycle multiplier was not utilized for this section since its timing delays dominated the timing analysis and skewed the results. As opposed to the previous examples, eight different topologies stemming from three disjoint alterations are considered. Two of these alteration concerned whether switching elements M1 or M2 would be realized by a multiplexer or bus switch model. The third alteration considered including the direct line between the register file and the multiplier shown in dashed lines. The effects that these alterations have on the propagation delays are modeled by the dynamic computation presented in Section 6.2.1. The floorplan used to compute these values are displayed at the bottom of Figure 6.5. This positional information was combined with the physical properties listed Table 6.3 to generate the estimated delays. The same delay computation was used to generate the results described in the introductory example in Section 1.1. For the data path considered here, the propagation delays along a single wire ran as high as 2.32ns.

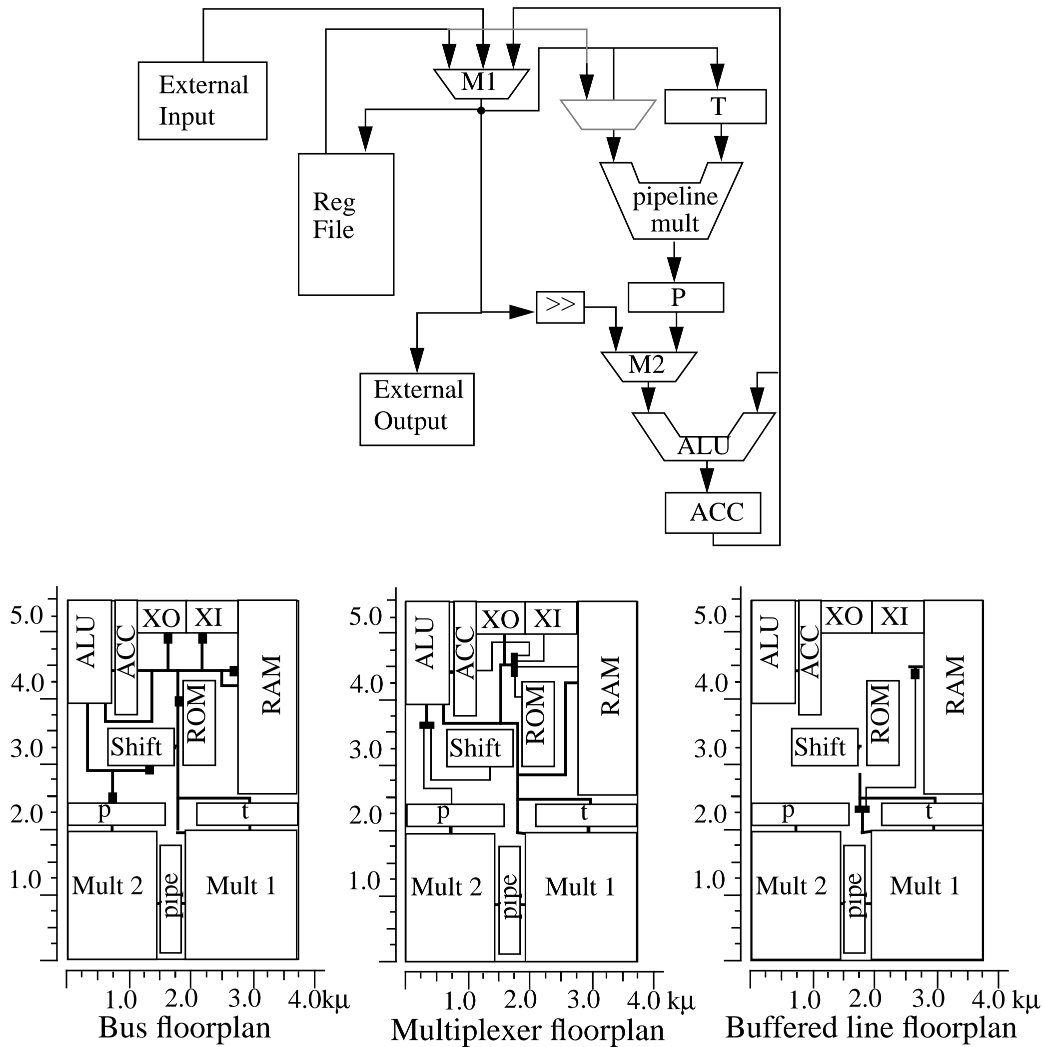


Figure 6.5 TMS32010 based data-path model and floorplans

Table 6.3: Physical Parameters

Parameter	Symbol	Value
Output port resistance	Rd	2 kΩ
Output port capacitance	Cd	60 ff
Input port capacitance	Ct	20 ff
Wire resistance	Rw	0.166 Ω/micron
Wire capacitance	Cw	0.176 ff/micron

The four data-flow graphs discussed in Section 5.1.2 were analyzed on this

data path. Initially these benchmarks were analyzed in terms of a data-path routing problem. Such routing problems, as discussed in Section 5.1, utilize a predefined set of memory binding for operand storage. Attempts to minimize the cycle time in this constrained environment is another means of determining the minimal cycle time for a complete design. The binding that were provided for these runs were selected to ensure that they were valid assignments for any of the eight topologies. The analysis then reported the minimal cycle and the set of associated topologies on which the cycle time was realized.

The actual designs that are selected are highly dependent upon a number of issues: binding freedom, floorplans, and physical parameters of the topology. But these issues are not the focus of this work. Since the techniques which have been presented are based on providing the designer an efficient means to analyze and rate these issues, this section shall concentrate of the efficiency issues.

