Architectural-Level Synthesis Of Asynchronous Systems

Article · April 1999

Source: CiteSeer

CITATIONS
6

READS
68

3 authors, including:

Erik Brunvand
University of Utah
117 PUBLICATIONS **1,342** CITATIONS

SEE PROFILE

ARCHITECTURAL-LEVEL SYNTHESIS OF ASYNCHRONOUS SYSTEMS

by

Brandon M. Bachman

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Electrical Engineering

The University of Utah

December 1998

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Brandon M. Bachman

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

	Chair: Chris J. Myers
	Erik Brunvand
	Christian Schlegel

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Brandon M. Bachman in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

Om P. Gandhi
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

Asynchronous circuit design has the potential to produce circuits superior to those of synchronous circuit design. Current synchronous methods of architectural-level synthesis do not exploit properties inherent to asynchronous circuits. This research describes potential optimizations and techniques that can be applied to the architectural-level design of asynchronous systems. The proposed methods take advantage of asynchronous circuit properties such as data-dependent delays, modularity, and composiblity. The optimization problems of scheduling and allocation are studied. For scheduling, some counterintuitive properties of delays in a system are shown. The design space is studied and several filters to reduce the size of the design space are proposed. To evaluate and test these ideas the CAD tool `Mercury` was developed and is described in detail. `Mercury` is unique in that it can take an abstract model of a design, in this case a data flow graph, and from that generate both an optimal structural view of an asynchronous datapath for the design, as well as the necessary behavioral control to operate that datapath. Several case studies are presented utilizing the tool and methods to illustrate the practical aspects of this work.

To my loving wife, Marianne, and to my parents, Danel and Patricia.

CONTENTS

ABSTRACT iv

LIST OF FIGURES viii

LIST OF TABLES x

ACKNOWLEDGEMENTS xi

CHAPTERS

1. INTRODUCTION 1

1.1 Motivation 2
1.2 Related Work 3
1.3 Contributions 5
1.4 Thesis Outline 6

2. ARCHITECTURAL LEVEL MODELING 8

2.1 Representation and Modeling 8
2.2 Modeling Resources 12
2.3 System Constraints 15
2.4 Output 16
2.4.1 Datapath Generation 16
2.4.2 Control Generation 20

3. DESIGN SPACE EXPLORATION 25

3.1 Binding 26
3.2 Scheduling 26
3.2.1 ASAP Scheduling 27
3.2.2 ALAP Scheduling 28
3.2.3 Mobility 29
3.2.4 Force-Directed Scheduling 30
3.2.5 Statistical Delay Calculation 31
3.2.6 Monte-Carlo Delay Calculation 32
3.3 Typical Delay 33
3.4 Resource Allocation 34
3.4.1 Left-Edge Algorithm 36
3.4.2 Clique Covering 39

4. THE DESIGN SPACE 41

4.1 Reducing the Design Space 42
4.2 Filters 43
4.2.1 Infeasible Edges 43

4.2.2 Redundancy . . . 44
4.2.3 Implied Edges . . . 46
4.2.4 Minimal Latency . . . 47
4.2.5 Constraints . . . 48
4.2.6 Maximally Shared Resources . . . 48
4.2.7 No Change In Objectives . . . 49
4.3 Hierarchal Exploration . . . 49

5. SYSTEM IMPLEMENTATION . . . 52

5.1 General Algorithm . . . 52
5.2 Optimizations . . . 55
5.2.1 Dynamic Transitive Closure . . . 56
5.2.2 Dynamic Scheduling . . . 57

6. CASE STUDIES . . . 60

6.1 Differential Equation Solver . . . 60
6.2 Elliptical Wave Filter . . . 65
6.2.1 Comparison with Synchronous Methods. . . . 69
6.3 Inverse Discrete Cosine Transform . . . 70

7. CONCLUSIONS . . . 75

7.1 Possible Extensions . . . 76

APPENDICES

A. SAMPLE VHDL DATAPATH . . . 79

B. SAMPLE VHDL CONTROL . . . 82

C. SAMPLE VHDL CONFIGURATION . . . 85

REFERENCES . . . 86

LIST OF FIGURES

1.1 `Mercury` design flow. 7
2.1 Behavioral VHDL and corresponding data flow graph. 10
2.2 Input format for a data flow graph (DFG). 11
2.3 Sample data flow graph and model description. 12
2.4 Request/acknowledge interface with four-phase handshake protocol. 13
2.5 Input format for the datapath resource library (DRL). 14
2.6 Sample datapath resource library (DRL). 15
2.7 Input format for constraints. 15
2.8 Sample constraints specification. 16
2.9 Datapath format. 17
2.10 Datapath generated from sample model description. 18
2.11 Timing diagram showing handshaking protocol. 23
2.12 Structural control generated by the `ATACS` CAD tool. 24
3.1 As-soon-as-possible and as-late-as-possible scheduling. 28
3.2 Critical windows derived from as-soon-as-possible scheduling. 29
3.3 As-soon-as-possible and as-late-as-possible algorithms. 30
3.4 Force-directed scheduling. 31
3.5 A data flow graph with four operations: A, B, C and D. 31
3.6 Data flow graphs without and with resource edges. 35
3.7 Synchronous methods vs. asynchronous approach. 36
3.8 Asynchronous left-edge algorithm. 37
3.9 Example using the asynchronous left-edge algorithm. 38
3.10 Compatibility graph for clique covering. 39
4.1 Exploration space of 3 compatible operations. 42
4.2 Infeasible edge. 44
4.3 Procedure to determine if adding a resource edge creates a valid design. . . . 45
4.4 Design space showing redundancy. 45
4.5 Implied edge. 46

4.6 Minimal latency filter. 48
4.7 Grouping of resources for hiearchal exploration. 50
5.1 Pareto Points. 53
5.2 Exploring the design space using a branch-and-bound search. 54
5.3 Exploration tree. 55
5.4 Updating dynamic transitive closure for insertion of an edge. 58
5.5 Updating dynamic transitive closure for deletion of an edge. 58
5.6 Updating ASAP schedule for insertion or deletion of an edge. 59
5.7 Updating ALAP schedule for insertion or deletion of an edge. 59
6.1 DIFFEQ: minimum latency solution 64
6.2 DIFFEQ: minimum area solution 64
6.3 Functional notation for the elliptical wave filter. 65
6.4 Elliptical wave filter data flow graph. 66
6.5 Elliptical wave filter datapath. 68
6.6 Comparison with synchronous methods. 70
6.7 Inverse discrete cosine transform data flow graph. 72
6.8 IDCT: sample minimum latency datapath. 74

LIST OF TABLES

6.1 DIFFEQ: experimental results using nonhierarchal approach. 62
6.2 DIFFEQ: experimental results using hierarchal approach. 63
6.3 DIFFEQ: comparison of unique Pareto point solutions. 65
6.4 EWF: experimental results using hierarchal approach. 67
6.5 EWF: comparison of unique solutions using hierarchal approach. 69
6.6 IDCT: experimental results using hierarchal approach. 73

ACKNOWLEDGEMENTS

I am in debt to many people who have made this work possible. Among my colleagues, I am thankful to Robert Thacker for always offering a helping hand, to Wendy Belluomini for much good criticism and advice, to Chris Krieger for many excellent nuggets of understanding, and to Hao Zheng for carrying this work forward. In addition, I would like to thank Luli Josephson for reviewing this work, and Hans Jacobson for his comments.

I would like to express a deep gratitude to my advisor Dr. Chris J. Myers. Over the past several years I have been continually impressed with his constant guidance, technical expertise, and endless encouragement. I would also like to thank Dr. Erik Brunvand and Dr. Christian Schlegel for serving on my supervisory committee. Their comments and assistance have been valuable.

Most of all, I express my thanks to Eric Mercer. I am grateful for his loyal friendship, which included much patience, perseverance, and assistance on my behalf. His savvy technical skills have made this work much better than it would have been otherwise. Through many years of school he has complemented my many weaknesses with strength and he is the unsung hero of this work.

Finally, my wife and family deserve special thanks. I would like to express my gratitude for their love, support, and sacrifice.

CHAPTER 1

INTRODUCTION

Sometimes when I consider the tremendous consequences from little things
. . . a chance word . . . a tap on the shoulder or a wink of an eye,
I am tempted to think there are no little things.

—Emily Dickensen

Asynchronous designs are rapidly becoming an attractive alternative to synchronous designs. As technology advances, the integrated circuit industry continues to increase clock speeds, increase density, and decrease transistor sizes making global synchronization across large chips more difficult to maintain. To solve this problem, many modern chips have a number of communicating clocking domains which can greatly increase design complexity. As a result, asynchronous design is being looked at as an alternative because it has the potential to reduce, and in some cases, eliminate the growing challenges of synchronous design. Asynchronous circuits consist of groups of independent modules which communicate using handshaking protocols. This makes asynchronous designs attractive because they do not have clock skew problems, thus reducing power-expensive global clocks and routing issues. In addition, asynchronous design offers the potential for average-case performance in place of worst-case performance, they are adaptable to environmental conditions, and exhibit ease in composability. For these reasons, there is a growing interest in asynchronous design.

Architectural-level synthesis is the process of taking an abstract behavioral model of a desired circuit and refining it to an optimal macroscopic structure. In an ideal world, everything would be possible at no cost. But, there are no blank checks in circuit design. Issues such as latency, area, and power must be taken into consideration to balance trade-offs in a design. Architectural-level synthesis is an approach to managing these trade-offs at a macroscopic level.

The abstract model used at the architectural-level generally begins as a *data flow graph* that does not contain implementation parameters such as a mapping to specific

resources or technology. The synthesis process takes this abstract model and generates a structural view of the circuit by determining the necessary resources and parameters to implement the behavioral model. The goal of architectural synthesis is to generate an optimal circuit from an abstract model. The model consists of two components: *datapath* and *control*.

The datapath is the portion of the circuit composed of interconnected components that move data and operate on it. The components are usually multi-bit bused structures that contain a high density of arithmetic functions. The control circuitry directs the movement of data and execution of the datapath resources. When combined, the datapath and control work together to make a circuit functional.

The focus of this research is on the automation of architectural-level synthesis for asynchronous systems. This includes the automated generation of an optimal asynchronous datapath and corresponding control. This work merges methods from synchronous architectural-level design with those used to generate asynchronous control circuits and exploits asynchronous circuit properties to design highly optimized asynchronous systems.

1.1 Motivation

Digital signal-processing, high-speed multimedia, graphics, and telecommunications applications are computationally-intensive. In these applications, the datapath requires the largest area of the logic circuitry, sometimes as much as 80% of a complete design. For these applications the datapath is the critical factor when trying to achieve design objectives such as minimal area and latency. The challenge for a datapath designer is to arrive at the best implementation for a given function. Many datapaths today are hand-crafted using a *Register-Transfer Level* (RTL) specification. Using this model storage of data is represented using register variables, and transformations are represented by arithmetic and logical operators.

Typically, designers arrive at a particular design through trial and error methods. This approach is time-consuming and does not yield optimal results. Furthermore, such designs are rarely scalable to new technologies and it is easy for a designer to lose performance when they commit to a specific design early in a design cycle. To make matters worse, when designers find their datapath to be suboptimal they can rarely afford to go back and redesign it. The continuing trend in the design of *application-specific integrated circuit*

(ASIC) is one of increasing complexity and density, making a trial and error approach increasingly difficult. This leads to the growing need for automated methods which can quickly yield good designs.

The ideal asynchronous design tool would allow designers to quickly generate the desired structure and provide information that would help them determine the best solutions. Each possible solution would be superior in at least one objective, such as size or latency, or in a combination of two or more objectives. This would give the designer the ability to test a variety of good solutions, helping to quickly and efficiently decide on a datapath structure that best implements a function. Automating the design and implementation of such a major portion of the chip would yield substantial reductions in design time, increase productivity, ease specification, modification, and enhance design re-usability.

1.2 Related Work

It is a common practice for synchronous circuits to be formally modeled and automatically synthesized. There are many existing tools which support automatic translation of an algorithmic-level specification to a register-transfer level representation [22]. The use of such models and automated tools for asynchronous circuits has been limited to synthesizing control circuitry. Thus, many systems exist for the synthesis of untimed asynchronous control circuits [27].

A number of different styles for designing asynchronous control circuits exist. One method is to constrain signals to change only one at a time. The system must allow each signal time to settle before other signals can change [41]. This is called the *fundamental-mode* restriction. *Burst-mode* extends fundamental-mode to allow for a set, or burst, of inputs to arrive concurrently, followed by a burst of outputs [17, 35, 45]. Another method, *delay-insensitive* [12, 19, 33] assumes that the delays in wires and gates are unbounded. *Speed-independent* circuits [6, 16, 32] are similar, but assume that wire delays are negligible. Most methods are based on the assumption that nothing is known about the delays between signal transitions. This means that the circuit must be constrained to work correctly even in cases which never occur in physical implementations.

For asynchronous control circuits, an emerging area of research embraces timed asynchronous circuits [34]. This method allows a lower and an upper timing bound to be assigned to the relationships between signals. These circuits make use of the timing

information to eliminate unnecessary circuitry and to increase performance.

At the architectural-level, tools that automate datapath synthesis are just emerging. Heuristic techniques for synchronous design have been extended to asynchronous circuits [5], but many require the designer to manually specify where resources are shared [2, 8]. Work has also been done by Beerel to extend the synchronous techniques in [25] by using a mixed-integer linear programming technique to yield globally optimal solutions.

This work is related to work previously done in synchronous architectural-level synthesis and also work done in the area of asynchronous control circuits. For synchronous architectural-level synthesis, a vast array of algorithms and tools have been proposed. In general, these optimization problems are intractable and their solutions depend on solving associated sub-problems.

The subproblems are usually also intractable and are often solved through the use of heuristics. The subproblems are categorized into general areas which include *binding*, *allocation*, and *scheduling*. Binding is the process of mapping an operation to a resource. Where several resources can perform the same operation, the problem is extended to a *module selection* problem. When more than one operation has the same type, *resource sharing* or allocation can be employed. Allocation determines the quantity of each type of resource used to implement the operations. Scheduling is the process of denoting each operation's start time subject to precedence constraints specified by a data flow graph.

To solve the scheduling problem, it is broken down in its simplest form to a unit-delay model in which all operations have equivalent delay. Different algorithms have been proposed to address constrained and unconstrained scheduling of individual operations. These algorithms include unconstrained *as-soon-as-possible* (ASAP) scheduling, and latency-constrained *as-late-as-possible* (ALAP) scheduling [18]. These algorithms are specific to synchronous design problems. This research modifies these algorithms for application to asynchronous optimization problems.

Scheduling with resource constraints is also very important because with resource dominated circuits, resource usage determines the circuit area. Solutions have been developed using an exact integer linear-programming model [24, 13]. This approach is suitable for medium scale examples, but fails to solve problems with a large number of variables or constraints. Another method is *force directed scheduling* (FDS) [36]. This method attempts to use the concept of force to optimally schedule operations. All these algorithms are currently restricted to synchronous design problems.

The timed models used for control circuits motivate the use of delay assumptions in datapath resources. When a timed model is applied to asynchronous datapath resources, the design evaluation space can be reduced, unnecessary circuitry eliminated, and increased performance achieved. The work described here is designed to be used in conjunction with the `ATACS` tool framework [34], which can further refine the generated asynchronous control circuitry. The result is a completely automated tool flow for refining asynchronous specifications from a behavioral level to a structural level.

1.3 Contributions

The focus of this work has been to explore and develop a method of architectural-level synthesis for asynchronous circuits. In particular, the issues of scheduling and allocation for asynchronous resources are confronted. While binding and resource selection are also important issues that can affect scheduling and allocation this study does not attempt to utilize their potential benefits at this time.

For asynchronous circuits to become a viable and superior alternative to synchronous circuits, good asynchronous computer-aided design tools need to be created. These tools, at a minimum, need to have comparable functionality to synchronous tools while maintaining a similar ease of use. Since developing such tools would be a very large and time consuming process, it is argued that asynchronous tools should build on work already done and that they should be as compatible as reasonably possible with current synchronous tools. This would expedite the transition for designers from synchronous design to asynchronous design without learning a completely re-engineered design process.

Scheduling optimization problems use synchronous techniques to find critical windows of time for resources with asynchronous delays. Relative timing of operations is used in conjunction with the analysis of the critical window of operations. From this, an estimate of the typical delay of each configuration may be made. Furthermore, for allocating resources to specific operations, a technique was developed that uses information from scheduling in conjunction with the information derived from the data flow graph. Using both sources of information, a heuristic algorithm efficiently solves the allocation problem.

Exploring all possible configurations to implement a given design is difficult because the number of possible solutions grows exponentially with respect to the size of the data flow graph. Several exact and heuristic filters to reduce the size of the design space are implemented. These filters are very effective in reducing the exploration time for the circuit

design. These filters include: pruning the design space when implied edges are detected, removing redundant designs from consideration, solving for a minimal-latency solution efficiently, and detecting when a maximal configuration is achieved without exploring an entire branch of the design space. Several case studies illustrate the effectiveness of these filters.

This study necessitated a CAD tool for experimenting with the various automatic methods of scheduling, allocation, design space exploration, and the effect of the proposed filters. The CAD tool `Mercury` has been developed for this purpose. Figure 1.1 shows the design flow of the tool. `Mercury` is unique in that it can take an abstract model of a design, in this case a data flow graph, and from that generate both an optimal structural view of an asynchronous datapath for the design, as well as the necessary behavioral control to coordinate that datapath. The generated structural view consists of an interconnected block diagram of functional units, latches, control, and multiplexors.

The generated asynchronous control can be refined further to logic gates using the existing `ATACS` tool. The end result is a fully specified asynchronous design which can be tested and verified. `Mercury` implements these ideas by generating output which can leverage synchronous tools for a common framework of simulation and functional verification. Furthermore, the development and use of `Mercury` demonstrates the idea that a CAD tool can generate a reliable and efficient asynchronous circuit for minimum cost and design time.

1.4 Thesis Outline

Chapter 2 discusses issues relating to the architectural modeling of asynchronous designs. `Mercury's` model input format and intermediate circuit representations are examined. The chapter continues with a description of then generation of a structural view of the datapath and a behavioral view of the control from an abstract model. The chapter concludes with an illustration of how the resource and constraint libraries are modeled. The exponential nature of the design space is reviewed in Chapter 3. The chapter continues with illustrations of how each configuration in the design space is evaluated using asynchronous versions of binding, scheduling, and resource allocation. These techniques are compared and contrasted with traditional synchronous methods. The study proceeds with a discussion in Chapter 4 of the proposed methods of using filters to reduce the design space. Several filters are presented, some of which are exact

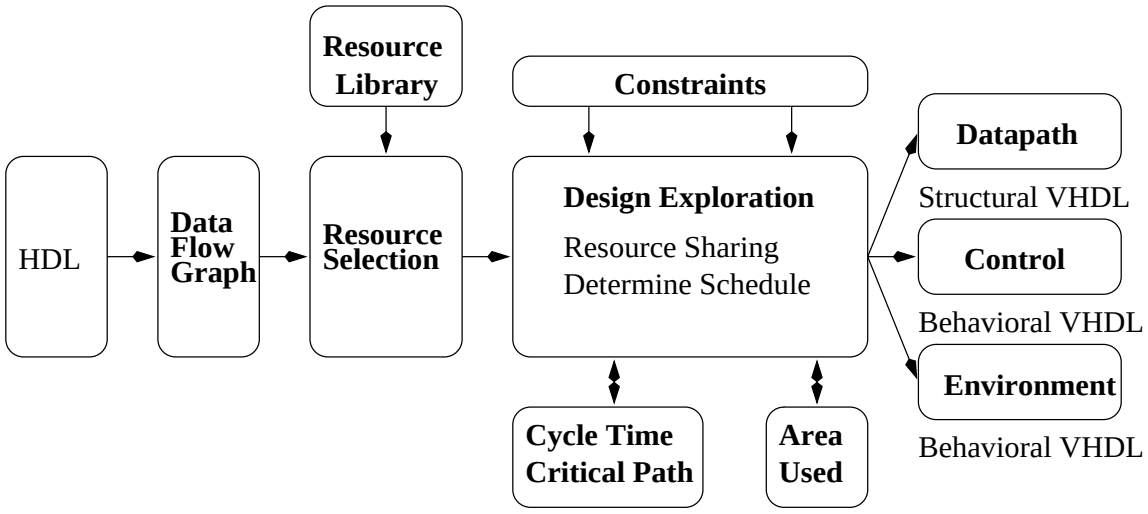


Figure 1.1. `Mercury` design flow.

and others which are heuristic. Chapter 5 describes the algorithm used to explore the design space. Optimizations used to reduce the execution time of exploration are also illustrated. These optimizations include using dynamic transitive closure and dynamic path analysis. Several case studies for using the presented methods to build asynchronous circuits are presented in Chapter 6. In the case studies, the effectiveness of each filter is given, along with examples of the resulting asynchronous circuits. Chapter 7 summarizes the contributions and results of this work and offers ideas for possible extensions.

CHAPTER 2

ARCHITECTURAL LEVEL MODELING

Mistrust endeavors which require new clothes.

—E. M. Forster

2.1 Representation and Modeling

It is often beneficial to simplify a circuit representation with a model. A useful model contains all of the relevant design features without including implementation details. These models give designers and CAD tools a common method of conveying information about a circuit. Circuits can be modeled differently according to the desired level of abstraction. Stages of abstraction include, but are not limited to, architectural, logic, and transistor. For example, at the architectural level, circuits are modeled showing required operations and their dependencies. At the logic level, circuits are modeled with interconnected logic blocks and logic networks. At the transistor level a physical view of the circuit is modeled.

Generally the design of a circuit progresses through these various tiers of abstraction until a physical view of the circuit is obtained. At each stage, the model of the circuit becomes less abstract as successively finer detail is introduced. Each level adds just enough information to capture essential features of that level. Before progressing to the next step, the model can be simulated, validated, and verified.

