Heuristic Bounds for Automated Logic Synthesis

A Heuristic Method For Evaluating Two Extremal Implementations for A RT Level Digital System Design

by

Wuyi Wu

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
in the School
of
Computing Science

© Wuyi Wu 1987
SIMON FRASER UNIVERSITY
January 1987

All rights reserved. This thesis may not be
reproduced in whole or in part, by photocopy
or other means, without the permission of the author.

Approval

Name: Wuyi Wu

Degree: Master of Science

Title of Thesis: Heuristic Bounds for Automated Logic Synthesis

B. K. Bhattacharya
Chairman

L. J. Hafer
Senior Supervisor

J. G. Peters

R. F. Hobson
External Examiner

January 26, 1987

Date Approved

PARTIAL COPYRIGHT LICENSE

I hereby grant to Simon Fraser University the right to lend my thesis, project or extended essay (the title of which is shown below) to users of the Simon Fraser University Library, and to make partial or single copies only for such users or in response to a request from the library of any other university, or other educational institution, on its own behalf or for one of its users. I further agree that permission for multiple copying of this work for scholarly purposes may be granted by me or the Dean of Graduate Studies. It is understood that copying or publication of this work for financial gain shall not be allowed without my written permission.

Title of Thesis/Project/Extended Essay

Heuristic Bounds for Automated logic synthesis

Author: ____________
(signature)

Wu Yi Wu
(name)

Dec 15, 1986
(date)

ABSTRACT

This thesis addresses an efficiency problem in the logic synthesis method of [Hafer 81]. The approach to the problem is to bound the solution space by using heuristics to generate extremal implementations from the behavioral description of a given design. By analyzing the implementations, we can extract a set of heuristic bounds and guidance information to improve the speed of the synthesis process.

Algorithms have been implemented to generate two extremal implementations which minimize the cost and time objectives respectively. These initial implementations provide bounds on the solution space and guidance information on design decisions. Based on this information, a designer can seek out the best solution by purposely exploring a limited set of candidates rather than searching blindly in a vast solution space. The guidance information extracted from the hardware allocations for the initial implementations will guide the synthesis model generator to incorporate only worthwhile variables into the equation system, thus reducing the synthesis solving time.

Two examples are examined to demonstrate how to apply heuristics to narrow the solution space and how to extract guidance information to speed up the synthesis process.

Acknowledgements

I owe a great debt of gratitude to my supervisor, Prof. Lou Hafer. It was he who initiated me into the area of "computer aided design" and gave me thoughtful guidance throughout this research. Without his painstaking efforts in wading through my initial draft and correcting my grammar, this thesis would not be as polished.

I am very grateful to Prof. Joe Peters for being in my supervisory committee; and Prof. Rick Hobson for being my external examiner. Constructive comments and suggestions from both of them have helped to improve the original version of this thesis.

Thanks are also due to Prof. Tiko Kameda and my colleagues: Shiv Prakash and Mimi Kao. They have been generous with comments and suggestions which provided useful ideas while this thesis was forming. Nor can I forget the help provided by Mr. Ed Bryant with his convenient tool "picit", which saved me a lot of time while generating the pictures.

I would like to acknowledge financial support from the Chinese government and from NSERC operating grant A5498 (Lou Hafer, principal investigator).

Finally, I owe equal thanks to my dear parents, who have unfailingly supported my studies. I wish to dedicate this work to them.

Table of Contents

Approval ii
ABSTRACT iii
Acknowledgements iv
Table of Contents v
List of Tables vii
List of Figures viii
1. INTRODUCTION 1
 1.1. Motivation 1
 1.2. Prior Work 3
 1.3. The Problem 6
 1.4. The Approach 7
 1.5. Assumptions about the Underlying Hardware Model 7
 1.6. Outline of the Thesis 8
2. THE M-OPTIMAL SCHEDULE METHOD AND THE CPA TECHNIQUE 9
 2.1. The M-optimal Schedule Method 9
  2.1.1. Definitions And Lemmas 10
  2.1.2. M-optimal Schedule Algorithm 13
  2.1.3. Example 14
  2.1.4. Construction of an M-optimal schedule for SP graph 15
 2.2. The Critical Path Analysis Method 15
  2.2.1. Construct A Network 16
  2.2.2. CPA Algorithm 17
3. SERIAL IMPLEMENTATION 19
 3.1. Objectives and Methods 19
  3.1.1. How to Find the Minimum Set of Registers 20
  3.1.2. Hardware Allocation 26
 3.2. The Serial Implementation Algorithm 27
4. PARALLEL IMPLEMENTATION 29
 4.1. The Hardware Allocation 29
 4.2. Estimate the Execution Time 30
  4.2.1. Build a Network to Represent the Parallel Implementation 30
  4.2.2. Find the Critical Path 31
 4.3. Further Optimization 31

5. GUIDANCE INFORMATION FROM THE SERIAL AND PARALLEL IMPLEMENTATIONS 33
5.1. The Estimation Results 33
5.2. The Guidance Information 34
5.2.1. The Guidance Information From the Serial Implementation 34
5.2.2. The Guidance Information From the Parallel Implementation 35
5.2.3. The General Guidance Information From Both Implementations 36
6. EXAMPLES 40
6.1. Generating Extremal Implementations 40
6.1.1. Logic Example 41
6.1.2. Criss.Cross Example 48
6.2. Computational Improvement Made by Bounds 52
6.2.1. Effects of Time Bounds 52
6.2.2. Effects of Cost Bounds 55
6.2.3. Computational Result Comparison for Bound Extraction 56
6.3. Computational Improvement Using S Set Guidance 56
6.3.1. How to Minimize the S Set 57
6.3.2. Synthesis Time Improvement 61
7. OBSERVATIONS AND CONCLUSIONS 63
7.1. Observations 63
7.2. Contributions 65
7.3. Suggestion for Further Research 66
References 67