As shown in the chart in Figure 6.6, the efficiency of the system are dependent upon a number of factors. The efficiency is dependent upon the timing bounds which are selected. The selection of an appropriate maximum timing bound can restrict a number of feasible paths thereby reducing the complexity of the state transformations. Likewise a minimum timing bound helps to reduce the extra states which result when timing bits distinguish states which would previously be equivalent. Figure 6.6 utilizes four point styles to show the relative overhead encountered when either no bounds, just a minimum, just a maximum, or both a minimum and maximum (also described as a *tight bound*) are used. The actual timing bounds are listed with the data-flow graph key since the appropriate bounds fluctuated according to the data path being considered. An additional factor to this overhead is the number of allocated timing partitions which is shown along the x-axis. Despite these two factors, the primary factor is the data-flow graph which is

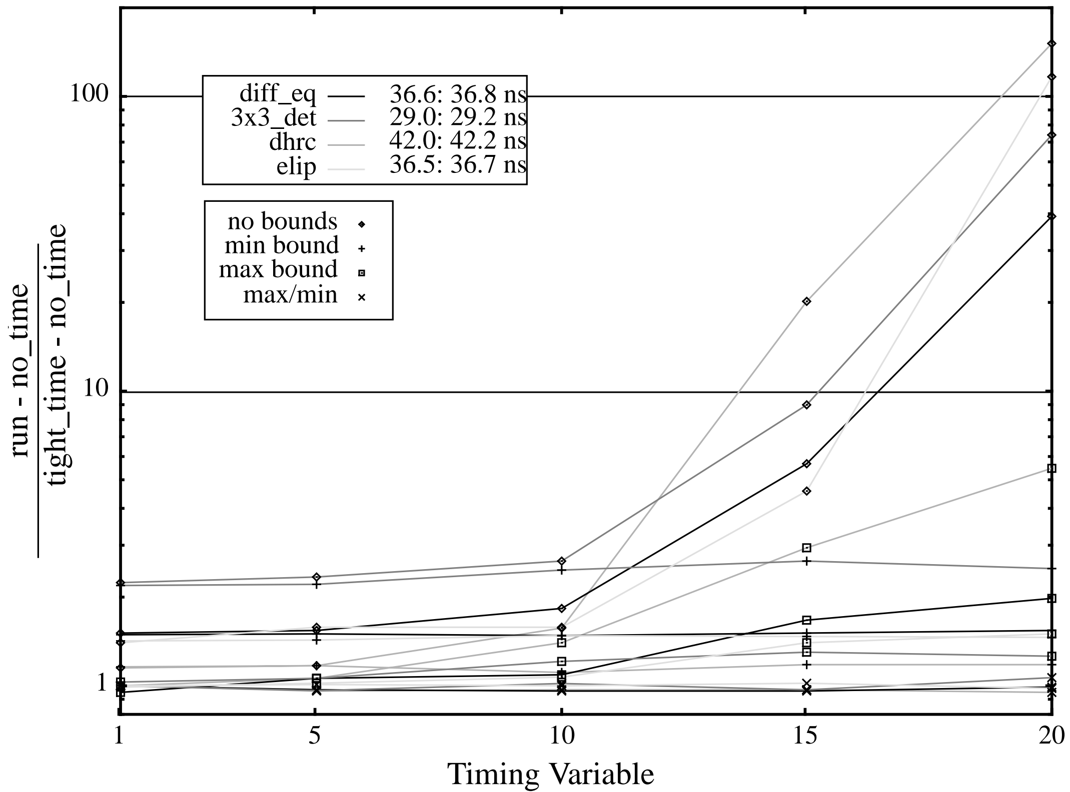


Figure 6.6 Timing analysis overhead in routing.

being analyzed. Obviously, execution times will reflect the length and complexity of the data-flow graph being analyzed. To mitigate this effect, each run time was divided by the complexity of the data-flow graph. This complexity was modeled by the increase in execution time when analyzing the data-path with a single timing variable and tight timing bounds. By dividing the increases in execution time over a data-path routing without timing analysis by this factor, the overhead rating utilized in this graph is derived.

Some general trends which are visible in the plot deserve attention. First off, the rate of overhead is relatively consistent across all data-flow graphs operating under similar timing bounds. This consistency is observable partially because of the logarithmic scale. Still, general trends are observable in terms of the different

timing bounds and timing variables. For example, use of both a minimum and maximum timing bound constraints the timing region so tightly that only a couple actual timing delays are ever marked, and therefore the efficiency was independent of the number of timing partitions (or timing markers). A more interesting phenomenon occurs when only a single bound is applied. The overhead is initially smaller if the applied bound limits the maximum timing value because paths which violate this bound are removed from the data-path analysis. But as the number of timing partitions increase, the overhead of a maximum timing bound overcomes that of a minimum timing bound. The source of this change of efficiency can be best understood when one realizes that the maximum and minimum bounds are practically equivalent. By selecting either a minimum or maximum bound, one is choosing to analyze either the top or bottom half of the timing spectrum. When one considers the bottom half, all communications are labeled since the top half are deemed infeasible. By contrast, an analysis of the top portion of the spectrum leaves all communication paths as feasible but only labels those paths with a substantial delay. Additionally, there are a larger number of communications whose timing delay are characterized by the lower half of the timing spectrum. This means that an analysis of the top half of the spectrum will leave a majority of the communications unlabeled and therefore does not suffer when more timing labels are introduced. Finally, an analysis of the complete timing spectrum starts off with the same efficiency of placing a minimal bound. But as multiple timing partitions are allocated, communications from the lower half of the spectrum become labeled and the overhead increases at a dramatic rate.