The top level of the design process, the most abstract, is the architectural-level. Here, the function of the entire system is described in algorithmic terms with the behavior of a circuit being modeled in a hardware description language (HDL). Consequently, this level of modeling is often referred to as behavioral modeling. An HDL provides well-defined semantics and syntax for a model. This gives a consistent and unambiguous representation of a specification which can be used to exchange information between designers and tools.

Although HDLs such as VHDL and Verilog evolved from traditional programming languages, they are different in many ways. For example, they generally default to

concurrent operations in place of statements which execute sequentially. In this regard, HDLs are related more closely to parallel programming languages than to traditional sequential programming languages. HDLs also allow for the definition of ports into and out of the circuit, along with their required data formats and parameters. HDLs place a large emphasis on the specification of detailed timing constraints for each circuit component. In addition, many of the HDLs support different views for a circuit. For example, a behavioral view and a structural view are typically supported. Architectural-level synthesis tools generally support the transformation of behavioral models into structural models.

Using such a formally defined model is beneficial for several reasons. First and foremost, when a system design is needed the system requirements can be specified unambiguously and completely. Engineers have the task of designing a system that meets customer requirements. Using a formal model to specify the system requirements reduces the risk of incomplete or ambiguous specifications. It also gives the engineer the opportunity to explore alternative implementations, and find the best design, given the customer's criteria.

Second, formal modeling allows for design validation and verification. Using a hierarchal approach, subsystems and subcomponents can be individually tested. At each level in the design hierarchy, the composite system can be tested and verified. While functional validation is useful, models can also be used as a starting point to formal design verification. Formal verification uses formal logic and rules of inference to deduce the correctness of a design. Formal verification is a complex problem itself and is an active area of research for both synchronous and asynchronous circuits [3, 7, 38]. While formal verification is not yet an everyday practice, there has already been significant progress in this area and there is an optimistic horizon in its future.

Finally, a formal model allows synthesizing a circuit automatically. If a design can be formally specified, it can, in theory, be translated to a circuit that performs that function. The automated generation of circuits is beneficial because it reduces the time of a design and thus, more time can be spent exploring alternative designs rather than being consumed with the details of a particular design. Furthermore, if the translation is automated and the translation process itself is verified, then confidence that the resulting circuit is correct rises.

In essence, a formal model used in conjunction with computer-aided design tools is

a means to achieving a reliable and efficient circuit for minimal cost and with minimum design time. By providing better tools for the design process, many errors can be avoided, delays minimized, and costs contained.

For this work, VHDL [4, 43] is used. VHDL is a verbose language used to specify and document large systems. It is used to model both the control and the datapath of a design. VHDL is employed to model the control, in part, because it can be simulated and verified using current synchronous CAD tools and also because a subset of the language can be successfully compiled into a format that can be synthesized into a timed asynchronous control circuit [46].

The first step in the design process is to take a formal behavioral model and translate it into a representation that illustrates the flow of data from one operation to the next. Figure 2.1 shows the behavioral VHDL representation of a differential equation solver with its corresponding data flow graph.

The refinement of the VHDL model into a good data flow graph is a difficult task because several optimizations are possible. These optimizations include: tree-height reduction, constant and variable propagation, common subexpression elimination, and dead code elimination. Each of these optimizations affects the resulting data flow graph, which in turn limits or enhances the synthesis process. Methods from compiler theory have been developed in [1, 42] which solve these and similar problems. It is assumed that

```
architecture BEHAVIOR of DIFFEQ is
begin
process
  variable x,a,y,u,dx,xl,ul,yl: in slv(7 downto 0);
begin
  wait until start'event and start = '1';
  x := x_port; y:= y_port; a := a_port;
  u := u_port; dx := dx_port;
  DIFFEQ_LOOP:
  while (x < a) loop
    xl := x + dx;
    ul := u - (3 * x * u * dx) - (3 * y * dx);
    yl := y + (u * dx);
    x  := xl; u := ul; y := yl;
  end loop DIFFEQ_LOOP;
  y_port <= y;
end process;
```

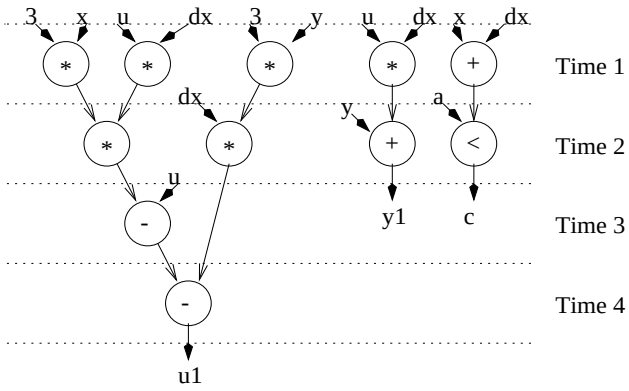


Figure 2.1. Behavioral VHDL and corresponding data flow graph.

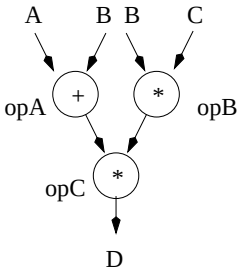
these optimizations have been made and translation to a data flow graph has already taken place, so `Mercury` accepts input in the format of a data flow graph similar to that shown in Figure 2.1.

A data flow graph is relatively simple to specify. Textually, the format given in Figure 2.2 is used. It first defines the name of the graph, then lists each of the primary inputs and primary outputs of the system with their corresponding sizes. Next, nodes in the graph are listed with their operation type, name, and size. The nodes are linked together using edges. An edge can exist between any two nodes as long as the edge does not create a cycle in the graph. Finally, inputs are linked to nodes which utilize them, and outputs emanate from nodes that produce them.

It should be noted that the user can optionally specify an ordering for input operands to a node. This can be important when an operation, such as subtraction, is non-commutative. If an ordering is not specified, then the operations are evaluated according to alphabetical order. Using this format allows the function of the circuit to be modeled. Figure 2.3 shows a sample data flow graph and its corresponding specification. This model permits a design to be further refined using binding, scheduling, and allocation.

dfg *name* {
  input in-name bit-width
  ...
  input in-name bit-width
  output out-name bit-width
  ...
  ouput out-name bit-width
  node operation node-name bit-width
  ...
  node operation node-name bit-width
  edge node-name → node-name port
  ...
  edge node-name → node-name port
  datain in-name → node-name port
  ...
  datain in-name → node-name port
  dataout node-name → out-name
  ...
  dataout node-name → out-name
}

Figure 2.2. Input format for a data flow graph (DFG).




```
dfg sample {
    input A 32
    input B 32
    input C 32
    output D 32
    node + opA  32
    node  * opB  32
    node  * opC 32
    edge opA -> opC
    edge opB -> opC
    datain A -> opA
    datain B -> opA
    datain B -> opB
    datain C -> opB
    dataout opC -> D
}
```

Figure 2.3. Sample data flow graph and model description.

2.2 Modeling Resources

To unify the design process, some underlying requirements must be made about functions in the resource library. Here it is required that each resource in the library is an asynchronous system that follows a specific communication protocol. None of the resources are synchronized by a global clock. A protocol is a sequence of events in a communication transaction. Many handshaking protocols exist, such as two-phase transition signaling or four-phase signaling. For these two predominate methods, there is ongoing debate concerning the better choice. The four-phase protocol requires twice as many actions as two-phase, but the actions are usually simpler. In general, when operator delays dominate communication costs, then four-phase is better. Four-phase may also be better for precharged arithmetic units since the return to zero naturally fits the precharge phase. When transmission delays dominate communication costs, two-phase is better.

`Mercury` currently supports only the four-phase protocol with a bundled data path. The bundled-data approach uses a set of control wires to indicate the validity of a set, or *bundle* of data wires. Similar self-timed modules that follow a two-phase protocol are used in [11]. In either protocol, the control wires for each bundle include two signals. The first signal is used to request (REQ) an action. Once the receiver of the request has completed its function it sends an acknowledgement (ACK) back to the sender to complete the transaction. Figure 2.4 illustrates the request/acknowledge protocol. With

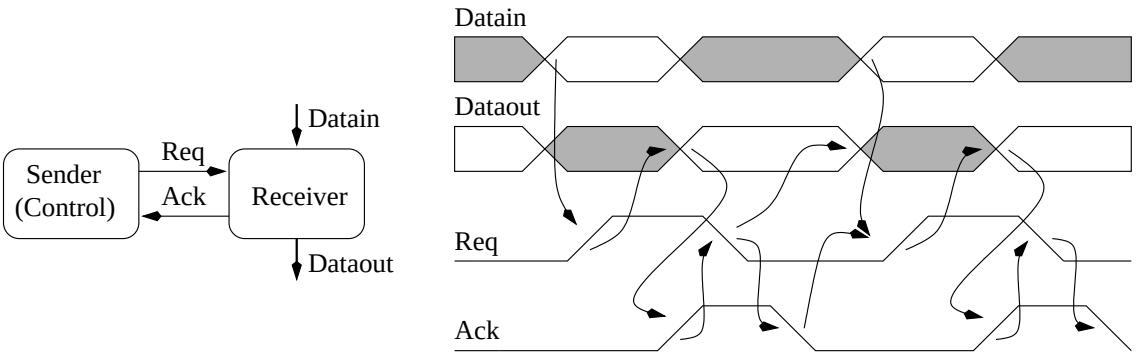


Figure 2.4. Request/acknowledge interface with four-phase handshake protocol.

a four-phase protocol, each signal transition is considered along with its direction of movement. For example, a rising request is distinguished from a falling request. There are several methods of employing the four-phase handshake, including, *early*, *late*, and *broad* protocols [20].

`Mercury` uses the *early* protocol in which the rising edge of the request line indicates that data is available, and the rising edge of the acknowledge line indicates that the computation has been completed and the sender no longer needs to hold the inputs stable. The falling edge of the ACK signal resets the component to an *available* state. The bundled data approach requires data in the bundle to be valid at the receiver before the receiver sees a change on the control signals. In figure 2.4 the light regions indicate when data are valid and the shaded regions indicate when data are invalid.

For simplicity, it is assumed that the interface of each device is delay-insensitive. This means that the protocol is insensitive to delays through circuit components and the wires that connect them. Obviously, this does not accurately model the physical properties of system components and wires. This makes building a truly delay-insensitive circuit difficult, as demonstrated by [30]. This issue is left for the next level of refinement in the synthesis process.

Although self-timed delay-insensitive circuits involve signaling overhead for the handshake communication, they offer several appealing advantages. Generally, they give better performance than synchronous systems because they tend to reflect average-case delays rather than worst-case delays for a system. In some cases, this alone can be a major performance benefit. Second, they allow a system to be upgraded incrementally. Each

component can be individually replaced without changing or doing extensive redesign of the entire system. Third, very robust systems can be implemented because timing and functionality are separated. For example, when a circuit is required to operate over a wide range of voltage and temperature conditions, self-timed systems are ideal because they easily adapt to their environment. Finally, self-timed components allow the construction of systems in a hierarchical and uniform fashion. This characteristic is very beneficial because designs can be assembled without considering detailed timing characteristics. When timing characteristics are available and considered, the circuit can be more aggressively optimized.

Using the self-timed circuit methodology, asynchronous resources are annotated in the resource library with timing information. This information is used to optimize the configuration of resources. Each supported function of a given resource is modeled with a minimum, maximum, and typical computational delay. These correspond to the data-dependent computational delays of each function. For this work, it is required that operations have bounded delays. In addition, the area and the bit-width of each resource must be given. The order in which the functional units are listed for a resource determines the operation select code used by control to select the proper operation. Figure 2.5 shows the input format of the resource library. The user can use standard operations such as addition, subtraction, and multiplication, or can create more complex custom resources and operations.

For the best results, the parameters of each resource should correspond to the physical properties of the resource. Each library generally maps to a specific technology. This

```
drl name {
    resource-name bit-width area
      operation-type [min-delay,max-delay,typ-delay]
      ...
      operation-name [min-delay,max-delay,typ-delay]
    ...
    resource-name bit-width area
      operation-type [min-delay,max-delay,typ-delay]
      ...
      operation-name [min-delay,max-delay,typ-delay]
  }
```

Figure 2.5. Input format for the datapath resource library (DRL).

gives a modular approach that leaves room for expansion to future technologies without requiring a change in the specifications of a design. A small sample library is shown in Figure 2.6.

2.3 System Constraints

Constraints on the design can also be specified. These are valuable because they can focus the design search, reducing the time required to achieve a good solution. The user can specify an upper limit on both the desired area and the desired delay. For example, if the user sets a maximum desired delay for a system, the tool stops exploration (evaluating all possible configurations) of any branch where that value of latency is exceeded, yielding a significant savings in execution time.

The user can also specify the maximum number of instances for any particular resource type. The format for specifying these values is shown in Figure 2.7. The use of constraints is optional, but they are usually beneficial for large designs, because the more constrained a design is, the quicker a good solution can be found. A sample constraints specification is shown in Figure 2.8.

```
drl myLib {
    ALU 32 452
      + [12,28,15]
      - [12,30,16]
    Multiplier 32 671
      * [34,82,61]
  }
```

Figure 2.6. Sample datapath resource library (DRL).

```
constraints name {
    max-area = val
    max-delay = val

    resource-name = val
    ...
    resource-name = val
  }
```

Figure 2.7. Input format for constraints.

```
constraints myCon {
   max-area = 923
   max-delay = 102
   ALU = 1
   Multiplier = 1
 }
```

Figure 2.8. Sample constraints specification.

2.4 Output

At this level, the goal of refinement is to generate an optimized structural view of a system from a behavioral description. Using the three sources of information—a data flow graph, a library of functional resources, and a set of constraints–all the necessary information to refine the system is available. In `Mercury`, after a data flow graph has been bound, scheduled, and allocated, a structural view of the datapath is generated and a behavioral view of control for the datapath is also generated. Both views are specified by `Mercury` using VHDL. The following sections describe these processes.

2.4.1 Datapath Generation

The datapath is organized as follows. First, latches are instantiated for each of the primary inputs, outputs, and data edges in the data flow graph. Latches for the outputs are not always required. They are only required when a functional unit is resued for other operations after the output is generated. In other words, in cases where the functional unit can not hold the output for the duration of the system. This relaxes the requirement for latches on all outputs. This optimization however, is not currently implemented. Next, Each functional unit is instantiated and when more than one operation is mapped to the same functional unit, multiplexors are used to route the appropriate operands at the appropriate time. At the output of the functional unit, latches are used to carry the data to the next operation, or to the primary outputs. This datapath format is illustrated in Figure 2.9.

Figure 2.10 shows one possible arrangement for the datapath generated by the `Mercury` tool from the specification given in Figure 2.3. The datapath works in the following way. When the global request signal, *sample_req*, is asserted, data from the primary inputs are latched. When all of the required operands for a given computation are available, and the functional unit is available, the computation begins. In the example given in Figure 2.10,

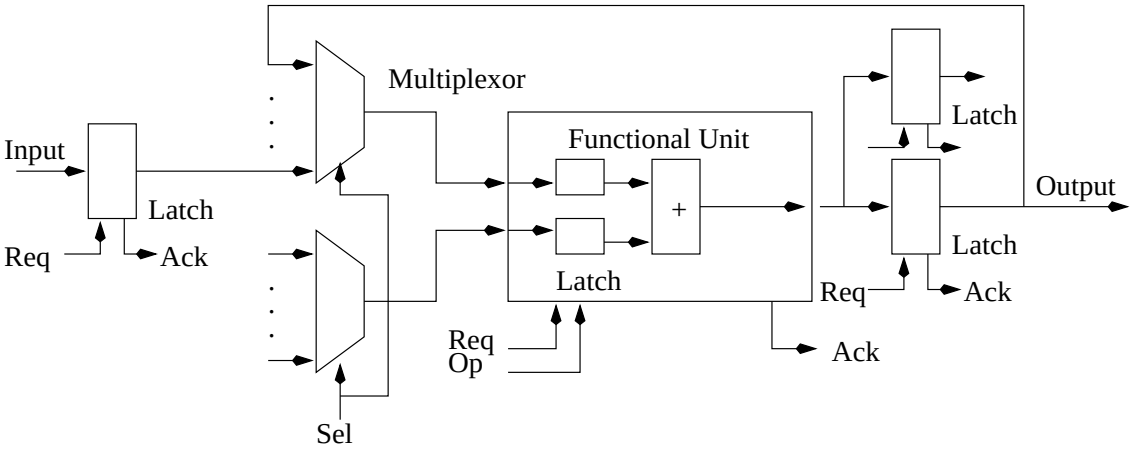


Figure 2.9. Datapath format.

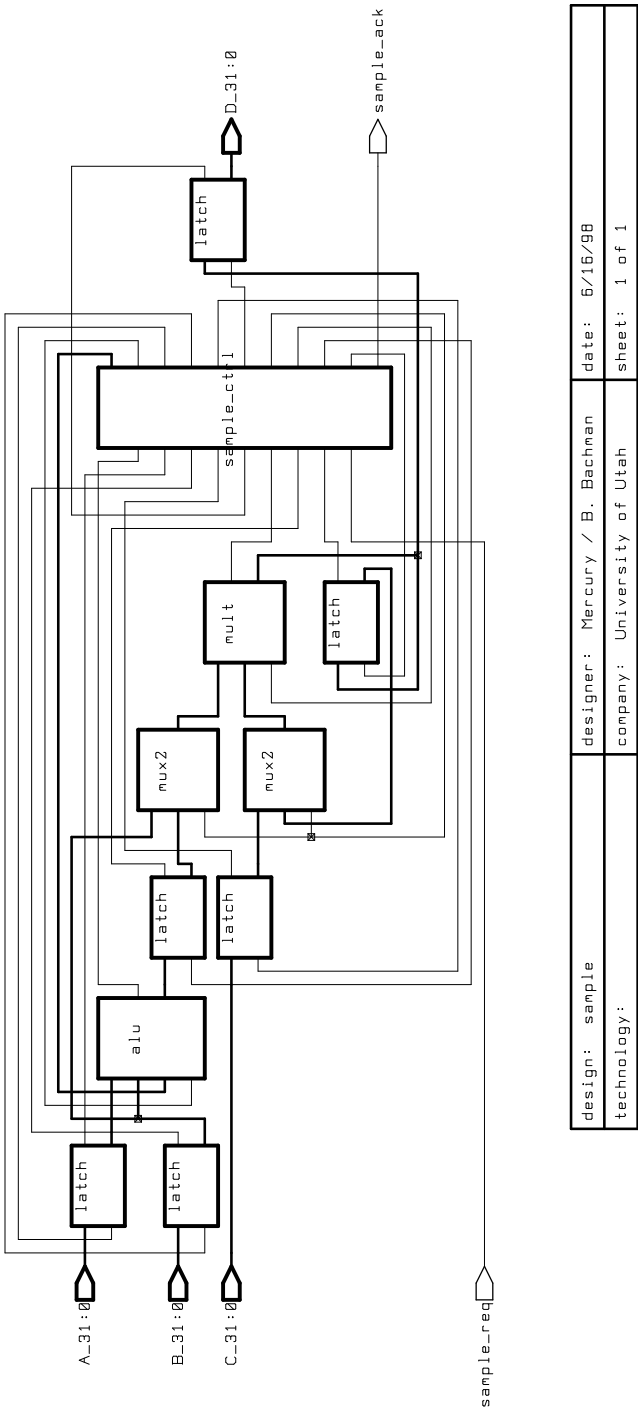


Figure 2.10. Datapath generated from sample model description.

the multiplier is shared between two operations, requiring two 2x1 multiplexors to be used (one for each operand).

After each computation is completed, the results are stored in a latch for future use by other operations and the functional unit then becomes available for its next operation. Finally, the system acknowledges on the *sample_ack* signal it has completed its computation when all of the primary outputs have latched their data.

While the entire system is modeled in this way, each subcomponent of the system is also modeled with a similar protocol. In the example, the ALU and multiplier, when requested, also latch their inputs, and hold their outputs until their results are acknowledged by their environment. This allows components to be generated and used in a hierarchal fashion. The only exception to this protocol is a multiplexor component, because it does not latch its inputs or its outputs.

Additional optimizations can be made to the datapath to reduce the number of latches required to implement a system. Sharing latches that hold data for disjoint periods of time is one such optimization. This, however, is a difficult problem to solve because of the inherently asynchronous lifetime of data. In theory, this problem is similar to the resource sharing problem, in that the sharing of latches can incur the use of additional multiplexors, which in turn require more area, and the additional complexity usually complicates the control further. For these reasons, and in order to simplify the generation of the datapath and control, this optimization is not currently applied.

To generate the datapath, `Mercury` takes a bound, scheduled, and allocated data flow graph and builds a VHDL model. A sample VHDL structural model for the datapath is listed in Appendix A. This model corresponds to the behavioral model shown in Figure 2.10. The generation of the model begins by first defining the entity of the primary system with primary outputs, inputs, and handshaking signals. For the structural architecture of the entity, the components used by the system are declared. In the example, the components ALU, Mult, mux2, latch, and CTRL, are all declared. The components refer to resources defined in the library shown in Figure 2.6, except for the controller component, *CTRL* and multiplexors, which `Mercury` generates.

Next, intermediate signals are declared. These are the signals which are used between the ports of the latches, multipliers, controller, and functional units. All of these signals are internal to the system. After the signals are declared, each of the required components is then instantiated. Where more than one instance of a device is used allowing concurrent

operation, multiple instantiations are declared. Where operations are serialized and more than one operation is mapped to an individual resource, multiplexors are instantiated to select the data. Finally, the ports of each of the components are wired up to the appropriate signals to complete the design of the datapath.

2.4.2 Control Generation

Generating the control corresponding to a particular configuration of the datapath is determined, in part, by the protocol used between components. Each of the components in the datapath follows a four-phase handshake protocol using request and acknowledge signals. Following this protocol, `Mercury` builds a control module for each datapath configuration.

The control is built using the request and acknowledge signals from each of the components, such as functional units, latches, and the primary request and acknowledge signals. When multiplexors are used, their select bits are generated by the control but no handshaking signals. Using VHDL, the communication protocol of each component is modeled with VHDL processes. When a process is activated during simulation, it starts executing from the first statement and continues until it reaches the last. All statements in a process take zero simulation time except for "wait" statements. So, it is only through the execution of "wait" statements that simulation time advances. Each process executes concurrently with respect to other processes. The behavioral VHDL for the controller is shown in Appendix B.