These same trends are observable when the timing analysis is applied to data-path binding. As shown in the comparative results of Table 5.2 and Table 5.3, the efficiency of these two techniques are very competitive. Therefore, the efficiency plotted in Figure 6.7 is very similar to that of the previous charts. This similarity is

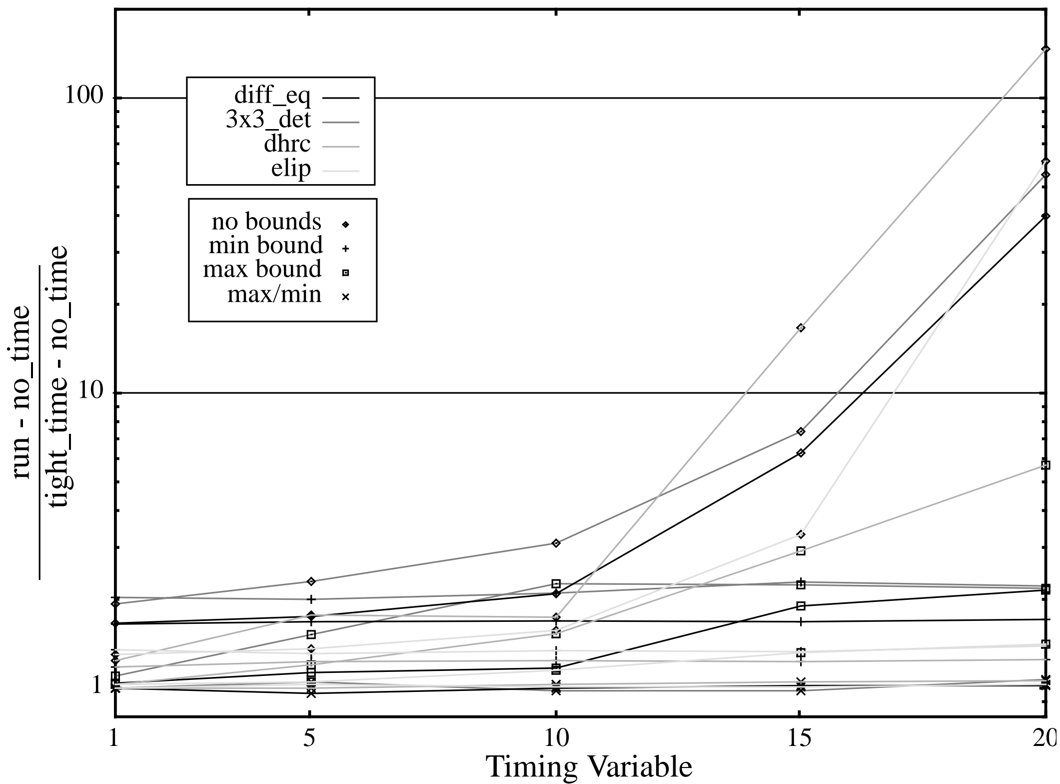


Figure 6.7 Timing analysis overhead in binding.

further underscored by plotting the overhead instead of the run times. Since run time of performing the data-path binding problem without timing analysis is subtracted from the run time, the relative amount of overhead may be compared. The plots look surprisingly similar. The resounding principle of these results is the need to limit the number of timing partitions until the effective bounds may be generated. Once they are obtained, the granularity of the timing analysis may be dramatically increased without adding substantial overhead.

Unfortunately the results for the combination of timing and data path scheduling are not as uniform. The problem stems from the unconstrained nature of the analysis combined with the explosive growth resulting from expanding the analysis of operand non-transfers. Of the example data-flow graphs, only diff_eq could be scheduled in the presence of timing analysis. But even here, substantial

amounts of overhead were incurred as the run time leapt from 500 to 17,826 seconds as a tight timing analysis was performed on the same data path. Clearly, more efficient means of modeling operand non-transfers must be derived before this technique may be applied to larger data-flow graphs. Until that time, the model is still highly suited towards modeling the scheduling freedom in short data-flow graphs such as CISC instructions.

Chapter 7

Discussion

7.1 Summary

This thesis presents a symbolic model to represent the execution freedom of an existing data path. The exploration of this freedom gives designers their first opportunity to generate optimal schedules and binding information for a given data-flow graph. While other researchers have explored parallel methods to generate quality schedules, I know of no work which is capable of capturing the freedom of data-path activity that is presented in this work. The central novelty of this approach is the exact nature in which data path activity is pruned. Instead of characterizing the data path by an artificial set of orthogonal operations, this technique considers any operation which is feasible as defined by the data-path design and any external constraints imposed by the user. This approach coupled with the reachable state technique allows optimal solutions to be proved by construction.

The power of this technique is enhanced by additions to the data-path and data-flow-graph models, as well as, a number of processing techniques to keep this technique practical. The decision to model the positional status of operands was an immense benefit. By shifting the focus away from operations and on to operands,

techniques, such as register constraints, alternative operations, operand recomputation, multiple topologies, and timing models, were incorporated with a minimal amount of overhead. The price of this technique is the cost of representing the multiple instances of the same operands in different data path locations. This requirement hinders the reachable state analysis by creating a large number of feasible states. To combat this issue of feasible states and make the technique viable, the following exact pruning techniques were developed: memory mapping, operand lifetimes, ALAP bounds, encoding techniques, and two phase executions. These pruning techniques permit the identification of optimal schedules for data-flow graphs containing sixteen operations on a practical data-path. These figures are suitable for scheduling micro-code instructions or tight inner loops. The scale of the suitable problem size can be expanded dramatically through the use of scheduling and binding constraints.

7.2 Future Research Lines

While this work has addressed many issues which are required to make this model practicable for real-world systems, a variety of related open research topics still exist. This section shall summarize the most pressing issues and list some tentative thoughts on how to address them.

7.2.1 Better lifetime bounds

The single factor which curtails the size of the suitable problems is operand-lifetime bounds. These bounds help to reduce the complexity at a given cycle by reducing the number of operands which are modeled. This reduction in states, in turn, increases the efficiency of the reachable state analysis. While Section 5.1 and Section 5.2 displayed that effective lifetimes can be generated from a detailed set of constraints, one would expect many applications to have less detailed

constraints. One can expect constraints will exist for external interfaces but that there will be minimal scheduling constraints for the rest of the chip in order to maximize the design freedom. Forming accurate lifetime bounds in the presence of such minimal constraints proves to be a daunting task. The top issues, in relative order of complexity, are: the recomputation of operands, the restrictive movement of operands, and the combined interaction of operands. The key to overcoming these issues will be in exploiting their restrictive nature to formulate lifetime constraints. An example of this was demonstrated in the ALAP bounds presented in Section 5.3.1 which utilized the restrictive nature of the operand movement to generate stronger ALAP bounds.