Each instance of a latch in the system is modeled in the control with an individual process. Each latch is initially unoccupied. Latches on primary inputs wait on the primary request of the system. The data are latched when the primary request is received. These latches return to an available state when the entire system has been acknowledged. Shown below is an example of this type of process:

```
-- controls latch l_A at the source
proc5:process
begin
    wait until sample_req = '1';
    A_req <= '1' after delay(2,4);
    wait until sample_req = '0';
    A_req <= '0' after delay(2,4);
end process;
```

Each functional unit instance is modeled with a unique process. Because each func-

tional unit can be used for more than one operation, these processes control the ordering of each operation using the resource. This is done by waiting for the appropriate operands to become available. When they become available, the multiplexors are set, and the correct data is steered to the functional unit. The functional units reset to an available state after they have completed their operations. Shown below is an example of this type of process:

```
-- controls resource Mult_1
proc9:process
begin
  wait until Mult_1_ack = '0' and B_ack = '1' and
             C_ack = '1' and sample_req = '1';
  Mult_1_mux2_sel <= '0' after delay(0,1);
  Mult_1_req <= '1' after delay(2,4);
  wait until l_2_ack = '1';
  Mult_1_req <= '0' after delay(2,4);
  wait until Mult_1_ack = '0' and l_1_ack = '1' and
             l_2_ack = '1' and sample_req = '1';
  Mult_1_mux2_sel <= '1' after delay(0,1);
  Mult_1_req <= '1' after delay(2,4);
  wait until D_ack = '1';
  Mult_1_req <= '0' after delay(2,4);
  wait until sample_req = '0';
end process;
```

Intermediate latches between operations wait for acknowledgement from the preceding computation before latching the resulting data. The latch then waits for the resource that uses that data to be requested before returning to an available state. Where many operations are performed by a single resource, the select bits of the multiplexor are used to ensure the resource is performing the right computation on the correct data values. Sequencing of operations is handled by the control. Shown below is an example of this type of process:

```
-- controls latch between nodes opA and opC
proc1:process
begin
    wait until ALU_1_ack = '1';
    l_1_req <= '1' after delay(2,4);
    wait until Mult_1_req = '1' and Mult_1_mux2_sel = '1';
    l_1_req <= '0' after delay(2,4);
end process;
```

When all the primary outputs of the system have acknowledged, the system's primary

acknowledge is asserted. After the environment responds by lowering the request signal, the entire system is reset to an available state. Shown below is an example of this process.

```
-- controls the ack of the entire system
proc3:process
begin
    wait until D_ack = '1' and sample_req = '1';
    sample_ack <= '1' after delay(2,4);
    wait until D_ack = '0' and sample_req = '0';
    sample_ack <= '0' after delay(2,4);
end process;
```

Using this control specification, `Mercury` is compatible with synchronous VHDL simulators. Figure 2.11 shows a wave diagram for the handshaking signals of the control. The simulation was done using ViewLogic's VHDL simulator FusionSpeedwave.

In addition, this model of the control is compatible with the asynchronous CAD tool `ATACS` [46, 34], which is designed to further refine the control to a gate-level model. Figure 2.12 shows a view of the control generated by `ATACS` using the behavioral VHDL description. The control has 40 literals and requires 94 transistors to implement.

For ease in compilation, simulation, and verification, a VHDL configuration is often very useful. The configuration maps each instance of a component to a specific VHDL architecture model of that component. Appendix C shows a sample configuration generated by Mercury.

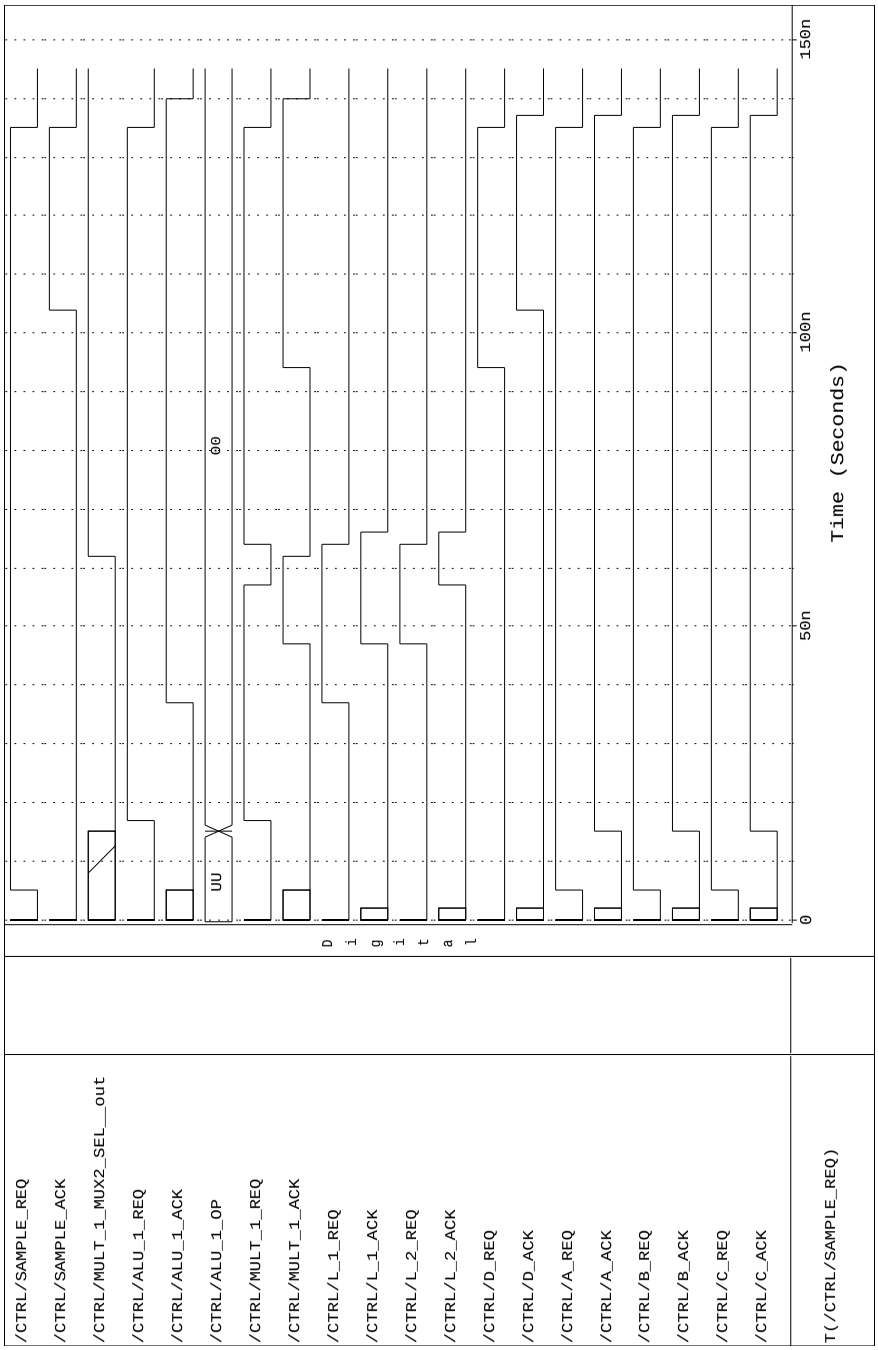


Figure 2.11. Timing diagram showing handshaking protocol.

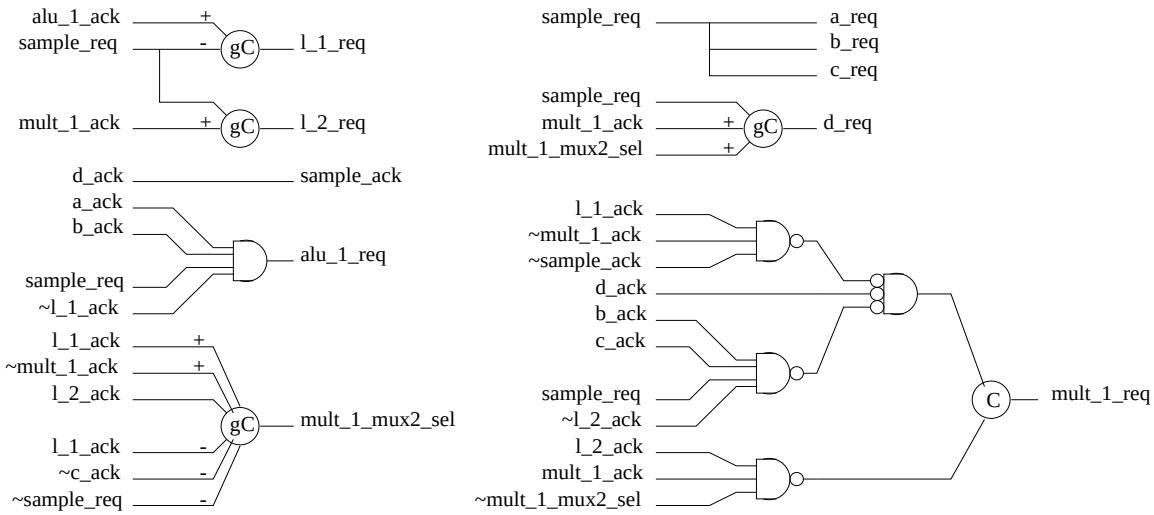


Figure 2.12. Structural control generated by the ATACS CAD tool.

CHAPTER 3

DESIGN SPACE EXPLORATION

The real voyage of discovery consists not in seeking new landscapes, but in having new eyes.

—Marcel Proust

Many different valid hardware implementations may exist for a given data flow graph, each with a specific configuration of hardware resources and corresponding control logic. The set of possible configurations is the *design space* of the system. For efficiency and structure, the exploration of the design space is divided into three main subtasks: binding, scheduling, and allocation. As discussed in Chapter 2, binding determines a mapping between operations in the data flow graph and resources in a library. Scheduling determines when operations are executed, and allocation determines which resources can be shared between operations, giving the quantity of each type of resource used to implement the operations in the data flow graph.

Some systems perform binding and allocation followed by scheduling [14, 31]. In these methods, delay and area information is estimated and then back-annotated for verification. Other methods perform scheduling before binding and allocation [21, 40]. This approach works well for resource-dominated circuits such as DSP and processor designs. ASIC circuits, however, are generally not resource dominated but control dominated, and therefore perform binding before scheduling and allocation. This permits estimating the required steering logic and also allows for a more precise assessment of delays. In this case, the scheduling problem is not constrained by binding or allocation and it can be solved efficiently. Binding and allocation, however, generally dominate the complexity of the problem [18, 15].

In an asynchronous circuit, an operation can execute as soon as the resource to which it is bound is available and all of its data inputs are available. Since time steps are not explicit and timing is not discrete, it is unclear how to effectively apply the tasks of binding, scheduling, and allocation. `Mercury` uses a hybrid approach in which it

directly extends the principles of synchronous scheduling. First, binding is performed, then scheduling and allocation are performed to determine the timing and allocation of resources. When accurate models are used for area, delay, and interconnect, this approach can work for both resource-dominated and non-resource-dominated circuits.

3.1 Binding

The first step is to create a binding for the operations in a data flow graph. This determines a mapping between operations in a data flow graph and resources. A binding may associate more than one operation to a specific resource type in the library. A covering relationship can be defined among types to represent the fact that a resource implements more than one operation. For example, an ALU may cover several operations like addition and subtraction.

In order to produce a valid binding it is necessary that all operations are bound to a library resource. Where this is not the case, a *partial binding* is created. `Mercury` does not consider nor allow partial bindings because this would prevent a data flow graph from being accurately scheduled and allocated.

There is currently a limited supply of asynchronous components in which to build a circuit. Therefore, `Mercury` does not focus on binding and resource selection. While binding and resource selection are important issues, their potential benefits are not utilized at this time. For simplicity, `Mercury` requires that the library contains at least one resource to satisfy each operation type used in the data flow graph. Then, to perform binding, `Mercury` binds the first resource in the library which satisfies the operation.

3.2 Scheduling

For synchronous systems, scheduling determines when operations are executed in time. This can be done efficiently using discrete-time intervals based on a global clock. In an asynchronous circuit, the absence of a global clock and the asynchronous timing of events make scheduling difficult. The scheduling of resources is dependent only on the availability of the resource and its inputs. For accurate asynchronous scheduling, resources must be modeled with data-dependent completion delays. Since binding is done prior to scheduling, delay information is extracted from the given binding.

Traditional scheduling of synchronous designs assigns a given start and finish time with each operation. In asynchronous scheduling, the start and finish times of operations have a limited use because data-dependent delays have a significant impact on performance

[9]. While scheduling information is useful, it has a limited use in asynchronous design because it is very difficult to break time into discrete bins. Furthermore, discrete methods rapidly become computationally infeasible as discretization constants are made small to allow for finer granularity.

For these reasons, scheduling information is not used here to explictily schedule an operation to a specific time. It is only used to determine conservative windows of time in which a operation *may* occur. The actual schedule is determined by resource sharing and the order of operations. This makes the scheduling of operations for asynchronous design an optional task. It is shown later, however, that this scheduling information can be very helpful in reducing the time required to find a good solution. We leverage synchronous methods to find the potential start and finish times of each operation.

For these tasks, two common algorithms are used. `Mercury` uses the synchronous method of ASAP (as-soon-as-possible) scheduling to find the conservative windows of time in which an operation may be utilizing a resource. ALAP (as-late-as-possible) scheduling is used in conjunction with ASAP scheduling to find the *mobility* of each operation in the data flow graph [18]. The following sections describe these methods.

3.2.1 ASAP Scheduling

ASAP scheduling, or scheduling without resource constraints, is used to determine a lower bound on the latency of the system. ASAP scheduling is solved in polynomial time by iterating through the nodes of the data flow graph in topological order. Each node is scheduled by setting its start time to the maximum ending time of all its predecessors. The ending time of each operation is computed by adding either the minimum, maximum, or typical delay of the operation to its starting time. This gives three ASAP schedules for each operation.

Figure 3.1a illustrates how the synchronous ASAP algorithm would schedule resources for the differential equation solver. The ASAP algorithm is shown in Figure 3.3. Because resources can be scheduled without limit, configurations that limit the number of resources only have a latency greater than or equal to that of the unconstrained ASAP schedule.

By design, operations in an asynchronous system always start as soon as they can. This means the difference between a minimum (best-case) ASAP schedule and maximum (worst-case) ASAP schedule yields a range of time in which the operation starts its computation. In this manner, scheduling takes on a nontraditional definition. Namely, scheduling involves denoting an operation with a window of time when a resource is most

likely used for a given operation. These ranges of time are called *critical windows* because these windows are used during resource sharing to determine the range of time when other operations should not try to utilize a given resource to avoid resource conflicts. Resource conflicts lead to a loss in performance because one operation may have to wait for the resource to become available.

Using the data flow graph from Figure 3.1, each node is scheduled as-soon-as-possible using a minimum, maximum and typical delay. For this example, it is assumed that each multiplication operation has a minimum delay of four, a typical delay of five, and a maximum delay of six. All other operations have a minimum delay of one, typical delay of two, and maximum delay of three. Figure 3.2 shows the critical windows for the start times of each operation determined by ASAP scheduling for the differential equation solver example.

3.2.2 ALAP Scheduling

ALAP scheduling is the complement of ASAP scheduling and is used for latency-constrained scheduling. In this case, operations are scheduled as late as possible by setting the finish time for each operation to be the minimum start time of all of its successors. Again, like the ASAP algorithm, unless explicitly constrained, resources can

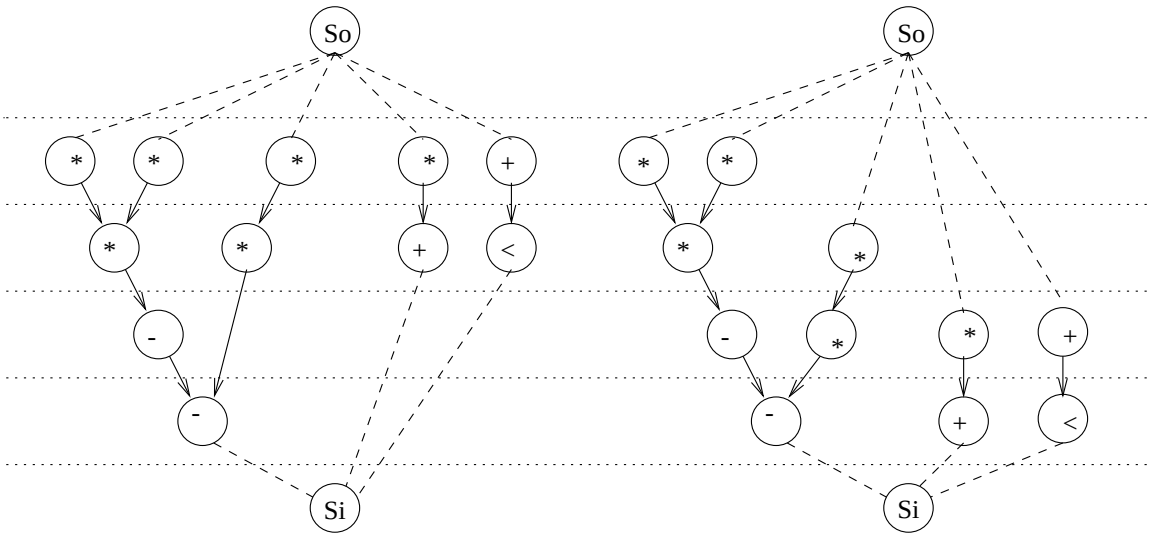


Figure 3.1. As-soon-as-possible and as-late-as-possible scheduling.

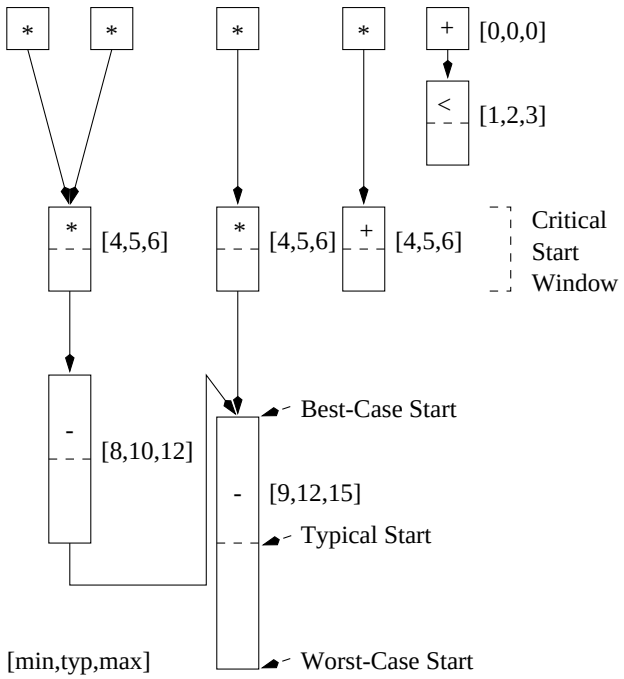


Figure 3.2. Critical windows derived from as-soon-as-possible scheduling.

be used without limit.

The ALAP algorithm is shown in Figure 3.3. The variable κ is the latency bound, and is chosen to be the delay of the schedule computed by the ASAP algorithm. Again, best-case, worst-case, and typical ALAP schedules are derived using minimum, maximum, and typical data-dependent resource delays respectively. Figure 3.1 illustrates how the synchronous ALAP algorithm would schedule resources for the differential equation solver. The computational complexity of both ASAP and ALAP is O(V+E).

3.2.3 Mobility

Using the difference of ALAP and ASAP scheduling, the mobility of each operation is computed. This is an important quantity because it represents the span of time in which an operation may be started. To illustrate, assume that for a specific operation the best-case ASAP start time is 5, and the best-case ALAP start time is 18. Then the mobility of the operation is 13.

If an operation has zero mobility then it is started only at a single given time, or else the schedule would exceed the calculated latency. The critical path of the system

```
ASAP (G(V,E)) {
  foreach v_i in topological order
    schedule t_i^S to max (t_j^S + delay_j) where j:(v_j,v_i) ∈ E
  return(t^S);
}

ALAP (G(V,E),κ) {
  Schedule v_n by setting t_n^S = κ
  foreach v_i in reverse topological order
    schedule t_i^L to min (t_j^L - delay_i) where j:(v_i,v_j) ∈ E
}
```

Figure 3.3. As-soon-as-possible and as-late-as-possible algorithms.

is determined when the latency bound of ALAP scheduling is set to that given by the ASAP algorithm. Then, taking the difference between scheduled operations according to ASAP and ALAP, each operation that has zero mobility is on the critical path of the system.

3.2.4 Force-Directed Scheduling

Another method used for discrete time based methods is Force-Directed Scheduling [36]. This method attempts to balance the concurrency of operations assigned to functional units. To do this, a concept of force is developed for each resource. One common analogy is to view the force of each operation as a spring. Then, a dataflow graph can be viewed as a set of springs pushing against their successors and predecessors. Additional forces to account for the sharing of functional units is also taken into consideration. With this model, a state of equilibrium is found between the forces. When this occurs, the concurrency of operations assigned to functional units gives the sharing of resources. An example of a schedule for the differential equation solver using this method is shown in Figure 3.4. Using this method, only two multipliers and two ALUs are required to complete the system in four time steps. This is an improvement over the ASAP method which requires four multipliers, and the ALAP method which requires three ALUs to complete in the same amount of time. Later, our results are compared against the FDS algorithm, and a similiar method called Force-Directed List Scheduling (FDLS) [36].

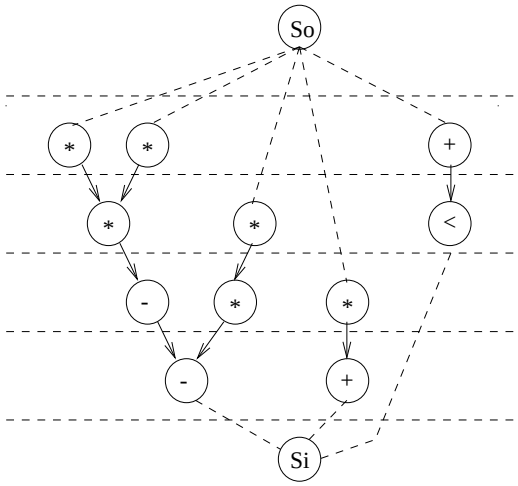


Figure 3.4. Force-directed scheduling.

3.2.5 Statistical Delay Calculation

It is critical that accurate estimates be used for asynchronous scheduling in order to achieve optimal scheduling and resource sharing. The delay of a system is more accurately calculated by using the statistical analysis approach presented in [9]. This method models operations with a probability distribution representing the likelihood of completion after a given amount of time. To illustrate, consider the example in Figure 3.5. In this case, operation D cannot begin its computation until all three of its incoming operands A, B, and C are available. An approach that simply finds the starting time of D by taking the maximum completion delay of A, B, and C is shown in [9] to potentially underestimate

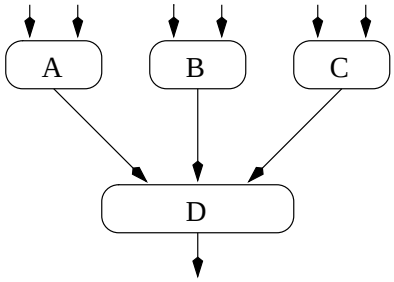


Figure 3.5. A data flow graph with four operations: A, B, C and D.

the performance of the system by as much as 21%. This is because worst-case timing analysis only calculates a pessimistic delay.

The statistical delay method is more accurate than these methods. It first calculates the uncertainty of the start time for the system by using delay distributions of operations A, B, and C. In this example, it is assumed that the completion times are normally distributed. The resulting distribution represents the window of time in which an operation may start its calculation. For any time t, assuming statistical independence on each of the inputs A through C, the distribution of D can be calculated. For the probability density function of A, the notation f_A is used. So, the probability that operation A has completed by time t is:

$$P(A_f \leq t) = F_A(t) = \int_0^t f_A(t)dt$$

This means the probability that D can *start* its computation at time t is:

$$P(D_s \leq t) = P(A_f \leq t) \cdot P(B_f \leq t) \cdot P(C_f \leq t)$$

By substituting the first equation into the second for A, B and C and taking the derivative of both sides, the start time distribution for D is:

$$f_D(t) = F_A(t) \cdot F_B(t) \cdot f_C(t) + F_B(t) \cdot F_C(t) \cdot f_A(t) + F_A(t) \cdot F_C(t) \cdot f_B(t)$$

Finally, to find the distribution of time for which the operation D is finished, the start time distribution of D is convolved with its computational delay distribution. The ending time distribution is then propagated in a like manner to D's succeeding operations. Using this model the data flow graph can be analyzed and a timing model can be generated.

It is clear that asynchronous scheduling using this method is more accurate but is a computationally expensive task. The shortcoming of this method is that the assumption of independence between signals is not always valid. Systems that have diverging signals from a common source operation that then reconverge at a later point in the data flow graph should, in theory, exhibit some dependence. This means that the assumption of independence in this method could lead to erroneous calculations.

3.2.6 Monte-Carlo Delay Calculation

Another method for computing the delay of the system is Monte-Carlo. The Monte-Carlo technique simulates the system until the overall delay of the system converges to a

specific time. Since in this method the delay of the system is calculated using Gaussian random variables to model the delays of each operation, Monte-Carlo yields the typical delay of the system.

The advantage of the Monte-Carlo method is that it takes into account signal dependencies. The drawback of the Monte-Carlo method is the large processing time required for convergence, making this type of delay estimation unsuitable during synthesis.

A couple of observations are in order at this point. First, the ASAP and ALAP scheduling techniques are suitable for calculating the conservative schedules for a system. From these schedules, the window of time in which an operation starts and completes can be found, albeit conservatively. Using the ASAP and ALAP schedules, it can be determined which resources are, or are not, in conflict and then share them appropriately. The ASAP and ALAP method cannot determine exactly when, in the typical case, an operation starts and completes. When a more accurate technique is required, the Monte-Carlo method can be used.

3.3 Typical Delay

Although very elusive, it is important to have some notion of the typical delay of a system, because in an asynchronous design, minimizing the typical delay is the primary objective when trying to reduce the overall latency. The worst-case, or even best-case, completion delays of two designs can be equal, but each can have different typical delays. Analyzing a design does not require knowing the typical system delay, it only requires knowing, with some degree of accuracy, if the typical delay is better or worse than a competing design.

It was originally thought that as operations are serialized to enable resource sharing the values of the worst-case delay and typical delay of the system would increase together. This would allow us to simply use the worst-case delay of the system as an indicator of the typical delay. Using this method, it seems logical that if a design had a larger worst-case delay, it would then also have a larger typical delay. This, however, is not the case, because it is possible for the worst-case delay of the system to increase, while the typical delay of the system actually decreases.

Using the differential equation solver, from Figure 2.1, the worst-case delay of the system was compared with the Monte-Carlo typical delay of the system. For the addition, subtraction, and comparison operations (ALU operations), delays of two, five, and eight

are used. These correspond to the minimum, typical, and maximum delay of each operation. For all other operations, delays of four, five, and six are used. Note that for this example, all operations in the data flow graph have the same typical delay. Only the best-case and worst-case delays vary. Exploration is then done on the system to find all configurations of the system. It was discovered, that as operations in the graph were serialized that the typical delay of the system did not track the worst case delay of the system. While the worst case delay of the system monotonically increased, the typical delay increased as expected, but it also decreased with certain configurations.

Further analysis of the differential equation solver example showed that this occurs because each ALU on the critical path improves the typical delay by three units. When other operations are on the critical path, they only improve the typical delay by one unit. In other words, it is better for the critical path to be composed with ALUs in place of other operations, because while the worst-case delay may increase, the typical delay is better. This illustrates that the typical delay of the system does not always track the worst-case delay of the system.

3.4 Resource Allocation

Resource sharing is used to minimize the area required for a design. This is done by determining which operations are scheduled to a particular instance of a resource. An optimum resource sharing is one that minimizes the number of required resource instances. Two or more operations can share the same resource if they are of the same type and they are not in conflict with each other. Operations are in conflict if their execution windows overlap in time. This happens when either operation starts before the other has completed. Operations that are scheduled in disjoint windows of time are guaranteed not to overlap and are, therefore, always compatible. The conflict window is determined by using the best-case ASAP schedule to determine the start time of the window and the worst-case ALAP schedule to determine the stop time of the window.

Another way to show that two operations are compatible is to analyze the data flow graph. If there is a path from operation i to operation j, then those two operations are compatible regardless of their scheduled windows of time. This is because the existence of a path guarantees that operation i must complete before operation j begins. The more edges present in a graph, the more sharing that can potentially occur. Edges used to explicitly denote two sharable operations are known as *resource edges* and are added to

the data flow graph during exploration. They are distinguished from *data edges*, because they do not imply the transfer of data from one operation to the next. Resource edges enforce that the two operations occur at disjoint times and denote the ordering in which operations must occur.

Figure 3.6 shows a data flow graph with only data edges. In this configuration, four multipliers are required and three ALUs. With resource edges, only two multipliers and one ALU are required. Granted, however, in this case, the overall delay of the system may not be equal. Note that there are many ways to add resource edges to the graph. Each resource edge added to the graph, in essence covers an aggregate of all the possible discrete time schedules that the given operation sequencing and resource sharing would produce. Hence, scheduling of operations is done independent of the discretization of time. For efficiency, `Mercury` utilizes both the information from the data flow graph and where applicable, conservative scheduling information to perform resource sharing.

This approach is beneficial for asynchronous design because the computational complexity is constant with regard to the discretization of time. Using resource edges, in effect, allows scheduling to take on a continuous time paradigm. Our tests showed that synchronous methods, such as Force-Directed Scheduling, become computationally

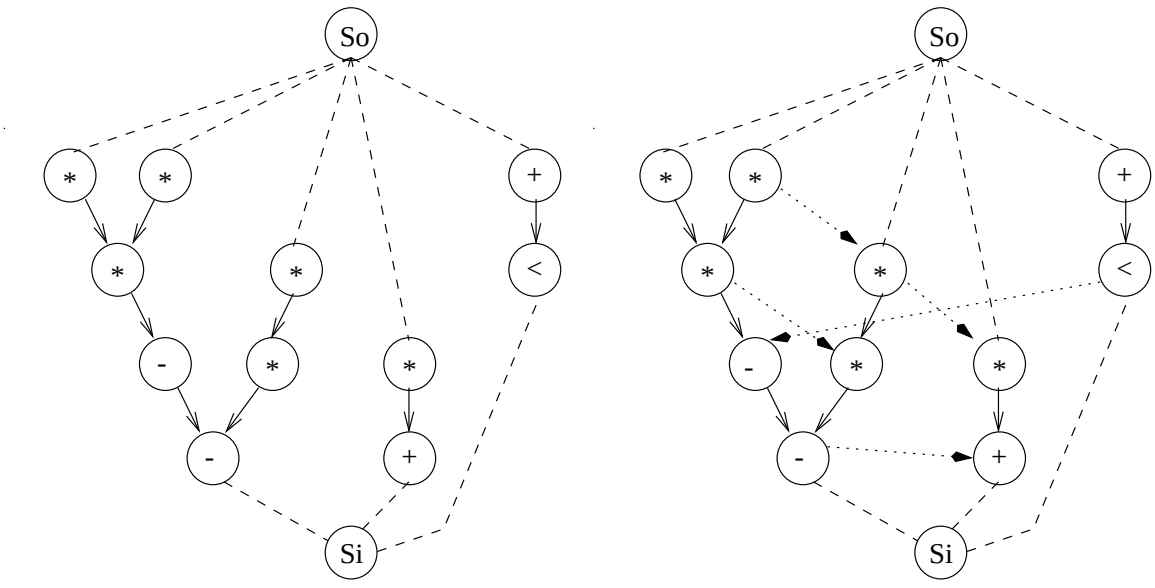


Figure 3.6. Data flow graphs without and with resource edges.

infeasible as the granularity of time is increased. Figure 3.7 illustrates the crossover in the usefulness of the two methods. Being able to discretize time without a loss in performance is important for asynchronous design because of the naturally continuous nature of events which can occur in an asynchronous circuit.

3.4.1 Left-Edge Algorithm

Using the left-edge algorithm from [26], it is possible to do resource sharing. The algorithm first sorts the operations or nodes by their scheduled start time, or *left-edge*. It considers one instance of a resource at a time and assigns as many operations as possible to that instance by searching the nodes sorted in ascending order. Each iteration of the algorithm considers a new instance of the resource, until all operations are allocated to a specific resource instance.

The algorithm in [26] is used to perform asynchronous resource sharing, but with two modifications. First, the left-edge of each operation is determined by its scheduled start time in place of a specific clock cycle. The right-edge of each operation is determined by its scheduled stop time. This reflects the window of time in which the resources should not be shared. In the tool `Mercury`, the user can specify a more liberal window of time

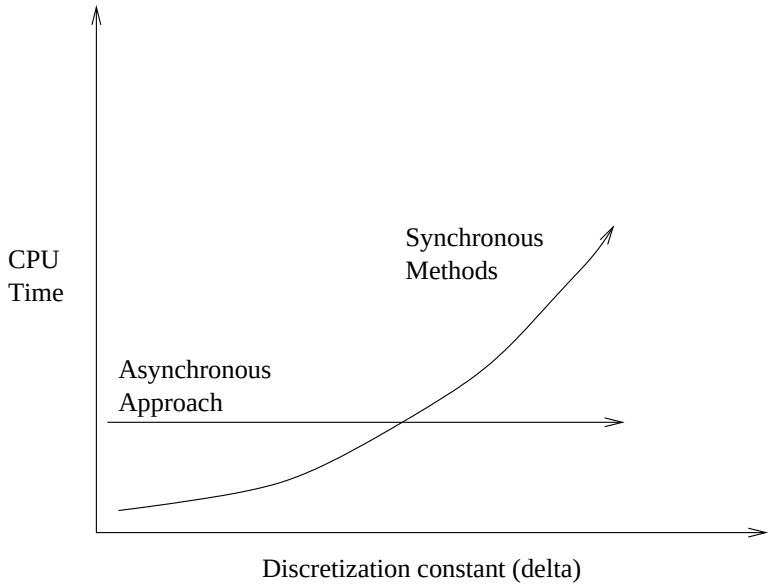


Figure 3.7. Synchronous methods vs. asynchronous approach.

if desired. For example the user could specify the typical start time to typical stop time. Second, the existence of a path between two operations is tested. When a path exists between two operations, it does not matter if the operations are scheduled at potentially conflicting times, the two operations are considered compatible because the existence of a path guarantees the operations are serialized with respect to each other.

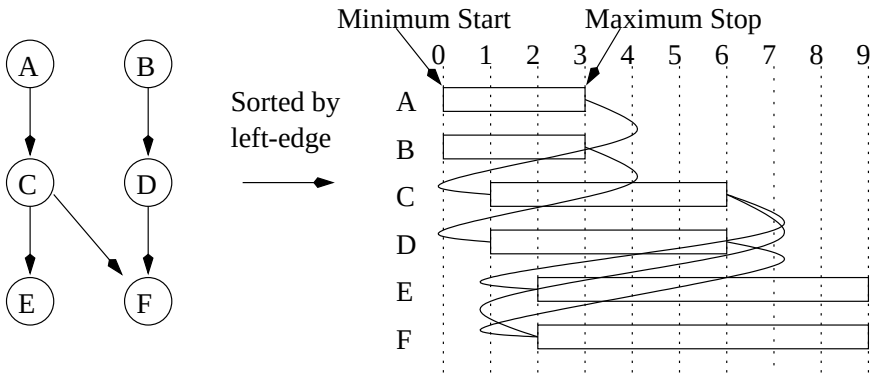
The asynchronous version of the left-edge algorithm is shown in Figure 3.8. The complexity of the algorithm is $O(VlogV)$. While the algorithm is not exact, it is found, in practice to give very good results efficiently.

Solving for an optimal configuration of resource sharing exactly is an exponentially difficult problem. Figure 3.9 illustrates the asynchronous left-edge algorithm on an example and shows why an exact solution is difficult. The example shows that the ordering of operations can, in some cases, determine the quality of the solution. In the example, each node in the data flow graph is the same operation, and the resource bound to each operation has a minimum computational delay of one and a maximum computational delay of three. The schedule for each node gives the interval in which the operation can use the resource. The nodes are sorted according to their start times.

Initially, the algorithm would pick A or B and start scheduling operations, since they both have the same left-edge. Assuming that operation A is picked first, the algorithm will share it with operation C, even though there is an overlapping window of time, because

```
Asynchronous-Left-Edge (I) {
    Sort elements of I in a list L in ascending order of l_start.
    instance = 1;
    foreach operation in l in L {
        l_instance = instance;
        t = l;
        foreach operation k in L starting at l + 1 {
            if k_min_start >= t_max_stop or path between t and k {
                k_instance = instance;
                t = k
                remove k from L;
            }
        }
        instance++;
        remove l from L;
    }
}
```

Figure 3.8. Asynchronous left-edge algorithm.



All operations are the same with minimum delay of 1, and maximum delay of 3.

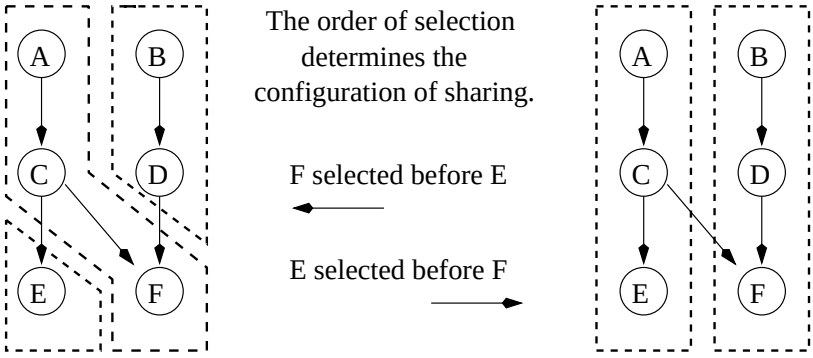


Figure 3.9. Example using the asynchronous left-edge algorithm.

a path exists between A and C, and C has the next lowest left edge. The next step is to pick either E or F to share with A and C. From the example, note that operations E and F both have the same potential start time. If E is evaluated before F as a potential node to share with A and C, then an optimal solution is found. If F is evaluated before E, then the solution is not optimal. To find an exact solution to resource sharing, the order of picking nodes has to be considered.

Because of the complexity of an exact solution, we do not consider the ordering of nodes. But for consistency, the algorithm is stabilized by sorting each node first, by its left-edge, and second, by the name of the node in the graph. This guarantees that the algorithm always gives the same answer for the same scheduled graph even though it may not always be an optimal solution.

3.4.2 Clique Covering

Another method of resource sharing is clique covering [23]. A clique is defined as a maximally connected set of operations in a graph. Clique covering is the process of finding a minimum number of maximal cliques. It attempts to solve the problem by creating a resource compatibility graph. The compatibility graph has edges between all nodes that are compatible. Similar to the left-edge technique, two operations are compatible if a path exists between them, or if they have disjoint time frames.

A solution to the clique covering problem can be determined by iteratively searching for the maximum clique in the graph and then deleting it from the graph until there are no more nodes in the graph. Each clique is assigned an instance of the resource. The clique covering problem is intractable, and so, heuristic techniques have been developed.

One common heuristic to find the maximum clique in a graph first calculates the degree of each node. The degree of a node is determined by the number of adjacent nodes. Next, the node with the highest degree is selected. Then, adjacent nodes are iteratively selected in a similar manner. If the node under inspection is adjacent to all previously selected nodes then it is selected to be part of the clique. It is hoped that by picking the node with the highest degree at each iteration the largest possible clique will be created.

The complexity of the clique covering algorithm is $O(V^2)$. This makes the algorithm much more complex than the left-edge algorithm. The algorithm is also slower because it requires dynamic creation and manipulation of the compatibility graph. In nearly all tests, clique covering gave the same or worse results than the left-edge algorithm. Figure 3.10 shows the compatibility graph for the example in Figure 3.9. The figure shows why

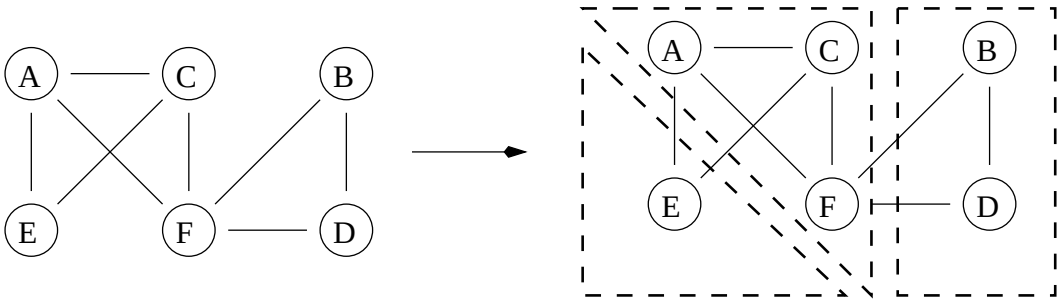


Figure 3.10. Compatibility graph for clique covering.

the left-edge algorithm can perform, in many circumstances, better than clique covering. Using the heuristic, operation F would be selected as the starting node since it is the highest degree node in the compatibility graph. Next, either operations A or C would be selected to be part of the clique, since they are the next highest degree nodes. By limiting the choices to A and C, the optimal solution which requires F to be paired with B or D does not occur.

The example shows that in this case the clique covering technique would not yield the best solution, whereas, the left-edge algorithm still has the potential to find the better solution. The clique covering heuristic could be modified to randomly select nodes instead of selecting those with the highest degree first. If this were the case, it would have the potential of finding the better solution, but the algorithm still requires much more computation time, making it an undesirable choice. For essentially the same results, the left-edge algorithm is a more efficient method of determining resource sharing.

Resource sharing and allocation determine, in part, the area a particular design requires. The total area can be computed by summing the area of each resource instance. In addition, where resource sharing has occurred, the area of each multiplexor is added to the total. The area each latch requires is neglected because the number of latches for each configuration in the design space is currently constant. The area of the control is not added to the total since `Mercury` only generates a behavioral model of the control. After refining the control to the gate level using `ATACS`, area estimations could be extracted and taken into consideration. For now, the supposition that the circuits will be resource-dominated is made, and it is assumed the area of the control is negligible.

CHAPTER 4

THE DESIGN SPACE

God is in the details

—Mies van der Rohe

The design space of this problem is all possible configurations of the data flow graph that can be used to create a datapath. Design space exploration starts with the user-provided data flow graph and incrementally adds resource edges to the graph. Each added edge serializes more operations. Each serialized operation reduces the area of the system because better resource sharing may occur. But this in turn may increase the latency of the system. When exploring the design space, three different types of edges are used. These include *data edges*, *resource edges*, and *implied edges*.

Data edges are edges that show the relationship between computed values and their future use. These edges are initially provided by the user via the specification. Resource edges are added between operations during exploration to allow the operations to share the same physical resource, they also imply the ordering precedence of operations. An implied edge is an edge which can be inferred between two compatible resources based on conservative timing analysis. A data dependency edge between two operations implies a resource edge between the operations if they are compatible operations, but a resource edge between two operations does not imply a data dependency edge.

To illustrate the design space, an example consisting of three compatible operations A, B, and C, with no data dependencies between their operations is used. Compatible operations are operations which can share the same resource. The design space for this configuration is shown in Figure 4.1. Edges between operations in the figure denote resource edges. In this case, the three concurrent operations yield a design space of 27 configurations. Infeasible configurations are denoted with an `I` and redundant configurations with an `R`. These are discussed later in more detail.