7.2.2 Cyclic data-flow graphs

The restriction that all data-flow graphs be acyclic helped simplify the automatic construction of the automata. But, this constraint is overly restrictive. A number of operands may be recomputed from their child operands. The most pressing example of this is a 32-bit operand which is partitioned into two 16-bit operands. The original operand may be recreated by simply merging the two child operands at a significant cheaper cost in terms of data path resources. This example is meant to demonstrate two points: 1) that cyclic data flows are common in real world designs, and 2) that accommodating cyclic data flows will enable the modeling of data paths with different sized bus widths. The alterations that cyclic data-flow graphs will require are: 1) a fixed point algorithm to incorporate the cyclic dependencies into the formulation of $F_k(\Theta, \Upsilon, \Sigma, V)$, and 2) a re-evaluation of the effect of cyclic bounds on lifetime bounds such as the ALAP bound.

7.2.3 Control data-flow graphs

The present representation allows control signals to sent to the controller, but the scheduler does not incorporate these control decisions into the scheduling decisions. Control data-flow graphs must be partitioned into a set of individual data-flow graphs with a consistent interface of operand placement in order to be scheduled on the present system. The main problem with this approach is the difficulty to generate the proper operand placement for the interface conditions. Additional benefits, such as loop unwinding and speculative execution, which have been developed for control data-flow graphs may not be explored. Efforts to address these issues will face many challenges but will be characterized by the incorporation of control settings into the data-path state format.

Bibliography

1. S. B. Akers, “Binary Decision Diagrams”, *IEEE Trans. Computers*, pp.509-516, June 1978.

2. P. Ashar and M. Cheong, “Efficient Breadth-First Manipulation of Binary Decision Diagrams”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.622-627, San Jose, USA, Nov. 1994.

3. R. I. Bahar, *et al.*, “Algebraic Decision Diagrams and their Applications”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.188-191, San Jose, USA, Nov. 1993.

4. A. Balachandran, D. M. Dhamdhere, and S. Biswas, “Efficient Retargetable Code Generation Using Bottom-Up Tree Pattern Matching”, Computer Languages, pp.127-140, 1990.

5. H. Bakoglu, Circuits, *Interconnections, and Packaging for VLSI*, Addison-Wesley Publishing Company, 1990.

6. J. Benkoski, *et al.*, “Timing Verification Using Statically Sensitizable Paths” *IEEE Trans. CAD/ICAS*, pp.1073-1084. Oct. 1990.

7. I. Bolsens, *et al.*, “Assessment of the Cathedral-II Silicon Compiler for Digital-Signal-Processing Applications” *ESA Journal*, pp.243-260, 1991

8. G. Borriello, *et al.*, “Embedded System Co-Design: Towards Portability and Rapid Integration,” *Hardware/Software Co-Design*, M.G. Sami and G. De Micheli, EDs., Kluwer Aacademic Publishers, 1995.

9. K. S. Brace, R. L. Rudell, and R. E. Bryant, “Efficient Implementation of a BDD package”, *Proc. 27th ACM/IEEE Design Automation Conf.,* pp.40-45, Orlando, USA, June 1990.

10. D. Brand, *et al.*, “Incremental Synthesis”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.14-18, San Jose, USA, Nov. 1994.

11. R. Brayton, *et al.*, “VIS”, *Proc. of the First Int. Conference on Formal Methods in Computer-Aided Design*, pp.248-256, San Jose, USA, Nov. 1996.

12. F. Brewer and D. Gajski “Chippe: A System for Constraint Driven Behavioral Synthesis” *IEEE Trans. CAD/ICAS*, pp.681-95, July 1990

13. R. E. Bryant, “Graph-Based Algorithms for Boolean Function Manipula-

tion", *IEEE Trans. Computers*, pp.677-691, Aug. 1986.

14. R. E. Bryant, "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams", *ACM Computing Surveys*, pp.293-318, Sep. 1992.

15. R. E. Bryant and Y.-A. Chen, "Verification of Arithmetic Circuits with Binary Moment Diagrams", *Proc. 32th ACM/IEEE Design Automation Conf.,* pp.535-541, San Francisco, USA, June 1995.

16. R. E. Bryant, "Binary Decision Diagrams and Beyond: Enabling Technologies for Formal Verification", *Proc. Int. Conf. Computer-Aided Design*, pp.236-243, San Jose, USA, Nov. 1995.

17. J. R. Burch, *et al.*, "Symbolic Model Checking for Sequential Circuit Verification", *IEEE Trans. CAD/ICAS*, pp.401-424, April 1994.

18. R. Camposano, "Path-Based Scheduling for Synthesis", *IEEE Trans. CAD/ICAS*, pp.85-93, Jan. 1991.

19. R. Camposano, *et al.*, "The IBM High-Level Synthesis System", *High-Level VLSI Synthesis*, R. Camposano and W. Wolf, eds., Kluwer, 1991

20. R. Camposano and W. Rosenstiel, "A Design Environment for the Sythesis of Integrated Circuits", *11th Symp. Microprocessing and Microprogramming EUROMICRO '85*, Brussles, Belgium, pp.211-215, Sept. 1985.

21. H.-C. Chen and D. Du, "Path Sensitization in Critical Path Problem" , *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.208-211, San Jose, USA, Nov. 1991.

22. H. D. Cheng and C. Xia, "High-Level Synthesis: Current Status and Future Prospects", *Cicuits Systems Signal Process*, pp.351-400, 1995

23. H. Cho, *et al.*, "Algorithms for Approximate FSM Traversal", *Proc. 30st ACM/IEEE Design Automation Conf.*, pp.25-30, Dallas, USA, June 1993.