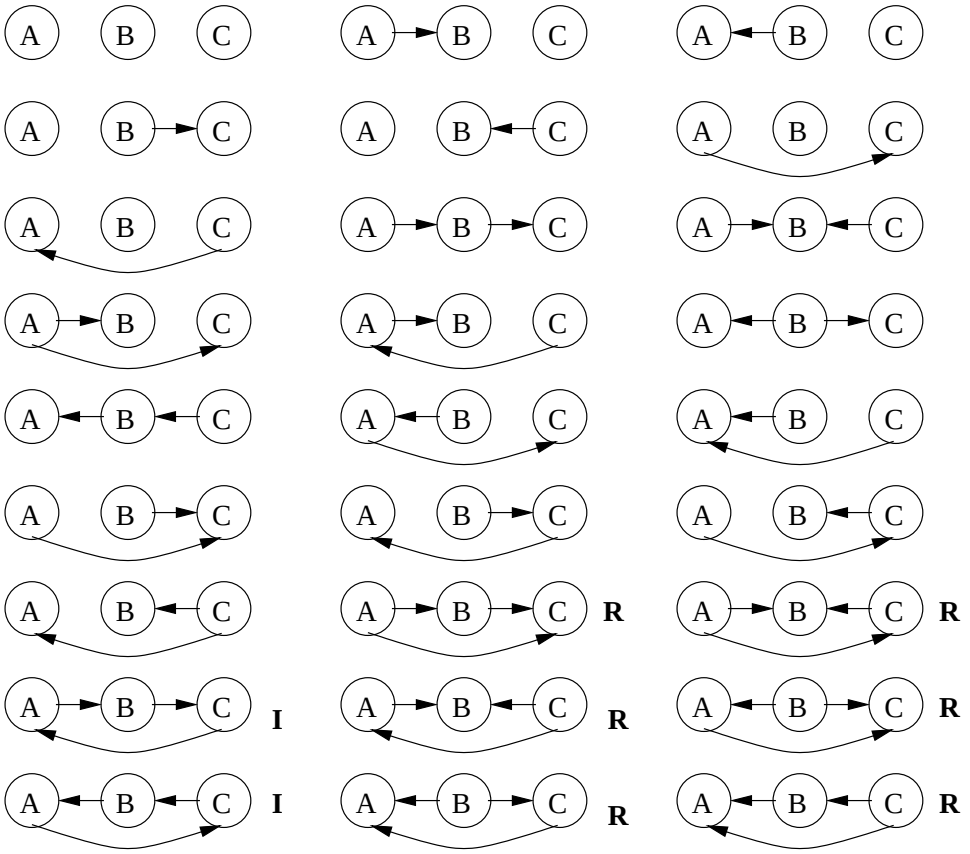


Figure 4.1. Exploration space of 3 compatible operations.

4.1 Reducing the Design Space

Many graphing problems are intractable because the design space grows exponentially in relationship to the number of nodes in the graph. The exponential explosion of the design space makes it very difficult to evaluate all possibilities in a finite amount of time. Evaluating each possible configuration of a data flow graph to find the best asynchronous datapath configuration has the exponential characteristic common to other graphing problems. Therefore, it is advantageous to eliminate as many configurations in the design space as possible before they are analyzed. Several optimizations to reduce the design space are used. This chapter first gives a brief overview of exploration and then discusses each of the optimizations.

The complexity of the design space for a configuration in which all operations are concurrent, compatible, and not dependent on one another grows at a rate of $O(3^{n(n-1)/2})$,

where n is the number of nodes in the graph. This configuration has the worst possible complexity, since graphs with data dependencies or graphs having noncompatible resources constrain the system and reduce the number of edges which can be added to serialize operations.

The complexity of the design space is derived as follows: between two independent nodes, a choice of three possibilities is made during exploration. An edge can be added from i to j, or an edge can be added from j to i, or no edge is added at all. Between n nodes, there are $n(n-1)$ possible directed edges. Picking both edges between any two nodes would create a cycle in the graph, so it is only possible for half of the edges to be added to the graph at any given time. So the complexity of the design space is $O(3^{n(n-1)/2})$.

4.2 Filters

Filters are special checks done during the exploration of the design space. Several filters are integrated into `Mercury` that eliminate much of the required design exploration. These filters include detecting and eliminating redundant designs, eliminating infeasible configurations, detecting maximum resource sharing, and considering operations which have implied serialization. An efficient pruning technique is also used when solving for a minimal-latency solution. While none of these filters exponentially decrease the design space for all configurations, each significantly reduces the design space and required runtime to find a solution.

4.2.1 Infeasible Edges

In exploring the design space, all possible orderings of adding resource edges to the original data flow graph are potentially considered. Since edges are directional, each direction of an edge between two nodes is explored. This means that with the addition of certain edges, the original acyclic data flow graph could become cyclic. Edges which create a cyclic graph are infeasible. The first filter eliminates designs which are created by an infeasible resource edge. More formally, if there is a path from the target of a candidate edge to the source of the candidate edge, then the candidate edge would create an infeasible design. The existence of a path from the target to the source node can be determined by finding the transitive closure of the data flow graph and using it to determine the existence of a path between the two nodes. If an edge is infeasible, then the design is not considered and design space exploration is pruned at that point.

Figure 4.2 illustrates a candidate edge from node C to A which would create an infeasible design, since it would create a cyclic dependency between operations. For the example in Figure 4.1, those designs which are created by infeasible edges are denoted with an `I`.

4.2.2 Redundancy

The second filter eliminates redundant designs. A design is redundant if the addition of an edge creates a design equivalent to one previously explored. To detect whether a candidate resource edge creates a redundancy, the algorithm shown in Figure 4.3 is used. The algorithm detects if a candidate edge is redundant by checking for the existence of a path from the source of the candidate edge to the target of the candidate edge. If there exists a path, then the candidate edge would create a redundant design. Next, the algorithm goes to the candidate edges' target and checks all of its resource edge predecessors. If there is a path from any one of those predecessors source operation back to the original candidate source operation, through a resource edge, then the newly added edge creates a redundant design. In the final step, the algorithm goes to the candidate edges' source and checks all of its resource edge successors. If there is a path from the original candidate edge target to any one of the successors targets, through a resource edge, then the design is also redundant.

The design space illustrated in Figure 4.1 has several redundant designs which are marked with an `R`. Each of these designs is eliminated from consideration with the redundancy filter. When a redundant design is detected, the design space is pruned and no further configurations along that branch of the exploration are considered. The

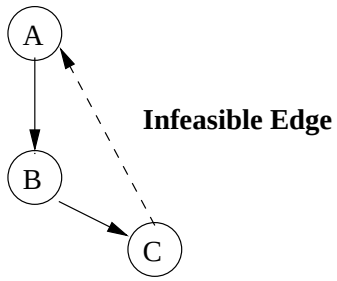


Figure 4.2. Infeasible edge.

```
foreach new edge e = (x,y)
  if (∃ path y → x) then
    infeasible
  if (∃ path x → y) then
    redundant
  foreach e' = (x',y) ∈ E_resource
    if (∃ path x' → x) then
      redundant
  foreach e' = (x,y') ∈ E_resource
    if (∃ path y → y') then
      redundant
```

Figure 4.3. Procedure to determine if adding a resource edge creates a valid design.

design space can be pruned because all future combinations of edges originating from a redundant one, are identical configurations to ones explored previously.

Many redundant designs are detected during exploration. It has been found that pruning the design space using this algorithm yields a significant reduction in exploration and runtime without sacrificing the quality of the solutions.

To further illustrate this filter, a simplified version of our original example is created by adding a data dependency edge from node A to B. Figure 4.4 shows the reduced design space. In this example, design 6 is redundant because of design 4. Note that in this

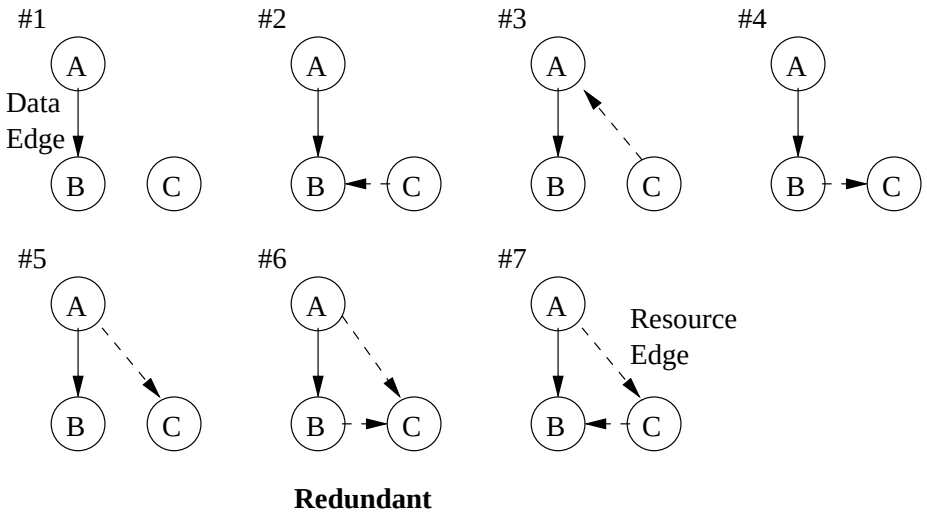


Figure 4.4. Design space showing redundancy.

case all operations are serialized in the same manner for both cases. Unlike the previous example, design 7 was not eliminated as being redundant because the edge from A to B is a data edge and not a resource edge.

4.2.3 Implied Edges

As mentioned earlier, implied edges are edges which are not in the original data flow graph, but can be added to the graph without affecting the scheduling of operations. Implied edges are important because they may affect the sharing of resources. An edge is implied between two operations if first, they have the same type of operation, and second, according to timing analysis it is determined that the two operations can never be in conflict with each other. To do a conservative resource analysis, the *critical window* of the resource is calculated using ASAP scheduling. If any two resources have overlapping critical windows then there cannot be an implied edge between those operations. Figure 4.5 illustrates an implied edge. Implied edges are always used to imply sharable operations when doing resource sharing, so if a candidate edge is an implied edge, then adding the candidate edge does not yield additional information and consequently the candidate edge does not need to be explicitly added to the graph. When an implied edge is detected, explicitly considering the edge is not required so the design space can be pruned.

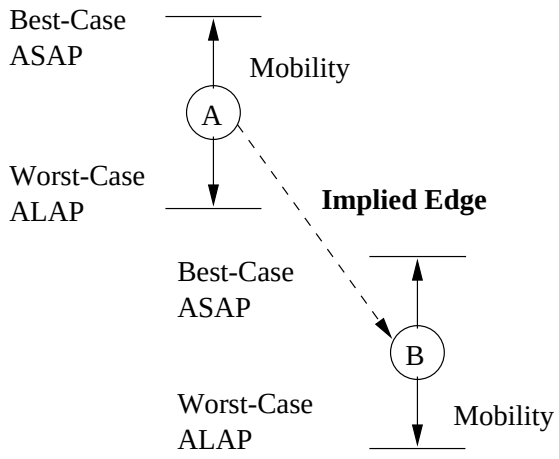


Figure 4.5. Implied edge.

4.2.4 Minimal Latency

When performance is the key optimization goal it is often the case that the designer would like the best possible performance for a design using minimal area. This is known as the *latency-constrained minimum-area* problem. When a designer seeks to find only minimal latency solutions, though, an additional optimization can significantly reduce the design space of the exploration.

For this optimization, it must be assumed that latency monotonically increases as each candidate edge is added to a design. It is believed that this is a fair assumption, because each additional edge either leaves the design unchanged or further serializes operations. Serializing an operation and employing resource sharing potentially add delay to the system, but does not decrease the delay. This is because larger muxes are required to feed multiple operands to the resource and the computation may potentially be delayed due to a resource conflict. From this, the design space is pruned when a candidate edge is found, which increases the overall latency of the system, because future designs originating from that configuration have equal or longer latencies. The overall system latency is calculated using unconstrained ASAP scheduling. When solving for a minimal-latency solution, if the overall system latency is greater than the value determined by typical ASAP scheduling, then the design space can be pruned.

This filter can be optimized further, and additional savings can be made by comparing the ASAP and ALAP bounds of source and target operations of a candidate edge. If the best-case start time of the source of a candidate resource edge is greater than the worst-case completion time of the target, then it can be concluded that there is no way to serialize the two operations without additional system delay. This is because the edge would force one of the operations out of its zone of mobility, which would in turn lengthen the critical path of the system. Figure 4.6 illustrates this comparison. The overall delay is increased because the edge forces the target and all of its successors to shift to later starting times, which in turn expands the overall delay of the schedule, forcing the design to have nonminimal latency. Using this technique is very efficient, because it does not require calculating the overall system delay with the added candidate resource edge. In order to prune the design space, this method only needs to examine the original schedule and determine if adding an edge between two given operations would lengthen the critical path.

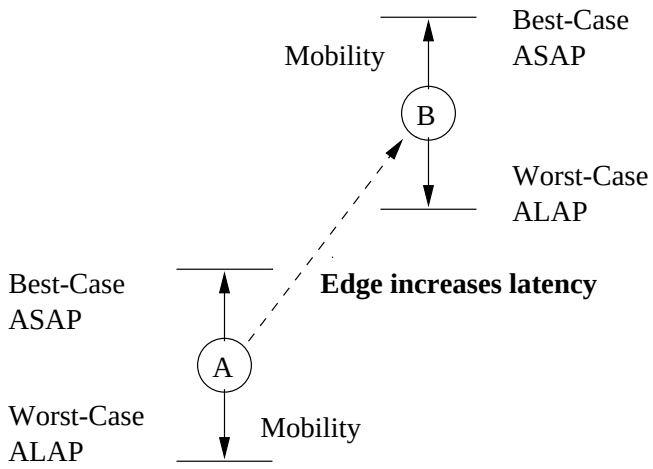


Figure 4.6. Minimal latency filter.

4.2.5 Constraints

Constraints can be specified by the designer to aide design space exploration. For example, the designer may know the total area that the design can occupy, or may specify a maximum limit on the number of allocated instances of a resource. Considering these constraints does not always significantly reduce the exploration of the system without opening the possibility of suboptimal solutions. Unlike delay, the system area does not monotonically increase or decrease as candidate edges are added. The area may increase if the cost of interconnect logic is greater than the savings given by resource sharing. This makes it difficult to prune the design space using a method similar to the minimal-latency optimization.

4.2.6 Maximally Shared Resources

Detecting when a resource is maximally shared is another optimization. Maximum sharing occurs when only one instance of the resource is required by allocation for a group of similar operations. Once a resource is maximally shared, further exploration of that resource is not productive. This is because the addition of more resource edges to the data flow graph, between compatible operations that are already assigned to the resource, does not change the allocation for the resource. For this reason, when a resource is maximally shared, no further exploration is performed for operations which are allocated to that resource. When all resources required by the data flow graph are maximally shared, all

operations of the system are maximally serialized, and this branch of the design space can be pruned from further exploration.

This optimization is exact because when all operations are serialized, further resource sharing is not possible. This means that the total area required by the design does not decrease, and the addition of resource edges either increases the latency of the system or leaves it unchanged. Since adding edges does not decrease the overall latency of the system, further exploration does not yield a configuration better than a configuration with maximum serialization, which uses a minimum number of resource edges.

4.2.7 No Change In Objectives

Another filter detects when, after the addition of a candidate resource edge, there is no change in the values of all objectives. If there is no change, then the design space is pruned and further exploration is not done with the edge included. In this case, it is assumed that a resource edge is useless when it has no effect, so the algorithm skips to edges that make a difference in the design. The assumption may not however be correct if, later in the exploration additional resource edges have been added to the graph and then the edge does make a difference in the sharing or sequencing of operations. When this occurs, the two paths of exploration are no longer parallel. If this occurs, then this filter may cause the quality of results to deteriorate since unique solutions will not be evaluated. This method is a very aggressive heuristic that significantly reduces the time required to find a solution; however there is a tradeoff in the quality of solutions.

4.3 Hierarchal Exploration

Another method to reduce the design space uses a hierarchal exploration approach. In this method, not all combinations of resource edges between operations are explored. The hierarchal approach groups each operation in the data flow graph according to their type. Then, exploration is done separately for each group. For example, exploration is done for all ALU operations separately from exploration for multiply operations.

Each group is explored by adding resource edges between operation pairs in the group. Edges between operations that are not in the group are not modified. Resource edges in a group that affect the overall area and delay in a favorable manner (*critical edges*) are stored. When the critical edges for each group are found, all possible combinations of the critical edges are added to the original data flow graph. Each new configuration is evaluated and a final set of solutions is discovered.

This approach detects edges that do not have an impact on scheduling or allocation and removes them from further consideration. Extracting groups of edges reduces the complexity of the design space. This happens because, in general, the sum of the complexity of each groups' design space is much smaller than the complexity of exploring the entire design space all at one time. Furthermore, if a group does not have any favorable edges, then that group, or set of operations, is dominated by other operations in the graph. This focuses exploration on groups of operations which have the potential to optimize the overall design further. Figure 4.7 illustrates the hierarchal grouping of resources for the differential equation solver.

The solutions produced using this method, however, may not be globally optimal: when critical edges are determined for each group, it is assumed that other operations are scheduled and allocated without constraint. This means that it is possible to skip critical edges that are dependent on other critical edges, which are not part of the current group being explored. For example, if edge A from group X is not a critical edge, independent of edges from other groups, it would not be considered. But, if critical edge B from group Y were added to the graph causing A to become a critical edge, then A should be considered. Using the hierarchal approach, edge A would be skipped.

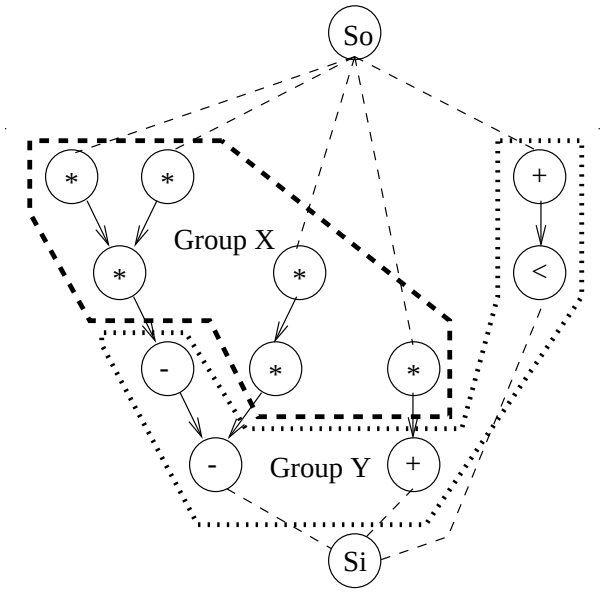


Figure 4.7. Grouping of resources for hiearchal exploration.

`Mercury` implements the hierarchal approach and we observe its effectiveness in the case studies of Chapter 6. It is has been found to be very beneficial when solving large designs where the design space is too large to efficiently solve using nonhierarchal methods.

CHAPTER 5

SYSTEM IMPLEMENTATION

To iterate is human, to recurse: divine!

—L.P. Deutsch

5.1 General Algorithm

The engine of the exploration system uses a branch-and-bound algorithm to explore the design space. The design space is searched for the best possible set of schedules and allocations by incrementally adding resource edges to the data flow graph. Each resource edge added can affect the performance, area, or other attributes of the system. Thus, after each edge is added, the newly created graph is analyzed for performance in terms of area and latency.

Trade-offs between area and latency are managed by using *Pareto points* [10]. Any point in the design space which is superior to all other points in one objective, or a combination of objectives, is a Pareto point. Each design space may have many Pareto points that correspond to unique design configurations not dominated by others. Therefore, each Pareto point is worth consideration as a candidate configuration for implementation. Figure 5.1 illustrates the concept of a Pareto point for two objectives: delay and area. The concept could can be extended into the third demension using another objective such as power.

`Mercury` evaluates each configuration in the design space according to two objectives: delay and area. These objectives are used to find Pareto points. If the new design is a Pareto point, then that configuration is stored in a set of Pareto point solutions. Solutions which are added to the set may be better than former solutions in the set, so any former solutions which are no longer Pareto points are removed from the list.

The branch-and-bound algorithm for this problem is illustrated in Figure 5.2. The algorithm begins by selecting two operations A and B from the graph and determining if adding a candidate resource edge between the two operations satisfies all of the bounding

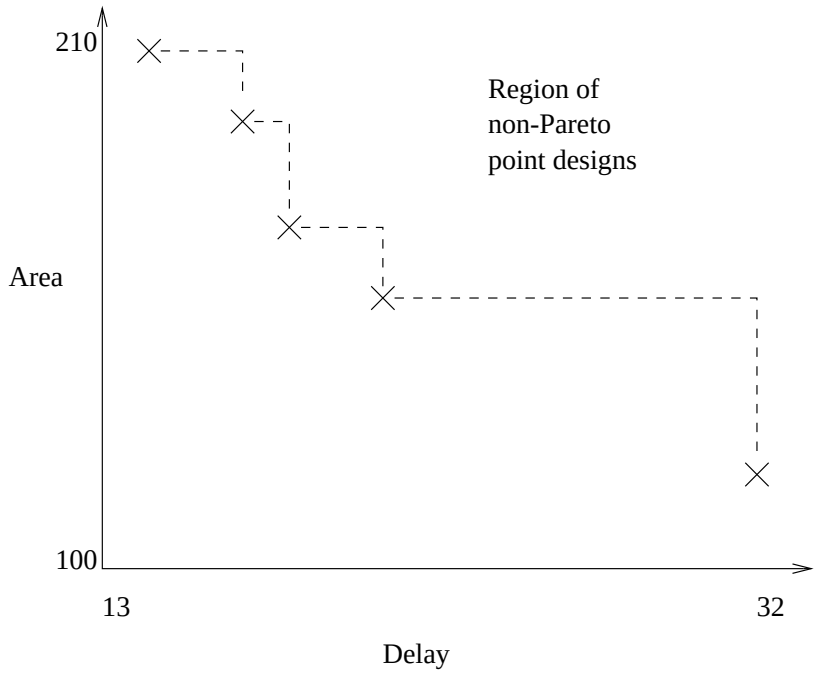


Figure 5.1. Pareto Points.

conditions. This includes not being filtered by any active filters from Chapter 4. For example, if the filter for checking redundant configurations is active, the algorithm skips over any edges that create a redundant graph. Or, if only a minimal-latency solution is desired, then the edge must satisfy the minimal-latency filter, or it is dropped.

Each time a candidate edge is filtered, or the algorithm exceeds constraints, the design space is pruned. If the candidate resource edge satisfies all of the bounding conditions, then the algorithm recurses into another level of the exploration. The next level considers all remaining edges with and without the candidate resource edge. Recursion continues until all possible edges between any two compatible operations have been explored or pruned. Once the algorithm completes, the Pareto points remaining in the solution set are the best solutions.

Using this algorithm on a data flow graph with three concurrent nodes, the design space shown in Figure 4.1 would be found from the exploration tree shown in Figure 5.3. In Figure 5.3, each branch indicates a level of recursion from the algorithm and each bold node represents one of the configurations in the design space. For this example, infeasible

```
explore(operation a, operation b, binding bi) {
    pareto_points pp;

    if (b == end_operation) {
        b = first_operation;
        a = next_operation;
    }

    if (a == end_operation)
        return pp;

    while (any active filter is not satisfied) {
        b = next operation;
        if (b == end_operation) {
            b = first_operation;
            a = next_operation;
            if (a == end_operation)
                return pp;
        }
    }

    /* Recurse without adding resource edge */
    pp + = explore(a,b+1,bi);
    add-resource-edge(a→b,bi);

    /* Calculate the area and latency of the configuration */
    pareto p = evaluate_design(binding );
    if (p == Pareto_point)
        pp + = design(bi,p);

    /* Recurse with the resource edge added */
    pp + = explore(a,b+1,bi);
    remove-resource-edge(a→b,bi);

    return pp;
}
```

Figure 5.2. Exploring the design space using a branch-and-bound search.

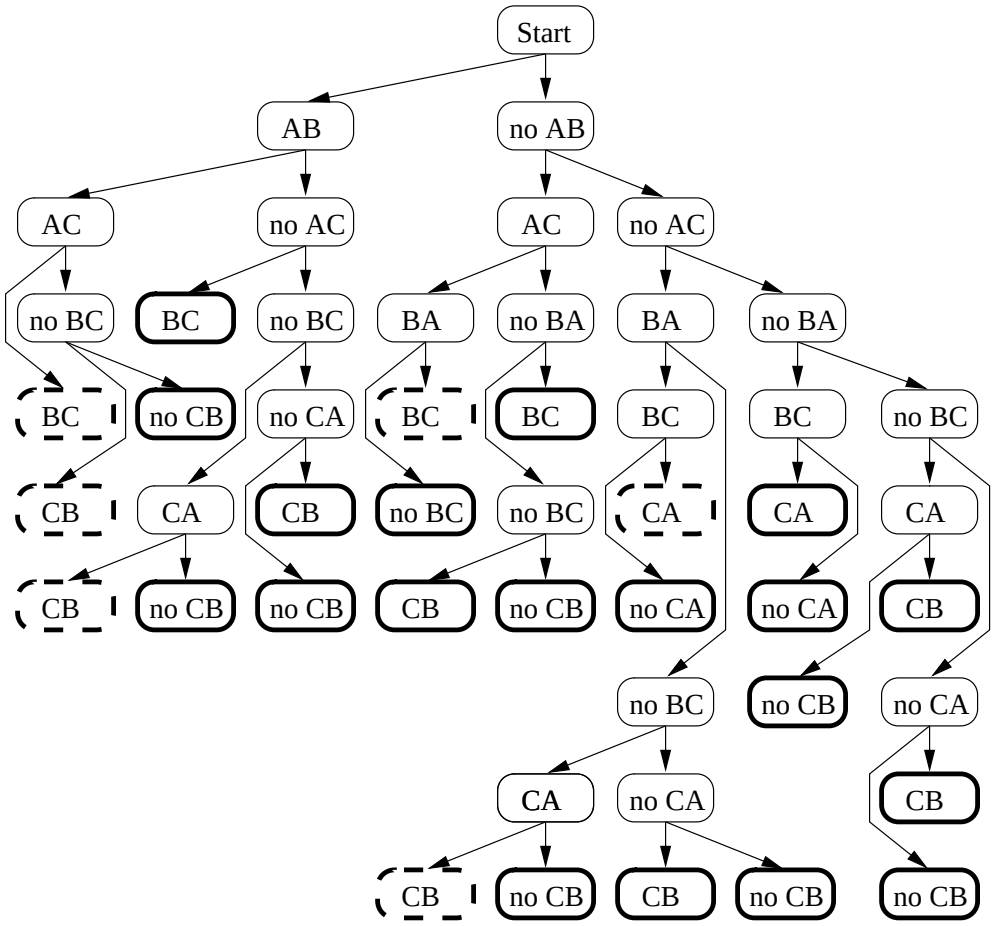


Figure 5.3. Exploration tree.

designs are filtered from exploration and are not shown. Six redundant designs are shown with dashed boxes. They correspond to the redundant designs in Figure 4.1.

Note that exploration does not grow symmetrically, but rather grows to the right. This is because each added edge constrains the system further. Edges which are not added to the system do not constrain the system, so more options are available latter in exploration. Twenty-five designs that are evaluated during exploration.

5.2 Optimizations

At each step in the branch-and-bound tree, the graph is evaluated with only one change from the prior step. This change is the addition or removal of a single edge from the graph. Each time an edge is added or removed from the graph, a topological sort

must be done on the graph, and the ASAP and ALAP schedules must be updated. In addition, the transitive closure of the system, which determines whether a path exists between any two operations, must be updated.

To gain as much efficiency as possible for incremental changes, two optimizations are employed. The first is dynamic transitive closure on the graph; the second is dynamic analysis of the ASAP and ALAP schedules.

5.2.1 Dynamic Transitive Closure

The transitive closure of a graph supports reachability queries. For example: is there a path between vertices i and j in the graph? Once a transitive closure has been calculated for a graph, boolean-path queries can be answered in constant time. This makes the classical transitive closure method good for static graphs which are created once and queried many times. If, however, as in the method presented here, there are many updates to the graph, followed by only a few queries, the classical approach is computationally expensive because it would compute the answer to all queries from scratch after each update.

Using a dynamic transitive closure algorithm, it is possible to update the reachability of vertices more efficiently. A dynamic transitive closure algorithm developed by Cicerone [39] which is a generalization of another algorithm proposed by La Poutré and van Leeuwen [37] is used. Other similar algorithms include Italiano's [28, 29] algorithm and Yellin's algorithm [44].

The algorithm proposed by Cicerone uses a counting technique to solve the problem. Information on edges existing in the graph is maintained explicitly in an adjacency matrix.

As resource edges are added or deleted, the adjacency matrix is updated to reflect the changes in structure of the graph. If the transitive closure matrix contains a count of zero, then no path exists between the pair of operations. If the count is greater than zero, then a path exists and the count represents the number of adjacent edges creating a path through the pair of operations. When an edge is inserted into the graph, the transitive closure is updated. Figure 5.4 shows the algorithm for updating the adjacency matrix when an edge is added to the graph.

The algorithm works as follows. First, the new edge is added to the graph G. Then, each operation, or node k with a path to the source i, is considered. A queue is initialized with the value of the target operation j. Then, while that queue is not empty, the count, which indicates a path from k to the current operation on the queue, h, is incremented.

If the count becomes equal to one, then each successor of h is added to the queue. The successors are added to the queue in this case because the new edge caused two previously unreachable nodes to become reachable. The transitive closure of each of the successors needs to reflect the existence of the new path. If the count is greater than one, then the successors are already aware that a path exists between the two operations from a previous edge insertion, and no further updating is required. Otherwise, the successors are pushed onto the queue. The algorithm continues until all successors of the newly added edge are updated with the new path, or are already aware of the path. The delete operation works in a similar, but opposite, manner and is shown in Figure 5.5.

Using the algorithms in Figure 5.4 and 5.5 when inserting and deleting edges from the graph, it has been proven in [39] that the total time required to insert q consecutive edges in a graph G, with n vertices and m edges is $O(n(q+m))$, and the total time required to delete q consecutive edges in graph G is also $O(n(q+m))$. Furthermore, it has been proven that it achieves $O(n)$ amortized time per operation. The adjacency matrix requires $O(n^2)$ storage space. Like other methods, queries can still be answered in constant time. This is an improvement over non-dynamic solutions that have a complexity of $O(n^2)$.

5.2.2 Dynamic Scheduling

After each insertion or removal of a candidate edge, the schedule of a data flow graph must be recomputed. Since only one edge has changed in the data flow graph, a method called dynamic scheduling updates the ASAP and ALAP schedules without recalculating the schedule for every operation in the graph.

For dynamic ASAP scheduling, each added edge between a source and target operation can only affect the schedule of the target operation and the target's successor operations. Therefore, when an edge is added, the schedule of the target operation is recomputed. If its schedule is changed, then the algorithm recurses to each of its successor operations and recomputes their schedules. The algorithm continues recursing through any successors whose schedule has changed until there are no more successors, or there is no change in any successor's schedule. When removing an edge, the same algorithm can be used, since again, only the target operation, and its successors are affected by the removal of the edge. Figure 5.6 illustrates the ASAP-update algorithm.

For dynamic ALAP scheduling, the concept is the same, but the operation is more difficult. In this case, if the overall delay of the system changes, all operation's schedules also change, so the ALAP schedule must be entirely recomputed. To determine if the

```
insert(i,j) {
    G = G ∪ {(i,j)}
    foreach k ∈ V
        if (C[k,i] ≥ 1)
            set-queue(Q_k,(i,j))
            while Q_k is not empty
                pop-queue(Q_k,(l,h))
                if (h ≠ k)
                    C[k,h] = C[k,h] +1
                    if (C[k,h] == 1)
                        foreach (h,y) ∈ out[h]
                            push-queue(Q_k,(h,y))
}
```

Figure 5.4. Updating dynamic transitive closure for insertion of an edge.

```
delete(i,j) {
    G = G − {(i,j)}
    foreach k ∈ V
        if (C[k,i] ≥ 1)
            set-queue(Q_k,(i,j))
            while Q_k is not empty
                pop-queue(Q_k,(l,h))
                if (h ≠ k)
                    C[k,h] = C[k,h] −1
                    if (C[k,h] == 0)
                        foreach (h,y) ∈ out[h]
                            push-queue(Q_k,(h,y))
}
```

Figure 5.5. Updating dynamic transitive closure for deletion of an edge.

overall delay of the system has changed, the delay of the new ASAP schedule can be compared with the delay of the prior ALAP schedule. If there is no change in the overall delay of the system, then the algorithm recomputes only those operations which are affected using a method similar to dynamic ASAP scheduling. Figure 5.7 illustrates the ALAP-update algorithm.

```
ASAP-update(target) {
    compute new-schedule for target;
    if (target's prior-schedule != target's new-schedule)
        prior-schedule = new-schedule;
        foreach (successor of target i)
            ASAP-addedge(i);
}
```

Figure 5.6. Updating ASAP schedule for insertion or deletion of an edge.

```
ALAP-update(target) {
    if (new overall delay == old overall delay)
        recompute(target);
    else
        schedule-ALAP();        }

recompute(target) {
    compute new-schedule for target;
    if (targets prior-schedule != targets new-schedule)
        prior-schedule = new-schedule;
        foreach (successor of target i)
            recompute(i);
}
```

Figure 5.7. Updating ALAP schedule for insertion or deletion of an edge.

CHAPTER 6

CASE STUDIES

Be careful of going in search of Adventure.
It is ridiculously easy to find.

—William Least Heat Moon

To test the effectiveness of the filters, three common high-level synthesis benchmarks are used: a differential equation solver (DIFFEQ), a fifth order elliptical wave filter (EWF), and an inverse discrete cosine transform (IDCT). The differential equation solver is the smallest of the three examples with a total of 11 operations. The elliptical wave filter is larger yet with 32 operations. The inverse discrete cosine transform is the largest, with 46 operations.

All of the case studies were performed using a Pentium II 400 Mhz processor with a 512 kilobyte level 2 cache and 384 megabytes of synchronous DRAM. The operating system used is RedHat Linux version 5.0, and the source code for `Mercury` was compiled using GNU C++ version 2.8.1. Memory was not an issue for any of the tests. Maximum memory utilization during exploration was approximately 13 megabytes. Throughout each test, CPU utilization was at, or near capacity.

6.1 Differential Equation Solver

Using the data flow graph for the differential equation solver shown in Figure 2.1, exploration is done using both the hierarchal and nonhierarchal approaches. By default, the infeasible edge filter is always active for each of these tests, since exploring infeasible designs is not useful. ALU operations are modeled with a minimum delay of one, typical delay of two, and maximum delay of three. It is assumed that they require 21 units of area. Multiply operations have a minimum delay of four, a typical delay of five, and a maximum delay of six. It is assumed that they require 43 units of area. Multiplexors are modeled with a base area of three units, corresponding to a 2x1 multiplexor. For an $(Nx1)$ multiplexor the area is modeled as $base * (N - 1)$.

A selection of designs from the solution set were tested using ViewLogic's VHDL simulator FusionSpeedwave. A simple validation procedure was done in which each of the primary inputs were assigned values, then the global request of the system was asserted. When an acknowledge was received, the values of the primary outputs were checked. All designs were checked to have correct functionality.

The results of exploration using hierarchal and nonhierarchal methods are shown in Table 6.1 and Table 6.2 respectively. The table shows which filters are active for each test, the amount of CPU time required to run the test to completion, the total number of configurations explored, and the number of solutions in the final Pareto point set. Performing a complete exploration of the design space, in the worst case, required the evaluation of over 22 million configurations and took several hours to complete. With the use of the filters, the design space is pruned and runtime is reduced.

The hierarchal approach reduced the design space even further. This heuristic broke the graph up into two sets: ALU operations and multiplication operations. Using this approach, fewer solutions are found. However, the quality of the solutions is nearly as good.

For example, comparing the results of the nonhierarchal approach using none of the filters, with the hierarchal approach, also using none of the filters, it is found that the first method yielded 292 solutions, while the second method yielded only 82 solutions. Of the 292 solutions, there are five unique Pareto points. Of the 82 solutions from the hierarchal approach there are also five unique Pareto points. In other words, all solutions landed on one of the five Pareto points, but did so with differing configurations of resource edges. The unique Pareto points are shown in Table 6.3. The tradeoff in the two methods between the quality and quantity of solutions and time required to find a solution seems reasonable. The comparison is also done for the case when all filters, excluding the minimal-latency filter, are used. Again, both methods yield five unique Pareto points. The results are shown in Table 6.3.

A structural view of one solution with minimum latency is shown in Figure 6.1. A solution with minimum area is shown in Figure 6.2. In the figures, note that the control block area appears to dominate the chip. This however, is actually not the case. It only appears larger in the figure to facilitate showing the individual control wires.

Table 6.1. DIFFEQ: experimental results using nonhierarchal approach.

Filters						
Implied	Redundant	Shared	No Change	CPU Time	Size	Solutions
				8318.58s	22167679	292
X				8132.70s	21714011	292
		X		7326.50s	19614054	292
X		X		7313.12s	19214280	292
	X			558.68s	1503207	81
X	X			539.96s	1489156	81
	X	X		515.15s	1436817	81
X	X	X		513.47s	1423064	81
			X	10.72s	32280	34
X			X	10.77s	32250	34
		X	X	9.96s	30059	34
X		X	X	10.13s	30029	34
	X		X	3.53s	10801	16
X	X		X	3.69s	10788	16
	X	X	X	3.49s	9972	16
X	X	X	X	3.48s	9959	16
Solving for minimal-latency solutions only:						
				329.34s	865444	31
X				299.10s	797678	31
		X		281.54s	758056	31
X		X		254.10s	700662	31
	X			34.98s	95134	14
X	X			33.69s	91524	14
	X	X		32.81s	93894	14
X	X	X		31.76s	90284	14
			X	.81s	1984	14
X			X	.82s	1976	14
		X	X	.76s	1964	14
X		X	X	.79s	1956	14
	X		X	.42s	968	12
X	X		X	.45s	963	12
	X	X	X	.42s	950	12
X	X	X	X	.50s	945	12

Table 6.2. DIFFEQ: experimental results using hierarchal approach.

Filters						
Implied	Redundant	Shared	No Change	CPU Time	Size	Solutions
				72.18s	162015	82
X				64.78s	159913	82
		X		65.75s	161935	82
X		X		65.06s	159833	82
	X			8.43s	20909	26
X	X			8.44s	20741	26
	X	X		8.71s	20905	26
X	X	X		9.71s	20737	26
			X	1.45s	2741	24
X			X	1.31s	2737	24
		X	X	1.23s	2653	24
X		X	X	1.30s	2649	24
	X		X	.48s	887	12
X	X		X	.45s	885	12
	X	X	X	.52s	878	12
X	X	X	X	.51s	876	12
Solving for minimal-latency solutions only:						
				.47s	1039	3
X				.45s	1038	3
		X		.28s	579	3
X		X		.29s	579	3
	X			.30s	578	3
X	X			.25s	578	3
	X	X		..07	53	3
X	X	X		.08s	53	3
			X	.07s	52	3
X			X	.10s	52	3
		X	X	.08s	53	3
X		X	X	.09s	53	3
	X		X	.11s	52	3
X	X		X	.11s	52	3
	X	X	X	.13s	53	3
X	X	X	X	.07s	53	3

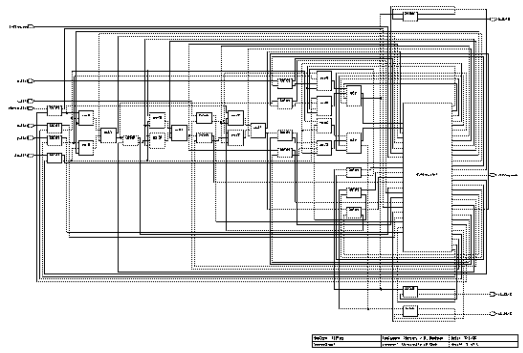


Figure 6.1. DIFFEQ: minimum latency solution

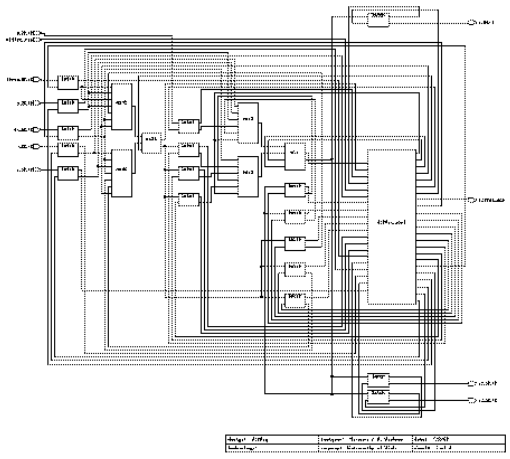


Figure 6.2. DIFFEQ: minimum area solution

Table 6.3. DIFFEQ: comparison of unique Pareto point solutions.

No Filters				With Filters			
Nonhierarchal		Hierarchal		Nonhierarchal		Hierarchal	
Area	Delay	Area	Delay	Area	Delay	Area	Delay
119	32	119	32	119	32	119	32
157	19	157	19	157	19	157	19
172	17	172	17	172	17	172	17
195	16	195	16	195	16	195	16
210	14	210	14	210	14	210	14

6.2 Elliptical Wave Filter

The second case study uses a fifth order digital elliptical wave filter. The functional dependencies of the filter are shown in Figure 6.3 after it has been transformed using common subexpression elimination and distributivity to reduce the number of multiplications and additions. Figure 6.4 shows the resulting data flow graph and Table 6.4 shows the experimental results using the hierarchal approach.

The same parameters for the functional unit were used as in the differential equation solver example. For these results, the hierarchal approach is used with a maximum block size of ten. This means that the algorithm randomly breaks each set of similar operations into blocks of ten. Exploration is then done only considering resource edges between operations in each block. Runtime grows rapidly as the block size is increased. After exploration is done on all sets, exploration is done again considering only critical resource edges which are included in the individual block Pareto point solutions. One datapath generated by `Mercury` is shown in Figure 6.5.

A comparison between the quality of solutions for the hierarchal approach using all of

$$
\begin{aligned}
o_1 &= i_1 \\
o_2 &= 126 * i_1 + 125 * i_2 + 112 * i_3 + 56 * (i_4 + i_7 + i_8) \\
o_3 &= 160 * (i_1 + i_2) + 152 * i_3 + 9 * i_5 + 80 * (i_4 + i_7 + i_8) \\
o_4 &= 7 * (i_1 + i_2 + i_3 + i_7 + i_8) + 6 * i_4 \\
o_5 &= 140 * (i_1 + i_2) + 133 * i_3 + 8 * i_5 + 70 * (i_4 + i_7 + i_8) \\
o_6 &= 144 * (i_1 + i_2 + i_3 + i_4) + 9 * i_6 + 232 * i_7 + 240 * i_8 \\
o_7 &= 162 * (i_1 + i_2 + i_3 + i_4) + 10 * i_6 + 261 * i_7 + 270 * i_8 \\
o_8 &= 150 * (i_1 + i_2 + i_3 + i_4) + 250 * i_7 + 269 * i_8 \\
o_9 &= 135 * (i_1 + i_2 + i_3 + i_4) + 225 * i_7 + 243 * i_8
\end{aligned}
$$

Figure 6.3. Functional notation for the elliptical wave filter.