24. P. Chou, E. Walkup and G. Borriello, "Scheduling Issues in the Co-Synthesis of Reactive Real-Time Systems," *IEEE Micro*, Aug. 1994. pp.37-47.

25. E. M. Clarke, *et al.*, "Multi-Terminal Binary Decision Diagrams: An Efficient Data-Structure for Matrix Representation", *Int. Workshop on Logic Synthesis*, pp. 610-615, 1993.

26. R.J. Cloutier and D.E. Thomas, "The Combination of Scheduling, Allocation, and Mapping in a Single Algorithm", *Proc. 27th ACM/IEEE Design Automation Conf.,* pp.71-76, Orlando, USA, June 1990.

27. C. N. Coelho Jr and G. De Micheli, "Dynamic Scheduling and Synchronization Synthesis of Concurrent Digital Systems under System-Level Constraints", *Proc. IEEE Int. Conf. Computer-Aided Design*, p.175-181, San

Jose, USA, Nov. 1994.

28. O. Coudert, C. Berthet and J. C. Madre "Verification of Synchronous Sequential Machines Based on Symbolic Execution", *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, pp.365-373, Grenoble, France, 1989.

29. O. Coudert and J. C. Madre. "A Unified Framework for the Formal Verification of Sequential Circuits," *Proc. Int. Conf. Computer-Aided Design*, pp.126-129, San Jose, USA, Nov. 1990.

30. O. Coudert, "Two-level Logic Minimization: An Overview", *Integration, the VLSI journal*, pp.97-140, Oct. 1994.

31. S. Davidson, *et al.*, "Some Experiments in Local Microcode Compaction for Horizontal Machines", *IEEE Trans. Computers*, pp.460-477, July 1981.

32. G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, Inc., 1994.

33. D. Du, S.Yen and S. Ghanta, "On the General False Path Problem in Timing Analysis", *Proc. 26th ACM/IEEE Design Automation Conference Proc.*, pp.555-560, Las Vegas, USA, June 1989.

34. C. Ewering, "Automated High Level Synthesis of Partitioned Busses" *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.93-102, San Jose, USA, Nov. 1990

35. S. J. Friedman and K. J. Supowit, "Finding the Optimal Variable Ordering for Binary Decision Diagrams", *IEEE Trans. Computers*, pp.710-713, May 1990.

36. M. Fujita, *et al.*, "Application of Boolean Unification to Combinational Logic Synthesis", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.510-513, San Jose, USA, Nov. 1991

37. D. Gajski, *et al.*, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, 1992.

38. D. Gajski and L. Ramachandran, "Introduction to high-level synthesis", *IEEE Design & Test of Computers*, pp.44-54, Winter 1994.

39. T. Granlund and R. Kenner, "Eliminating Branches using a Superoptimizer and the GNU C Compiler", *Proc. of the ACM SIGPLAN'92 Conference on Programming Language Design and Implementation (PLDI)*, pp.341-352, San Francisco, USA, 1992

40. K. Hamaguchi, A. Morita and S. Yajima, "Efficient Construction of Binary Moment Diagrams for Verifying Arithmetic Circuits", *Proc. Int. Conf. Computer-Aided Design*, pp. 78-82, San Jose, USA, Nov. 1995.

41. B.S. Haroun and M.I. Elmasry, “Architectural Synthesis for DSP Silicon Compiler”, *IEEE Trans. CAD/ICAS*, pp.431-47, April 1989.

42. A. Hu, *et al.*, “Higher Level Specification and Verification with BDD’s” *Computer-Aided Verification: Fifth Int. Conference*, 93, Lecutre Notes in Computer Science v.697, Springer-Verlag, 1993.

43. S. H. Huang, *et al.*. “A Tree-Based Scheduling Algorithm for Control Dominated Circuits”, *Proc. 30th ACM/IEEE Design Automation Conf.*, pp.578-582, Dallas, USA, June 1993.

44. C.-T. Hwang, J.-H. Lee and Y.-C. Hsu, “A Formal Approach to the Scheduling Problem in High Level Synthesis”, *IEEE Trans. CAD/ICAS*, pp.464-475, Apr. 1991.

45. S.-W. Jeong and F. Somenzi, “A New Algorithms for the Binate Covering Problem and its Application to the Minimization of Boolean Relations”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.417-420, San Jose, USA, Nov. 1992.

46. T. Y. K. Kam and R. K. Brayton, *Multi-valued Decision Diagrams*, Memo. no. UCB/ERL M90/125, UC Berkeley, Dec. 1990.

47. S. Kimura, “Residue BDD and its Application to the Verification of Arithmetic Circuits”, *Proc. 32th ACM/IEEE Design Automation Conf.,* pp.542-545, San Francisco, USA, June 1995.

48. D. W. Knapp, “Fasolt: A Program for Feedback-Driven Data-Path Optimization”, *IEEE Trans. CAD*, pp.677-695, June 1992.

49. Y.-T. Lai, M. Pedram and S. B. K. Vrudhula, “EVBDD-Based Algorithms for Integer Linear Programming, Spectral Transformation, and Function Decomposition”, *IEEE Trans. CAD/ICAS*, pp.959-975, Aug. 1994.

50. R. Leupers and P. Marwedel, “Time Constrained Code Compaction for DSPs” in *IEEE Trans. on VLSI Systems,* pp.112-122, 1997

51. R. Leupers and P. Marwedel, “Retargetable Generation of Code Selectors from HDL Processor Model” *Proc. of European Design & Test Conference*, p.140-144, Paris, France, March 1997

52. R. Leupers and P. Marwedel, “Algorithms for Address Assignment in DSP Code Generation” *Proc. Int. Conf. Computer-Aided Design*, pp.109-112, San Jose, USA, Nov. 1996.