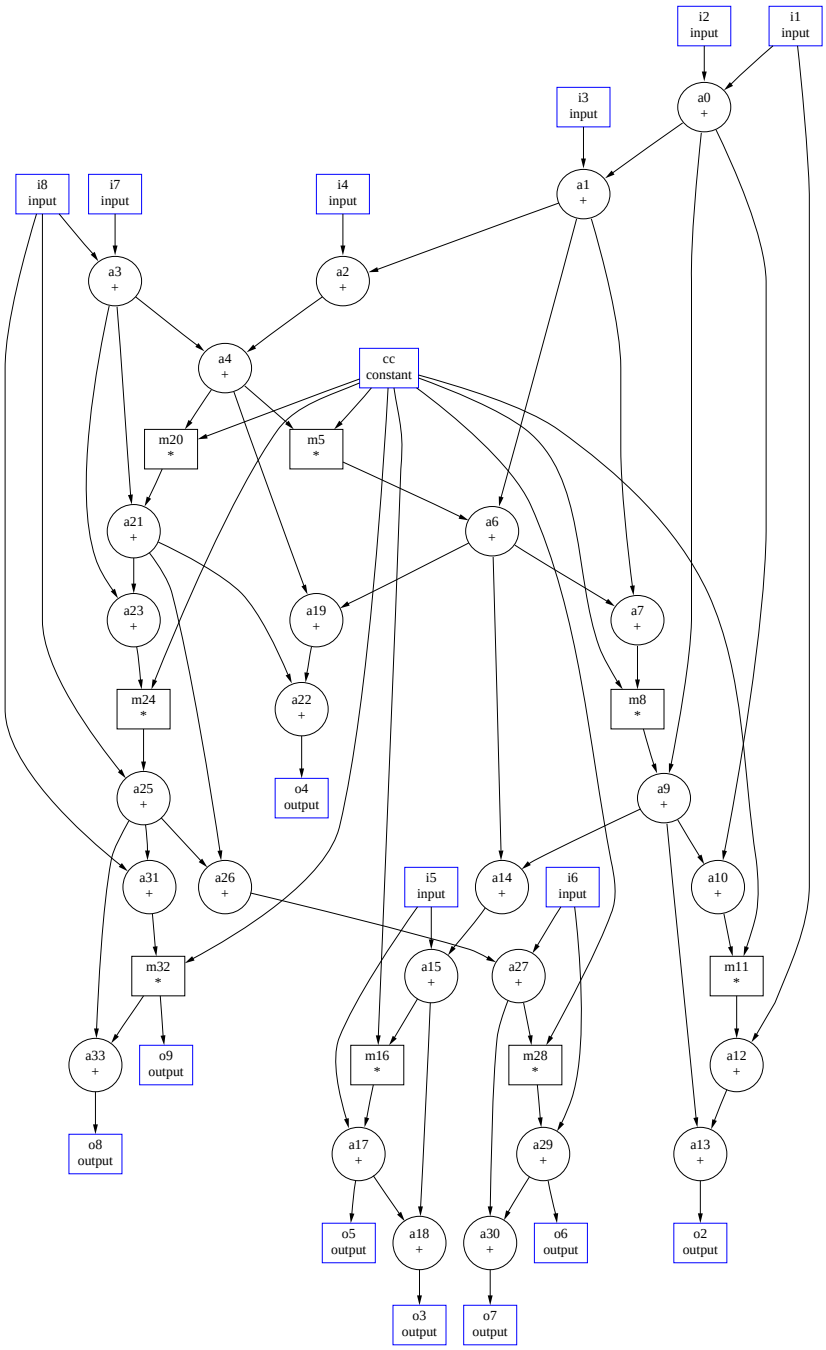


Figure 6.4. Elliptical wave filter data flow graph.

Table 6.4. EWF: experimental results using hierarchal approach.

Filters						
Implied	Redundant	Shared	No Change	CPU Time	Size	Solutions
				160814.64s	58194121	18
X				160534.51s	58171711	18
		X		160631.13s	58194121	18
X		X		160447.96s	58171711	18
	X			1361.72s	5444983	12
X	X			1380.11s	5444569	12
	X	X		1350.12s	5444983	12
X	X	X		1355.12s	5444569	12
			X	51.33s	20509	11
X			X	52.50s	20500	11
		X	X	49.71s	20509	11
X		X	X	49.68s	20500	11
	X		X	62.89s	6291	16
X	X		X	14.84s	6282	16
	X	X	X	14.89s	6291	16
X	X	X	X	14.99s	6282	16
Solving for minimal-latency solutions only:						
				325.65s	88630	12
X				346.81s	88630	12
		X		323.35s	88630	12
X		X		298.69s	88630	12
	X			80.91s	22047	12
X	X			76.77s	22047	12
	X	X		75.61s	22047	12
X	X	X		80.07s	22047	12
			X	0.95s	392	4
X			X	0.91s	392	4
		X	X	0.92s	392	4
X		X	X	0.87s	392	4
	X		X	0.83s	365	4
X	X		X	0.95s	365	4
	X	X	X	0.89s	365	4
X	X	X	X	0.82s	365	4

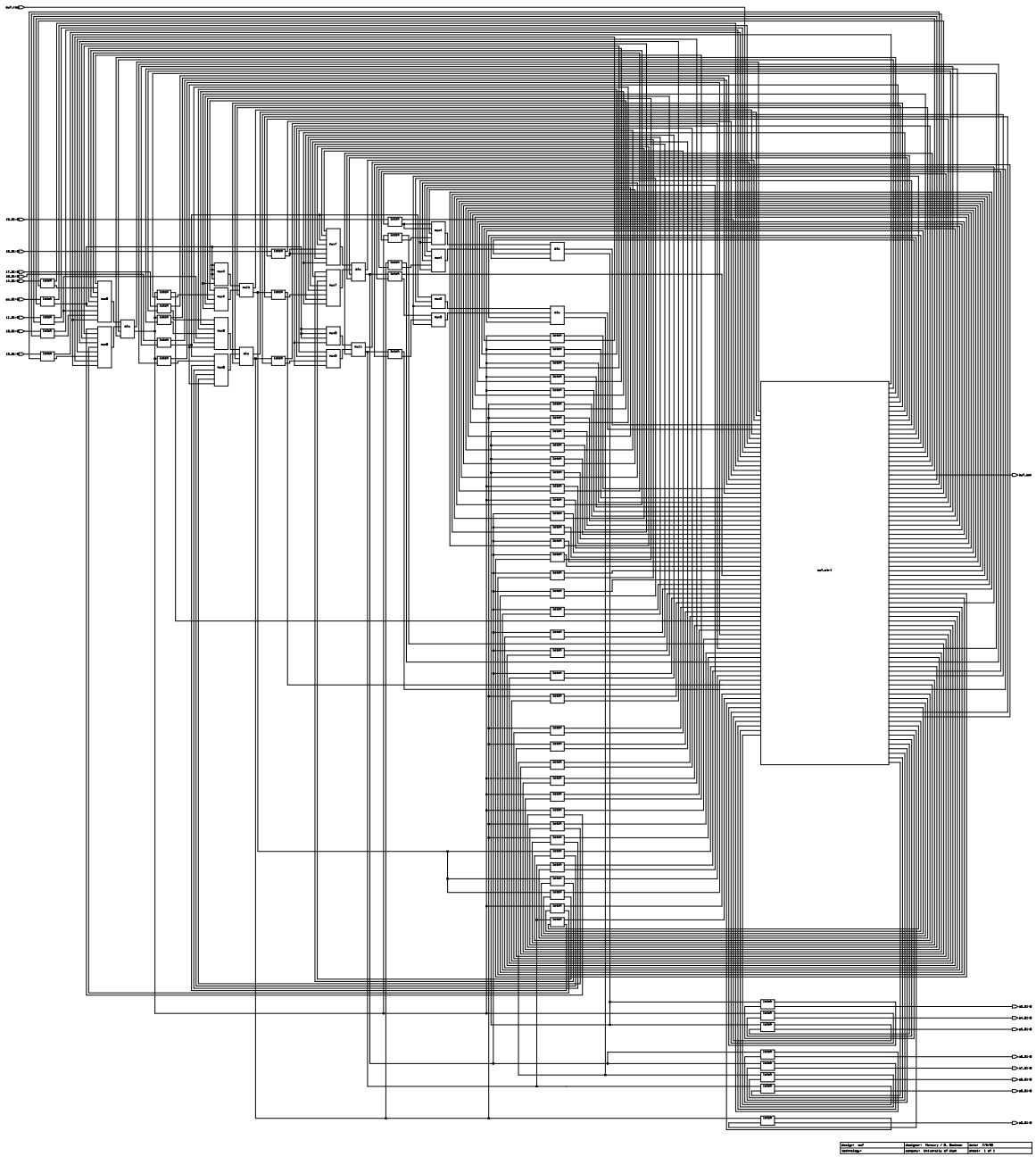


Figure 6.5. Elliptical wave filter datapath.

the filters and using none of the filters is shown in Table 6.5. Here, some of the nonfiltered Pareto points are better than the filtered solutions. In addition, the quality of solutions between using all of the filters except the no change in objectives filter is also compared.

6.2.1 Comparison with Synchronous Methods.

To compare our methods with synchronous designs the elliptical wave filter is used with modified resource delays. In this case, the minimum, typical, and maximum delays for ALU operations is set to one, and for multiply operations each delay is set to two. Because the minimum, typical, and maximum delays are all equal, the model corresponds to a synchronous design. Then to compare our results with those obtained in [36], the maximum delay of the system is set to 21 time units. This means exploration finds all solutions with a delay equal to, or less than 21. The area of a multiplier is modeled to be twice the size of adders. Using all filters and the hierarchal approach to exploration, it took just over 10 seconds to find all solutions in which the latency of the system is between 17 and 21 time units. Our results are comparable with FDS, FDLS, and ASAP methods. Figure 6.6 shows the results. It shows that the more time given for the system to complete, the less adders and multipliers are required because operations are serialized and share fewer functional units. The FDLS method found better results for a case where the delay of the system is 18 this result, however, was achieved by re-timing.

Next, the delay of the adders and multipliers were scaled by a factor of 10. The granularity is adjusted to allow for the modeling of a typical delay. A typical delay of 9 for adders and 17 for multipliers is used. The system is then optimized for typical delay with a maximum system delay of 210 time units. Figure 6.6 shows the results. Again,

Table 6.5. EWF: comparison of unique solutions using hierarchal approach.

No Filters		With All Filters		Except No Change	
Area	Delay	Area	Delay	Area	Delay
272	57	272	72	272	61
287	50	287	65	287	59
302	48	302	61		
310	45	310	50	310	48
325	40	325	48	325	40
363	39	340	40	363	39
378	38	378	38	378	38
416	37	416	37	416	37

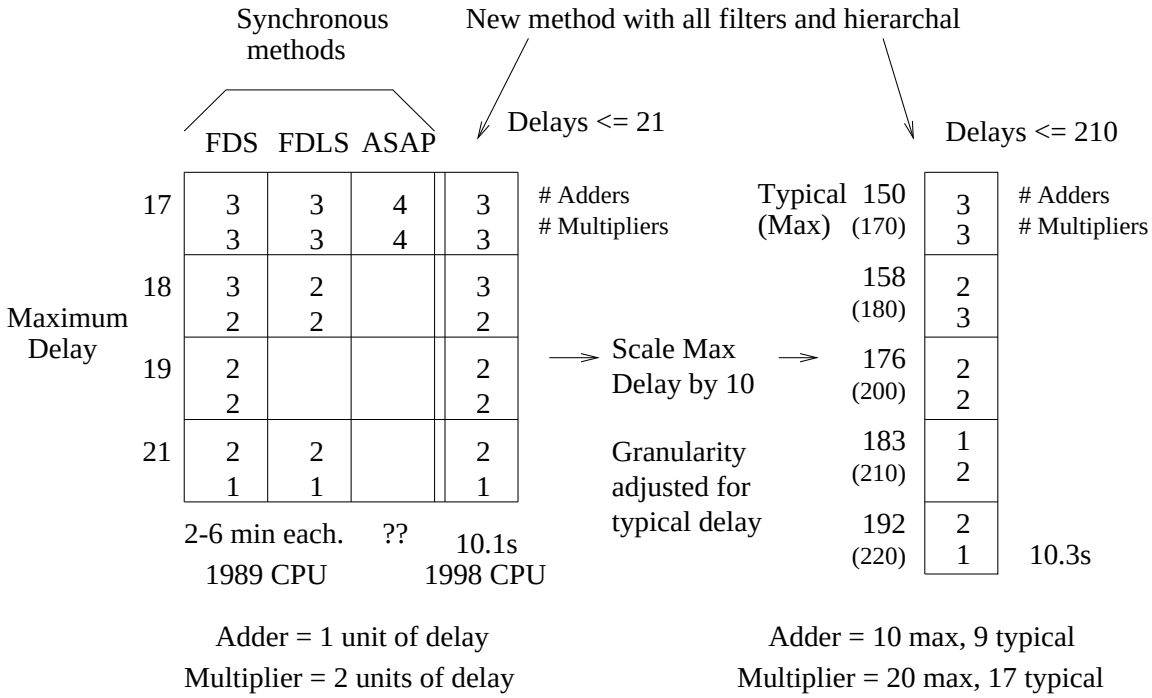


Figure 6.6. Comparison with synchronous methods.

exploration took just over 10 seconds, and several solutions were obtained. While the required time to find the solutions remained constant, the FDS and FDLS methods at this point become computationally unreasonable.

It should also be noted, that several nonintuitive results were obtained. For example, the case were the typical delay is 158, and the case where the typical delay is 183. In both of these solutions, the number of allocated adders is less than the number of allocated multipliers. This is because the typical delay of multipliers compared with its worst-case delay is proportionally less than the typical delay of adders and their worst-case delay. Hence, the typical delay of the system can be optimized in greater proportion when more multipliers are on the critical path in place of adders.

6.3 Inverse Discrete Cosine Transform

The inverse discrete cosine transform (IDCT), is the most difficult example to solve because of the high degree of parallelism between operations. The data flow graph for the IDCT is shown in Figure 6.7. The only reasonable method to solve this problem is to

use the hierarchal approach. The results, using this method with a block size of four, are shown in Table 6.6. A sample minimum latency datapath using this method is shown in Figure 6.8.

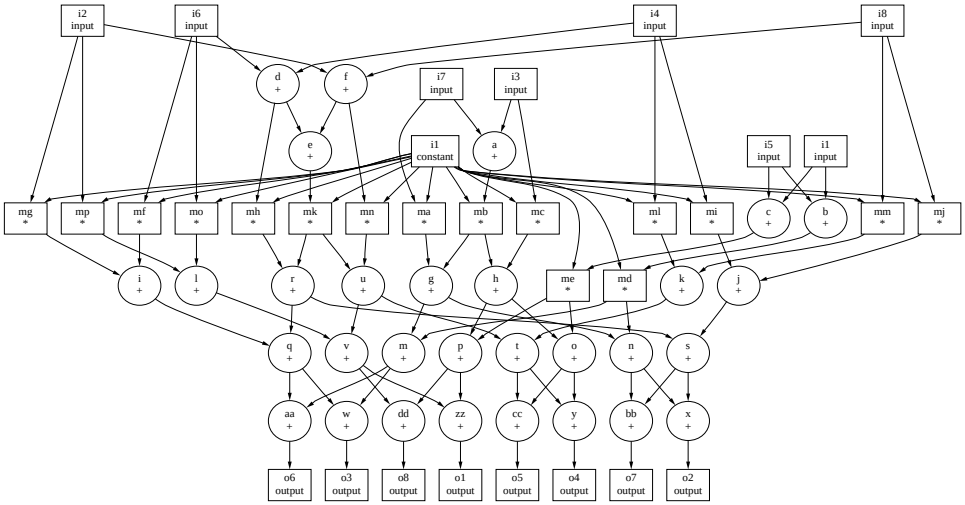


Figure 6.7. Inverse discrete cosine transform data flow graph.

Table 6.6. IDCT: experimental results using hierarchal approach.

Filters						
Implied	Redundant	Shared	No Change	CPU Time	Size	Solutions
X						
		X				
X		X				
	X					
X	X					
	X	X				
X	X	X				
			X	5459.77s	1542648	1142
X			X	5268.17s	1542647	1142
		X	X	5350.47s	1542648	1142
X		X	X	5441.79s	1542647	1142
	X		X	4407.99s	1245450	1142
X	X		X	4421.15s	1245449	1142
	X	X	X	4405.91s	1245450	1142
X	X	X	X	4413.49s	1245449	1142
Solving for minimal-latency solutions only:						
				15.35s	9885	62
X				15.37s	9885	62
		X		15.38s	9885	62
X		X		14.77s	9885	62
	X			13.14s	7511	62
X	X			14.00s	7511	62
	X	X		13.78s	7511	62
X	X	X		13.58s	7511	62
			X	5.80s	898	148
X			X	5.79s	898	148
		X	X	5.92s	898	148
X		X	X	6.19s	898	148
	X		X	6.29s	898	148
X	X		X	6.27s	898	148
	X	X	X	5.99s	898	148
X	X	X	X	5.92s	898	148

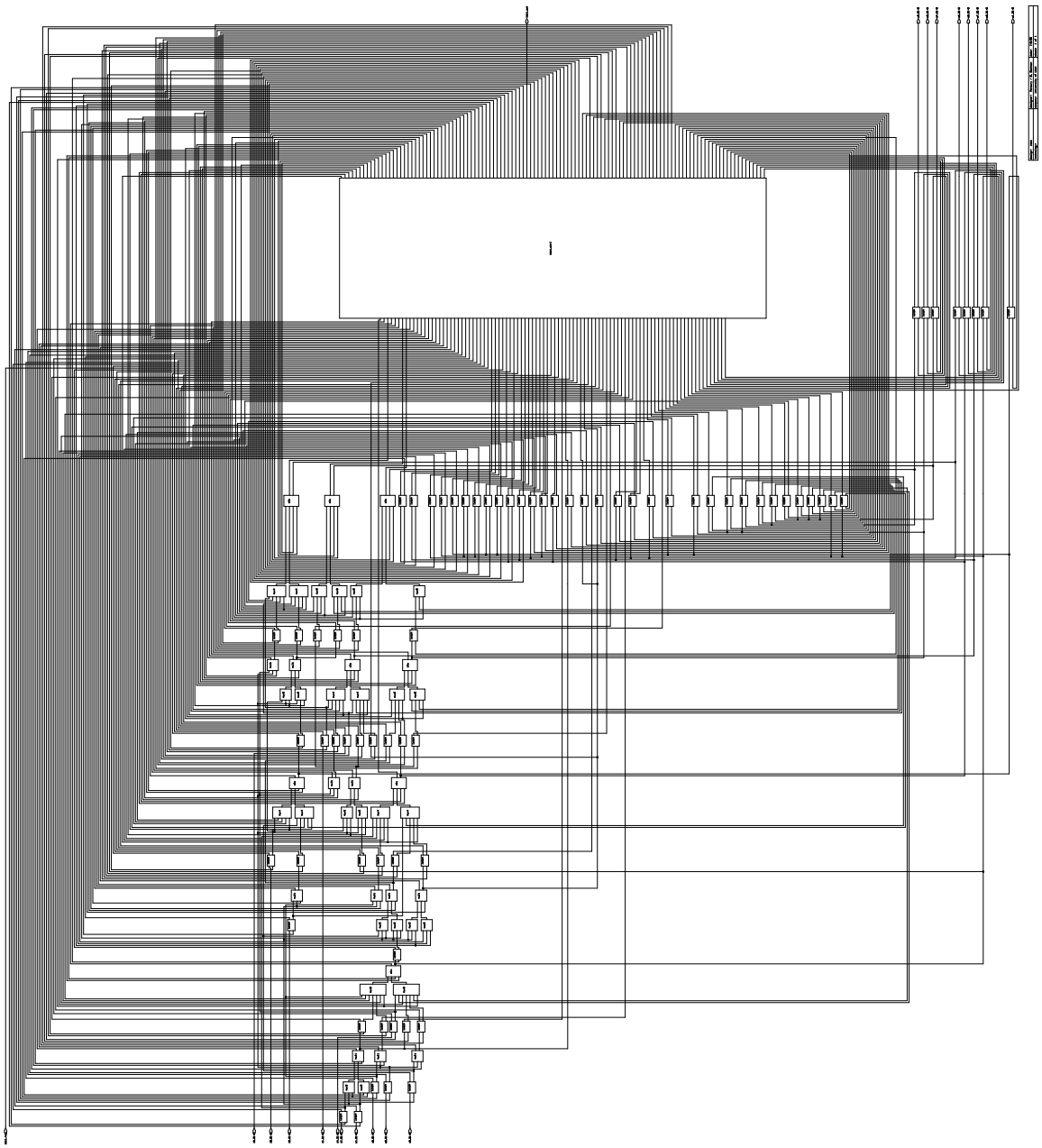


Figure 6.8. IDCT: sample minimum latency datapath.

CHAPTER 7

CONCLUSIONS

A man who has lived in many places is not likely to be deceived by the local errors of his native village; the scholar has lived in many times and is therefore in some degree immune from the great cataract of nonsense that pours from the press and the microphone of his own age.

—C. S. Lewis

Architectural-level synthesis of asynchronous circuits is indeed a very difficult problem. Several factors contribute to this. First, accurately calculating the typical delay of a system is hard. Second, the design space grows at an exponential rate. Finally, exact methods of determining resource sharing are computationally infeasible.

There are several advantages over synchronous architectural-level synthesis methods. For example, scheduling of resources is optional. After all, for asynchronous design, resource sharing determines the final schedule. Regardless of when operations are scheduled, they always execute as-soon-as-possible, because of the asynchronous control paradigm. Second, the cycle-time of the clock is one objective that does not need to be considered. For synchronous design, determining the optimal cycle time for a design can itself be an intractable problem. For asynchronous design, this problem is not a factor.

Architectural-level synthesis is beneficial in producing optimized designs in a short time and is a vital part of analysis in the design conceptualization phase. It also elevates the abstraction level of models to that of hardware languages. This supports a way of reasoning about multiple objectives in a design using a common framework.

Part of this research necessitated the design and development of a CAD tool for exploration, simulation, and analysis of asynchronous circuits at the architectural-level. This brings an automated design flow for asynchronous circuits one step closer to realization. While most research in asynchronous circuits focuses on controller synthesis, this work addresses data path synthesis.

A methodology for the design and synthesis of asynchronous circuits from high-level specifications has been presented. Our method extends synchronous methods of schedul-

ing and resource allocation to asynchronous circuit design. Techniques presented in this work have been used for the architectural optimization of systems.