53. R. Leupers and P. Marwedel, “A BDD-based Frontend for Retargetable Compilers” *Proc. the European Design & Test Conference*, pp.239-243, Paris, France, March 1995

54. S. Liao, *et al.*, “Code Optimization Techniques for Embedded DSP Micro-

processors", *Proc. 32nd ACM/IEEE Design Automation Conference Proc.*, pp.599-604, San Francisco, USA, June 1995.

55. S. Liao, *et al.*, "Storage Assignment to Decrease Code Size", *ACM Trans. on Programming Languages and Systems*, vol.18, (no.3), ACM, May 1996. pp.235-53.

56. H.-T. Liaw and C.-S, Lin, "On OBDD-Representation of General Boolean Functions", *IEEE Trans. Computers*, pp. 661-664, June 1992.

57. B. Lin, *Synthesis of VLSI Designs with Symbolic Techniques*, PhD thesis, memo. no. UCB/ERL M91/105, UC Berkeley, Nov. 1991.

58. B. Lin and S. Devadas, "Synthesis of Hazard-Free Multi-level Logic under Multiple-Input Changes from Binary Decision Diagrams", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 542-549, San Jose, USA, Nov. 1994.

59. C-C. Lin, *et al.*, "Logic Synthesis for Engineering Change", *Proc. 32nd ACM/IEEE Design Automation Conference Proc.*, pp. 647-652, San Francisco, USA, June 1995.

60. S. Malik, *et al.*, "Logic Verification using Binary Decision Diagrams in a Logic Synthesis Environment", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp. 6-9, San Jose, USA, Nov. 1988.

61. P. Marwedel and G. Goosens (eds.), *Code Generation for Embedded Processors*, Kluwer Academic Publishers, 1995.

62. H. Massalin, "Superoptimizer -- A Look at the Smallest Problem" in *Proc. of the Second Int. Conference on Architectural Support for Programming Languages and Operating Systems*, pp.122-126, 1987

63. M. C. McFarland, A. C. Parker, and R. Camposano, "The High-Level Synthesis of Digital Systems", *Proc. IEEE*, vol. 78, no. 2, pp.301-318, Feb. 1990.

64. M. C. McFarland and T. J. Kowalski, "Incorporating Bottom-Up Design into Hardware Synthesis", *IEEE Trans. CAD/ICAS*, pp.938-50, Sept. 1990.

65. S.-I. Minato, "Zero-Suppressed BDDs for Set Manipulation in Combinatorial Problems", *Proc. 30th ACM/IEEE Design Automation Conf.*, pp.272-277, Dallas, USA, June 1993.

66. S.-I. Minato, "BDD-Based Manipulation of Polynomials and Its Applications", *Proc. Intl. Workshop on Logic Synthesis*, pp.5.31-5.43, 1995.

67. S.-I. Minato, *Binary Decision Diagrams and Applications for VLSI CAD*, Kluwer Academic Publishers, 1995.

68. D. Mintz and C. Dangelo, "Timing Estimation for Behavioral Descriptions"

Proceedings of the Seventh International Symposium on High-Level Synthesis, pp.42-47, Niagara-on-the-Lake, Canada, May 1994.

69. T. Miyazaki and M. Ikeda, “High Level Synthesis Using Given Datapath Information”, *IEECE Trans. Fundamentals*, p.1617-1625, Oct 1993

70. C. Monahan and F.Brewer, “Communication Driven Interconnection Synthesis”, *Proc. of 6th International Workshop on High Level Synthesis*, Dana Point CA, Nov. 1992

71. C. Monahan and F. Brewer, “Symbolic Modeling and Evaluation of Data Paths”, *Proc. 32nd ACM/IEEE Design Automation Conference Proc.*, pp.389-394, San Francisco, USA, June 1995.

72. M. Nourani and C. Papachristou, “False Path Exclusion in Delay Analysis of RTL-Based Datapath-Controller Designs” *Proceedings EURO-DAC ’96. European Design Automation Conference with EURO-VHDL ’96*, pp.336-341, Geneva, Switzerland, Sept. 1996.

73. S. Note, *et al.*, “Combined Hardware Selection and Pipelining in High-Performance Data-Path Design” *IEEE Trans. CAD/ICAS*, pp.413-423, April 1992.

74. S. Panda, F. Somenzi and B. F. Plessier, “Symmetry Detection and Dynamic Variable Ordering of Decision Diagrams”, *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.628-631, San Jose, USA, Nov. 1994.

75. S. Panda and F. Somenzi, “Who Are the Variables in Your Neighborhood”, *Proc. Int. Conf. Computer-Aided Design*, pp.74-77, San Jose, USA, Nov. 1995.

76. B. M. Pangrle and D. D. Gajski, “Design Tools for Intelligent Silicon Compilation”, *IEEE Trans. CAD/ICAS*, pp.1098-1112, Nov. 1987.

77. N. Park and F. Kurdahi, “Module Assignment and Interconnect Sharing in Register Transfer Synthesis of Pipelined Data-Paths” *Proc. IEEE Int. Conf. Computer-Aided Design,* pp.16-19, San Jose, USA, Nov. 1989.

78. N. Park and A. C. Parker., “SEHWA: A Software Package for Synthesis of Pipelines from Behavioral Specifications”, *IEEE Trans. CAD/ICAS*, pp.356-370, March 1988.

79. P.G. Paulin and J.P. Knight, “Force-Directed Scheduling for the Behavioral Synthesis of ASIC’s”, *IEEE Trans. CAD/ICAS*, pp.661-79, June, 1989.

80. S. Perremans, L. Claesen and H. De Man, “Static Timing Analysis of Dynamically Sensitizable Paths”*, Proc. 26th ACM/IEEE Design Automation Conference Proc.*, pp.568-573, Las Vegas, USA, June 1989.

81. I. Radivojević and F. Brewer, “A New Symbolic Technique for Control-

Dependent Scheduling", *IEEE Trans. CAD/ICAS*, pp.45-57, Jan. 1996.

82. R. Rudell, "Dynamic Variable Ordering for Binary Decision Diagrams", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.42-47, San Jose, USA, Nov. 1993.

83. T. Shinsha, *et al.*, "Incremental Logic Synthesis through gate logic structure identification", *Proc. 23rd ACM/IEEE Design Automation Conference Proc.*, pp.391-397, San Francisco, USA, June 1986.

84. D. E. Thomas, *et al.*, "Automatic Data Path Synthesis", *Computer*, pp.59-70, Dec. 1983.

85. A. Timmer, *From Design Space Exploration to Code Generation*, Ph.D. Thesis Eindhoven University of Technology, 1996

86. H. J. Touati, *et al.*, "Implicit State Enumeration of Finite State Machines using BDD's," *Proc. Int. Conf. Computer-Aided Design*, pp.130-133, San Jose, USA, Nov. 1990.

87. F. S. Tsai and Y.C. Hsu, "Data Path Construction and Refinement", *Proc. IEEE Int. Conf. Computer-Aided Design*, pp.308-311, San Jose, USA, Nov. 1990.

88. J. Van Praet, *et al.*, "A Graph Based Processor Model for Retargetable Code Generation", *Proc. of European Design and Test Conference*, pp.102-7, Paris, France, 1996

89. K. Wakabayashi and H. Tanaka, "Global Scheduling Independent of Control Dependencies Based on Condition Vectors", *Proc. 29th ACM/IEEE Design Automation Conf.,* pp.112-115, Anaheim, USA, June 1992.

90. R. A. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems*, Kluwer Academic Publishers, 1991.

91. Y. Wantanabe and R. Brayton, "Incremental Synthesis for Engineering Change", *Proc. IEEE ICCD*, pp.40-43, Boston, 1991.

92. J. C.-Y. Yang, G. De Micheli and M. Damiani, "Scheduling and Control Generation with Environmental Constraints based on Automata Representations", *IEEE Trans. CAD/ICAS*, p.166-83, Feb. 1996.

Appendix A

Binary Decision Diagrams

Binary Decision Diagrams (BDDs) are one of the biggest breakthroughs in CAD in the last decade. BDDs are a *canonical* and *efficient* way to represent and manipulate Boolean functions and have been successfully used in numerous CAD applications. Although the basic idea has been around for more than 30 years (e.g. [1]), it was Bryant who described a canonical BDD representation [13] and efficient implementation algorithms [9]. References [14,16,67] are very readable introductions to BDD representations and applications.

Ordered Binary Decision Diagram of a Boolean function *f* can be obtained by iterative application of the Shannon decomposition with respect to a specified variable ordering:

$$f = xf_x + \bar{x}f_{\bar{x}} \quad \text{(EQ 7.1)}$$

A decision tree obtained in such a manner is reduced using two rules: (*i*) eliminate all nodes that have isomorphic sons ("don't care" elimination), and (*ii*) identify and share all isomorphic subgraphs. This process results in a Reduced Ordered BDD which is a canonical representation of a Boolean function for a specific variable ordering.

Using the *ite* (if-the-else) terminology, the Equation (7.1) can be re-written as:

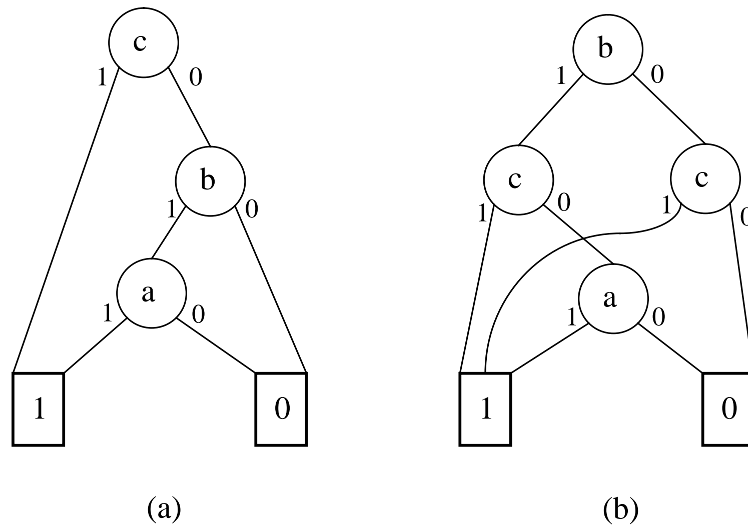


Figure 7.1 ROBDD forms of f=AB+C using different orderings

$$f = ite(x, f_x, f_{\bar{x}}) \quad \text{(EQ 7.2)}$$

All basic Boolean function manipulations can be described using *ite* templates. For example:

$$And(g, h) = ite(g, h, 0) \quad \text{(EQ 7.3)}$$

and:

$$Not(g) = ite(g, 0, 1) \quad \text{(EQ 7.4)}$$

The property that all Boolean manipulations can be treated in a unique manner (using *ite* calls) enables efficient implementations using computer hashing/ cashing techniques [9].

Figure 7.1 illustrates ROBDD forms of $f = AB + C$ for two different variable orderings. An edge labeled by “1” (“0”) corresponds to a variable’s phase x ($\bar{x}$) in the decomposition formula above. The problem of finding the ordering that results in the smallest ROBDD (in terms of the number of nodes in the graph) is NP-complete. An exact variable ordering algorithm was developed in [35], but found a very limited application due to its computational complexity. Moreover,

theoretical analysis of general Boolean functions [56] indicates that, for the majority of functions, “good” orderings do not exist (i.e. the best ordering still leads to exponentially complex graphs). However, ROBDDs have performed extremely well in many practical CAD applications. Typically, the underlying structure of the problem solved using ROBDDs allows development of efficient heuristic ordering strategies (e.g. [60]).