The large size of the design space has been addressed and several filters have been proposed and implemented to reduce the required exploration of the design space. In addition, a hierarchal approach has been presented and applied, allowing large complex designs to be optimized.

An automated method for generating an optimal data path and its control has been presented. The structures are specified using VHDL to provide a standard language interface. The benefit of this method is not only using a style compatible with synchronous simulation tools, but also one that is compatible with asynchronous controller logic synthesis techniques. The method and algorithms presented have been implemented in the tool `Mercury`. Using the tool, a set of examples has been synthesized.

It was found that the filters are very effective in reducing the required exploration time. When heuristic methods are used, there is a reasonable trade-off between the time required to generate a solution, and the quality and quantity of solutions. Where exact methods failed to efficiently solve a complex problem, the heuristic methods made the problem manageable.

An actual comparison between our method and hand designs was not performed, but, designs created by trial and error methods usually outperform automated methods. This is because of the extra refinement done by hand which can hide protocol, control, and computation delays. Our method, however, yields a substantial reduction in design time.

Compared with synchronous methods it is demonstrated that the proposed methods are advantageous as time is made more discrete to increase granularity. This is because synchronous methods become computationally infeasible, but, the complexity of our method remains constant regardless of the granularity of time. This is important for asynchronous scheduling, because time can be modeled very accurately without sacrificing performance. It has been illustrated that using resource edges is an effective way to serialize operations and determine scheduling. In addition, it was illustrated that solutions using this method are competitive with traditional synchronous methods.

7.1 Possible Extensions

This work demonstrates a feasible method to perform asynchronous architectural-level synthesis. Several open problems and possible extensions are briefly discussed here.

The current method associates a latch with each data edge in the data flow graph. While this suffices, it is inefficient. Latch sharing can be employed to reduce the number of latches required for a design without, in most cases, affecting latency. It is hypothesized that the latch sharing problem is analogous to the resource sharing optimization, but potentially more complex. This is because not only are the *lifetimes* of variables data dependent, but when the results of a computation are used more than once, there are multiple lifetimes that can be considered for alternative configurations. A thorough analysis of latch sharing methods and their applicability to specific situations and communication protocols is necessary.

Another improvement to the process could be made by generating more realistic information about asynchronous resources. The physical properties of asynchronous devices need to be accurately evaluated. Accurate models of the physical characteristics of asynchronous devices is necessary for architectural-level synthesis to generate optimal solutions.

Furthermore, because low power is an often cited advantage of asynchronous design, the evaluation should include not only the required area and data-dependent delays, but also a detailed power analysis. Generating an accurate model to estimate power consumption in an asynchronous device would be very useful. It would permit power consumption to be a third objective considered in the evaluation of a design.

The goal of an ongoing and promising area of related research is to find efficient methods of calculating the typical delay of a system. It was demonstrated here that accurately evaluating the delay is not a trivial task. `Mercury` currently uses a conservative method to perform this calculation. More accurate calculations will lead to better resource sharing and consequently better solutions.

To reduce the design space, several heuristic and nonheuristic methods have been presented, but they still do not completely harness the exponential explosion of the design space. Additional methods need to be developed to prune the required exploration of a system. One such method may consider *negative information* to prune branches of the exploration tree. Using negative information, a design would be evaluated not only in terms of what resource edges have been added to the data flow graph, but also in terms of which edges were selected *not* to be added to the data flow graph. Both pieces of information potentially contribute to what can be concluded about future configurations along a branch of exploration.

Where heuristic methods are used, they have the potential for improvement. For example, the hierarchal method of decomposing a high-level design into smaller, more manageable blocks may be improved. The grouping of operations into blocks is currently done randomly, a more structured approach may yield improved overall results.

The problem of optimally binding resources remains open for further research. This problem has been given considerable attention for synchronous design and it appears that some of these ideas can be extended to asynchronous design as well.

Another important, but neglected, area related to this research is protocol synthesis. In this work only a four-phase handshake protocol is used. However, this may not be the best choice for a design. Finding the optimal protocol for a design is not a trivial task because of timing considerations, the required overhead of each protocol, and the intricacies of each method. Determining an optimal protocol for asynchronous designs remains open for debate and additional research.

Finally, refining an asynchronous datapath from a structural level to a gate level is required before a realization of a circuit can be made. The modular use of functional units in this works allows a hierarchal approach of refinement to be used. In other words, each resource can be synthesized to the gate level independent of other modules. While this makes the task easier, it is still difficult. An automated, technology independent, approach for this task would further reduce the development time of an asynchronous device.

APPENDIX A

SAMPLE VHDL DATAPATH

```
library IEEE;
use IEEE.std_logic_1164.all;

entity sample is
port(A: in std_logic_vector(31 downto 0);
     B: in std_logic_vector(31 downto 0);
     C: in std_logic_vector(31 downto 0);
     D: out std_logic_vector(31 downto 0);
     sample_req: in std_logic;
     sample_ack: out std_logic);
end sample;

architecture structural of sample is
component ALU
  port(a  : in std_logic_vector(31 downto 0);
       b  : in std_logic_vector(31 downto 0);
       op : in std_logic_vector(1 downto 0);
       res: out std_logic_vector(31 downto 0);
       req: in  std_logic;
       ack: out std_logic);
end component;

component Mult
  port(a  : in std_logic_vector(31 downto 0);
       b  : in std_logic_vector(31 downto 0);
       res: out std_logic_vector(31 downto 0);
       req: in  std_logic;
       ack: out std_logic);
end component;

component mux2
  port(a  : in std_logic_vector(31 downto 0);
       b  : in std_logic_vector(31 downto 0);
       sel: in std_logic;
       res: out std_logic_vector(31 downto 0));
end component;

component latch
```

```
  port(d  : in std_logic_vector(31 downto 0);
       q  : out std_logic_vector(31 downto 0);
       req: in std_logic;
       ack: out std_logic);
end component;

component sample_ctrl
  port(signal Mult_1_mux2_sel  : inout std_logic;
       signal ALU_1_req        : inout std_logic;
       signal ALU_1_ack        : in  std_logic;
       signal ALU_1_op         : out std_logic_vector(1 downto 0);
       signal Mult_1_req       : inout std_logic;
       signal Mult_1_ack       : in  std_logic;
       signal l_1_req, l_2_req : out std_logic;
       signal l_1_ack, l_2_ack : in  std_logic;
       signal A_req, B_req     : out std_logic;
       signal A_ack, B_ack     : in  std_logic;
       signal C_req, D_req     : out std_logic;
       signal C_ack, D_ack     : in  std_logic;
       signal sample_req       : in std_logic;
       signal sample_ack       : out std_logic);
end component;

signal opA_1, opB_2            : std_logic_vector(31 downto 0);
signal Mult_1_mux2_sel         : std_logic;
signal Mult_1_a, Mult_1_b      : std_logic_vector(31 downto 0);
signal ALU_1_res               : std_logic_vector(31 downto 0);
signal ALU_1_req, ALU_1_ack    : std_logic;
signal ALU_1_op                : std_logic_vector(1 downto 0);
signal Mult_1_res              : std_logic_vector(31 downto 0);
signal Mult_1_req, Mult_1_ack  : std_logic;
signal l_1_req, l_1_ack        : std_logic;
signal l_2_req, l_2_ack        : std_logic;
signal A_req, A_ack            : std_logic;
signal B_req, B_ack            : std_logic;
signal C_req, C_ack            : std_logic;
signal D_req, D_ack            : std_logic;
signal A_isig, B_isig, C_isig  : std_logic_vector(31 downto 0);

begin
Mult_1_mux2_1: mux2 port map(B_isig,opA_1,Mult_1_mux2_sel,Mult_1_a);
Mult_1_mux2_2: mux2 port map(C_isig,opB_2,Mult_1_mux2_sel,Mult_1_b);

l_1: latch port map(ALU_1_res,opA_1,l_1_req,l_1_ack);
l_2: latch port map(Mult_1_res,opB_2,l_2_req,l_2_ack);
l_D: latch port map(Mult_1_res,D,D_req,D_ack);

l_A: latch port map(A,A_isig,A_req,A_ack);
```

```
l_B: latch port map(B,B_isig,B_req,B_ack);
l_C: latch port map(C,C_isig,C_req,C_ack);

ALU_1:  ALU port  map(A_isig,B_isig,ALU_1_op,ALU_1_res,
                     ALU_1_req,ALU_1_ack);
Mult_1: Mult port map(Mult_1_a,Mult_1_b,Mult_1_res,
                     Mult_1_req,Mult_1_ack);

CTRL: sample_ctrl port map(Mult_1_mux2_sel,ALU_1_req,ALU_1_ack,
                           ALU_1_op,Mult_1_req,Mult_1_ack,l_1_req,
                           l_1_ack,l_2_req,l_2_ack,D_req,D_ack,
                           A_req,A_ack,B_req,B_ack,C_req,C_ack,
                           sample_req,sample_ack);
end structural;
```

APPENDIX B

SAMPLE VHDL CONTROL

```
library IEEE;
use IEEE.std_logic_1164.all;
use work.nond.all;

entity sample_ctrl is
  port(signal Mult_1_mux2_sel : inout std_logic := '1';
       signal ALU_1_req       : inout std_logic;
       signal ALU_1_ack       : in  std_logic;
       signal ALU_1_op        : out std_logic_vector(1 downto 0) := "00";
       signal Mult_1_req      : inout std_logic;
       signal Mult_1_ack      : in  std_logic;
       signal l_1_req  : out std_logic;
       signal l_1_ack  : in  std_logic;
       signal l_2_req  : out std_logic;
       signal l_2_ack  : in  std_logic;
       signal D_req  : out std_logic;
       signal D_ack  : in  std_logic;
       signal A_req  : out std_logic;
       signal A_ack  : in  std_logic;
       signal B_req  : out std_logic;
       signal B_ack  : in  std_logic;
       signal C_req  : out std_logic;
       signal C_ack  : in  std_logic;
       signal sample_req  : in std_logic;
       signal sample_ack  : out std_logic);
end sample_ctrl;


architecture behavioral of sample_ctrl is
begin
    -- controls latch between nodes opA and opC
    proc1:process
    begin
        wait until ALU_1_ack = '1';
        l_1_req <= '1' after delay(2,4);
        wait until Mult_1_req = '1' and Mult_1_mux2_sel = '1';
        l_1_req <= '0' after delay(2,4);
    end process;
```

```
-- controls latch between nodes opB and opC
proc2:process
begin
    wait until Mult_1_ack = '1' and Mult_1_mux2_sel = '0';
    l_2_req <= '1' after delay(2,4);
    wait until Mult_1_req = '1' and Mult_1_mux2_sel = '1';
    l_2_req <= '0' after delay(2,4);
end process;

-- controls the ack of the entire sample system
proc3:process
begin
    wait until D_ack = '1' and sample_req = '1';
    sample_ack <= '1' after delay(2,4);
    wait until D_ack = '0' and sample_req = '0';
    sample_ack <= '0' after delay(2,4);
end process;

-- controls latch l_D between the nodes opC and sink
proc4:process
begin
    wait until Mult_1_ack = '1' and Mult_1_mux2_sel = '1' and
    sample_req = '1';
    D_req <= '1' after delay(2,4);
    wait until sample_req = '0';
    D_req <= '0' after delay(2,4);
end process;

-- controls latch l_A at the source
proc5:process
begin
    wait until sample_req = '1';
    A_req <= '1' after delay(2,4);
    wait until sample_req = '0';
    A_req <= '0' after delay(2,4);
end process;

-- controls latch l_B at the source
proc6:process
begin
    wait until sample_req = '1';
    B_req <= '1' after delay(2,4);
    wait until sample_req = '0';
    B_req <= '0' after delay(2,4);
end process;

-- controls latch l_C at the source
```

```
    proc7:process
    begin
        wait until sample_req = '1';
        C_req <= '1' after delay(2,4);
        wait until sample_req = '0';
        C_req <= '0' after delay(2,4);
    end process;

    -- controls resource ALU_1
    proc8:process
    begin
      wait until ALU_1_ack = '0' and A_ack = '1' and
                 B_ack = '1' and sample_req = '1';
      ALU_1_req <= '1' after delay(2,4);
      wait until l_1_ack = '1';
      ALU_1_req <= '0' after delay(2,4);
      wait until sample_req = '0';
    end process;

    -- controls resource Mult_1
    proc9:process
    begin
      wait until Mult_1_ack = '0' and B_ack = '1' and
                 C_ack = '1' and sample_req = '1';
      Mult_1_mux2_sel <= '0' after delay(0,1);
      Mult_1_req <= '1' after delay(2,4);
      wait until l_2_ack = '1';
      Mult_1_req <= '0' after delay(2,4);
      wait until Mult_1_ack = '0' and l_1_ack = '1' and
                 l_2_ack = '1' and sample_req = '1';
      Mult_1_mux2_sel <= '1' after delay(0,1);
      Mult_1_req <= '1' after delay(2,4);
      wait until D_ack = '1';
      Mult_1_req <= '0' after delay(2,4);
      wait until sample_req = '0';
    end process;

end behavioral;
```

APPENDIX C

SAMPLE VHDL CONFIGURATION

```
configuration cfg_sample of sample is
  for structural

    -- Latches for data edges
    for l_1: latch use entity WORK.latch(behavioral); end for;
    for l_2: latch use entity WORK.latch(behavioral); end for;

    -- Latches for outputs
    for l_D: latch use entity WORK.latch(behavioral); end for;

    -- Latches for inputs
    for l_A: latch use entity WORK.latch(behavioral); end for;
    for l_B: latch use entity WORK.latch(behavioral); end for;
    for l_C: latch use entity WORK.latch(behavioral); end for;

    -- System Control
    for CTRL: sample_ctrl use entity WORK.sample_ctrl(behavioral); end for;

    -- Resources used
    for ALU_1: ALU use entity WORK.ALU(behavioral); end for;
    for Mult_1: Mult use entity WORK.Mult(behavioral); end for;

    -- Muxes used
    for Mult_1_mux2_1: mux2 use entity WORK.mux2(behavioral); end for;
    for Mult_1_mux2_2: mux2 use entity WORK.mux2(behavioral); end for;
  end for;
end cfg_sample;
```

REFERENCES

[1] A. Aho, R. S., and Ullman, J. *Compiliers: Principles, Techniques and Tools.* Addison-Wesley, Reading, MA, 1988.

[2] Akella, V., and Gopalakrishnan, G. SHILPA: A high-level synthesis system for self-timed circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1992), IEEE Computer Society Press, pp. 587–591.

[3] Alur, R. *Techniques for Automatic Verification of Real-Time Systems.* PhD thesis, Stanford University, August 1991.

[4] Ashenden, P. J. *The Designer's Guide to VHDL.* Morgan Kaufmann, San Francisco, CA, 1995.

[5] Badia, R. M., and Cortadella, J. High-level synthesis of asynchronous systems: Scheduling and process synchronization. In *Proc. European Conference on Design Automation (EDAC)* (1993), IEEE Computer Society Press, pp. 70–74.

[6] Beerel, P., and Meng, T.-Y. Automatic gate-level synthesis of speed-independent circuits. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1992), IEEE Computer Society Press, pp. 581–587.

[7] Beerel, P. A., Burch, J. R., and Meng, T. H.-Y. Efficient verification of speed-independent circuits. In *IEEE Transactions on Computer-Aided Design* (May 1994).

[8] Berkel, K. v. *Handshake Circuits: An Intermediary between Communicating Processes and VLSI.* PhD thesis, Eindhoven University of Technology, 1992.

[9] Berkelaar, M. Statistical delay calculation, a linear time method. In *ACM/IEEE International Workshop on Timing Issues in the Specification and Synthesis of Digital Systems* (1997), IEEE Computer Society Press, pp. 15–24.

[10] Brayton, R., and Spence, R. *Sensitivity and Optimization.* Elsevier, 1980.

[11] Brunvand, E. Designing self-timed systems using concurrent programs. *Journal of VLSI Signal Processing 7*, 1/2 (Feb. 1994), 47–59.

[12] Brunvand, E., and Sproull, R. F. Translating concurrent programs into delay-insensitive circuits. In *International Conference on Computer-Aided Design, ICCAD-1989* (1989), IEEE Computer Society Press.

[13] C-T. Hwang, J.-H. L., and Hsu, Y.-C. A formal approach to the scheduling problem in high-level synthesis. In *International Conference on Computer-Aided*

Design, ICCAD-1991 (1991), IEEE Computer Society Press, pp. 464–475.

[14] Camposano, R., and Rosenstiel, W. Synthesizing circuits from behavioral descriptions. In *IEEE Transactions on CAD/ICAS* (1989), IEEE Computer Society Press, pp. 171–180.

[15] Camposano, R., and Wolf, W. *High-Level Synthesis.* Kluwer Academic, 1991.

[16] Chu, T.-A. *Synthesis of Self-Timed VLSI Circuits from Graph-Theoretic Specifications.* PhD thesis, MIT Laboratory for Computer Science, June 1987.

[17] Coates, B., Davis, A., and Stevens, K. The Post Office experience: Designing a large asynchronous chip. *Integration, the VLSI journal 15*, 3 (Oct. 1993), 341–366.

[18] De Micheli, G. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill, Inc., New York, New York, 1994.

[19] Ebergen, J. C. *Translating Programs into Delay-Insensitive Circuits.* PhD thesis, Dept. of Math. and C.S., Eindhoven Univ. of Technology, 1987.

[20] Furber, S. B., and Liu, J. Dynamic logic in four-phase micropipelines. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems* (Mar. 1996), IEEE Computer Society Press.

[21] Gajski, D. *Silicon Compilation.* Addison-Wesley, 1987.

[22] Gajski, D. *Introduction to High-Level Synthesis.* IEEE Design & Test of Computers, 1994.

[23] Garey, M., and Johnson, D. *Computers and Intractability.* Freeman, New York, 1979.

[24] Gebotys, C., and Elmasry, M. *Optimal VLSI Architectural Synthesis.* Kluwer Academic, 1992.

[25] Hafer, L. J., and Parker, A. C. A formal method for the specification, analysis, and design of register-transfer level digital logic. In *IEEE Transactions on Computer-Aided Design* (1983), IEEE Computer Society Press.

[26] Hashimoto, A., and Stevens, J. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the 8th Design Automation Workshop* (1971), IEEE Computer Society Press, pp. 155–163.

[27] Hauck, S. Asynchronous design methodologies: An overview. Tech. Rep. TR 93-05-07, Department of Computer Science and Engineering, University of Washington, Seattle, 1993.

[28] Italiano, G. F. Amortized efficiency of a path retrieval data structure. In *Theoretical Computer Science* (1986), pp. 48:273–281.

[29] Italiano, G. F. Finding paths and deleting edges in directed acyclic graphs. In *Information Processing Letters* (1988), pp. 28:5–11.

[30] Martin, A. J. The limitations to delay-insensitivity in asynchronous circuits. In *Sixth MIT Conference on Advanced Research in VLSI* (1990), W. J. Dally, Ed., MIT Press, pp. 263–278.

[31] McFarland, M. J. Using bottom-up design techniques in the synthesis of digital hardware from abstract behavioral descriptions. In *Design Automation Conference* (1986), IEEE Computer Society Press, pp. 474–480.

[32] Meng, T. H.-Y., Brodersen, R. W., and Messerschmitt, D. G. Automatic synthesis of asynchronous circuits from high-level specifications. *IEEE Transactions on Computer-Aided Design 8*, 11 (Nov. 1989), 1185–1205.

[33] Molnar, C. E., Fang, T.-P., and Rosenberger, F. U. Synthesis of delay-insensitive modules. In *1985 Chapel Hill Conference on Very Large Scale Integration* (1985), H. Fuchs, Ed., Computer Science Press, pp. 67–86.

[34] Myers, C. J. *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits.* PhD thesis, Stanford University, 1995.

[35] Nowick, S. M., and Dill, D. L. Automatic synthesis of locally-clocked asynchronous state machines. In *Proc. International Conf. Computer-Aided Design (ICCAD)* (Nov. 1991), IEEE Computer Society Press, pp. 318–321.

[36] Paulin, P., and Knight, J. Force-directed scheduling for the behavioral synthesis of asic's. In *IEEE/ACM International Conference on Computer-Aided Design* (1989), IEEE Computer Society Press, pp. 661–679.

[37] Poutré, J. A. L., and van Leeuwen, J. Maintenance of transitive closure and transitive reduction of graphs. In *Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science 314* (1988), Springer-Verlag, pp. 106–120.

[38] Rokicki, T. G., and Myers, C. J. Automatic verificaton of timed circuits. In *International Conference on Computer-Aided Verification* (1994), Springer-Verlag, pp. 468–480.

[39] S. Cicerone, D. Frigioni, U. N., and Pugliese, F. Counting edges in a dag. In *Graph-Theoretic Concepts in Computer Science, Lecture Notes in Computer Science 1197* (1996), Springer-Verlag, pp. 85–100.

[40] Thomas, D., Lagnese, E., Walker, R., Nestor, J., Rajan, J., and Blackburn, R. *Algorithmic and Register Trasfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic, 1990.

[41] Unger, S. H. *Asynchronous Sequential Switching Circuits.* Wiley-Interscience, John Wiley & Sons, Inc., New York, 1969.

[42] Watson, D. *High-Level Languages and their Compilers.* Addison-Wesley, Reading, MA, 1989.

[43] Yalamanchili, S. *VHDL Starter's Guide.* Prentice-Hall Inc, Upper Saddle River, NJ, 1998.

[44] Yellin, D. M. Speeding up dynamic transitive closure for bounded degree graphs. In *Acta Informatica* (1993), pp. 30:369–384.

[45] Yun, K. Y., Dill, D. L., and Nowick, S. M. Synthesis of 3D asynchronous state machines. In *Proc. International Conf. Computer Design (ICCD)* (Oct. 1992), IEEE Computer Society Press, pp. 346–350.

[46] Zheng, H. Specification and compilation of mixed-timed systems using vhdl. Master's thesis, University of Utah, 1998.