Decision diagrams and their applications are a very active research area. Some interesting, more recent developments include:

- algebraic decision diagrams [3],
- asynchronous circuit synthesis [58],
- binate covering problem (BCP) solver [45],
- BDDs for implicit set representation in combinatorial problems [65] and applications to polynomial algebra [66],
- efficiency improvements through dynamic variable reordering [74,75,82] and breadth-first manipulations [2],
- exact and approximate FSM traversal techniques [23,28,29,86],
- formal verification of arithmetic circuits [15,40,47],
- integer linear programming (ILP) solver based on edge-valued BDDs [49],
- implicit prime generation and two-level minimization [30],
- matrix representation and manipulations using multi-terminal BDDs [25],
- multi-valued decision diagrams [46],

- symbolic model checking [17],
- symbolic synthesis techniques [57].

This list is *by no means* complete!

Glossary

A_j: Active operand set

ALAP(e): As late as possible for operation e.

B_k: The set of children for operand p_k

C: Set of data path components

c_i: data path component

$C(\phi)$: function returns the component associated with input port ϕ

$C(\theta)$: function returns the component associated with output port θ

D_j: Dead operand set for cycle j

D_j': Suboptimal dead op set;

E: operation set

e: operation

$F_k(\Upsilon, \Theta, \Sigma, V)$: Conditions under which operand p_k is computed.

$F'_{\theta, \vec{\sigma}, \Pi, p, \Phi}(\Theta, \Upsilon, \Sigma, V)$: Conditions which support the spec. operation.

$M_k(\Theta, \Sigma, V)$: Conditions under which operand p_k is read from memory

$N(\Upsilon, V, V')$: State relation

$N_i(\Upsilon, \Sigma, V, V'_i)$: State relation for memory device c_i

$N_{i,k}(\Upsilon, \Sigma, V, V'_i)$: Storage of op_k in memory device c_i.

$N'_{i,k}(\Upsilon, \Sigma, V)$: Conditions under which op_k reaches memory device c_i

$N^{\circ}_{i,j}(\Upsilon, \Sigma, V, V'_i)$: State relation for memory device c_i on cycle j

$N^{\circ}_{i,j,k}(\Upsilon, \Sigma, V, V'_i)$: Relation on cycle j where op_k is stored in memory device c_i

null: special operand meaning “not of operand set”

P: Operand set = $\{P', P_1, null\}$

P': operands

P_i: operands at a memory device

P_0: set of operands created by external inputs

P_1: set of signals

$P_{i,j}$: operands at input ports of device c_i at cycle j

R_j: The state relation between cycle j-1 and j

$R_j(\Upsilon, V, V')$: a reachable state relation

$R^\circ_j(\Upsilon, V, V')$: state relation which leads to a solution

S: State set

$S_0(\Upsilon, V)$ init states

$S_f(\Upsilon, V')$ final states

$S_j(\Upsilon, V)$: reachable states

S'_j: frontier states

S''_j: suboptimal state set

S°_j: extraction states

T_j: Total reachable states at cycle j

$t_{i,j}$: cumulative delay to port i based on condition $\Lambda_{i,j}(\Upsilon, \Sigma)$

V: State variables

V': Next state set

V'': System wide operand state set

V_i: State set for memory device c_i

V_i': Next state set for memory device c_i

$v_{i,k}$: state encoding for operand p_k in memory device c_i

$\bar{v}_{i,k}$: state encoding for operand p_k not in memory device c_i

$v_{i,k}'$: next state encoding for operand p_k in memory device c_i

$\bar{v}_{i,k}'$: next state encoding for operand p_k not in memory device c_i

Θ : Output port set

Θ_i: Output port set for component c_i

Θ': Set of output ports connected to wires

Θ_i': Set of output ports connected to wires for component c_i

Θ'': Set of output ports connected to control

Θ_i'': Set of output ports connected to control for component c_i

θ_i: Output port

Π_i: Input operands for an operantion, e_i

π_i: the ith operand for a given set Π

Σ : Control line set

Σ_i: Control line set for component c_i

Σ': Control line set w/o dedicated control lines

σ_i: control line

$\sigma_{k,\theta}$: symbolic control line request for operand p_k from output port θ of a register file.

$\vec{\sigma}$: a control setting

$\vec{\sigma}_i(\phi)$: mux. control setting to select input port ϕ on component c_i

Φ_i: input port set of component c_i

ϕ : input port

ϕ_{i,p_k}: input port on component c_i used for input operand p_k

τ'_i: Output port set that reach wire w_i crossing any number of memory devices

$\tau_{j,x}$: Output port set that reach wire w_i crossing only x memory devices

$\tau(\theta_i, w_j)$: minimum number memory devices to link θ_i and w_j.

χ_j: Restriction for the jth relation.

$\Omega_i(\Upsilon, \Sigma, \Theta)$: relation of control and ports to connect to input port ϕ_i

Υ : Set of network topologies

υ : a given network topology

Ψ : Set of interconnections

$\Psi_i(\Theta, \Upsilon)$: Output port and topology relation set for each input port, ϕ_i

ψ : element of $\Psi_i(\Theta, \Upsilon)$ linking an input port to a network topology

Δ_i: Component delay

$\Delta_i(\vec{\sigma}, \phi, \theta)$: throughput delay

$\Delta_i(\vec{\sigma}, \varnothing, \theta)$: read delay

$\Delta_i(\varnothing, \phi, \varnothing)$: write delay

Δ'_i: Set of dealys from PI to ith port

$\delta'_{i,j}$: element of Δ'_i consisting of $(t_{i,j}, \Lambda_{i,j}(\Upsilon, \Sigma))$

Δ°: Set of descrete timing ranges

δ°_i: a descrete timing range

$\Lambda_{i,j}(\Upsilon, \Sigma)$: network path condition linking PI to ith port