List of Tables

Table 6-1: Hardware Elements Available for Logic Example 42
Table 6-2: The Matrix Table For The Storage Element Folding 44
Table 6-3: Hardware Elements Available for Criss.Cross Example 49
Table 6-4: Effect of Tightening Time Bounds 53
Table 6-5: TMAX Experimental Results 54
Table 6-6: Effect of Tightening Cost Bound 55
Table 6-7: Comparison of Different Bound Extraction ways 56
Table 6-8: Model Size Comparison 61
Table 6-9: Model Synthesis Result Comparison 62

List of Figures

Figure 1-1:	CMU-DA System Overview	5
Figure 2-1:	PC Graph and SP Graph	11
Figure 2-2:	M-rank	13
Figure 2-3:	Construction of an M-optimal Schedule	14
Figure 2-4:	Procedure of Scheduling an SP Graph	16
Figure 5-1:	The Parameter Bound Analysis	33
Figure 6-1:	Logic ISPS Behavioral Description	41
Figure 6-2:	Logic Data Flow Representation	43
Figure 6-3:	The Execution Sequence of Logic Example	44
Figure 6-4:	The Network For The Data Flow Of Logic	46
Figure 6-5:	Criss.Cross ISPS Behavioral Description	48
Figure 6-6:	Criss.Cross Data Flow Representation	49
Figure 6-7:	The Network For The Data Flow Of Criss.Cross	51
Figure 6-8:	CPA Result for Criss.Cross	51
Figure 6-9:	Initial S set Specification For Criss.Cross Example	58
Figure 6-10:	Revised S set Specification For Criss.Cross Example	58
Figure 6-11:	Initial S set Specification For Logic Example	60
Figure 6-12:	Revised S set Specification For Logic Example	60

Chapter 1

INTRODUCTION

The research reported in this thesis is a case study in the automated synthesis of digital systems at the Register-Transfer (RT) level. Design automation at this level is still in its infancy. Our primary goal is to speed up the logic synthesis process by introducing heuristics. Heuristic algorithms have been implemented to generate two extremal implementations for a given design described in the high-level hardware description language ISPS. We will show how these heuristics can reduce the solution space and provide useful guidance information for optimal synthesis.

In this chapter, we will briefly introduce some background information and the outline of the thesis.

1.1. Motivation

Why is it so important to automate the logic synthesis process and minimize synthesis time? Because of the exponential increase in VLSI scale and complexity, automatic synthesis systems need to be developed to aid the human designer in managing this complexity.

Logic synthesis is a time consuming step in conventional digital system design. Minimizing the time consumed in the synthesis process will effectively reduce the total design time.

Furthermore, the problem facing the designer in digital system design is a multiple-criterion optimization (MCO) problem. In general, the criteria imposed on a design are often in conflict. Therefore, it is impossible to find a unique optimal solution for a design. Instead, we can find a set of noninferior solutions which reflect the different tradeoffs among the design objectives. A noninferior solution has the property that improving the solution with respect to a given criterion requires a degradation in at least one of the other criteria in the design. To efficiently select the best solution, a comparison should be made among the noninferior solutions. The best solution will be the one which embodies the most acceptable tradeoff among the various criteria. In order to seek out the best solution, it is imperative that the full range of design tradeoffs be explored within an available design space by generating noninferior solutions which emphasize different criteria. Manual design cannot afford this because of its long design cycle and high design cost. Automated logic synthesis systems, however, are able to provide more alternative implementations for a designer to evaluate different tradeoffs.

The RT-level logic synthesis system described in [Hafer 81] was developed as part of the CMU-DA project. It formalizes the data path synthesis problem as an algebraic-relation model and solves it as a mixed-integer linear programming (MILP) problem. Varying the emphasis on different criteria by specifying different objective functions to the system, we can obtain a set of noninferior implementations.

Even with automated synthesis systems, the synthesis time is still an important factor to be considered. Since usually there are a very large number of noninferior

solutions for a given MCO problem, it is infeasible to enumerate all implementations. Solving the synthesis model to generate a single noninferior implementation is itself a time-consuming task. Therefore, it is desirable that the number of alternative implementations generated be kept as small as possible.

A methodology must then be devised to narrow the solution space and to reduce the logic synthesis time. Heuristics are the best candidates. They are adopted in this research.

1.2. Prior Work

This section introduces two previous research efforts which are important to our research. One, the CMU-DA project, provides the Value Trace representation and software, as well as a useful paradigm for the automated design of digital systems. The other, a data path synthesis algorithm developed by Hafer, provides the motivation for the thesis research.

The CMU-DA (Carnegie-Mellon University - Design Automation) project is a major research effort directed at developing a technology-relative structured design aid to help the designer explore a larger number of alternative implementations for a given digital system design. The overall structure proposed for the CMU-DA system is shown in Figure 1-1. The behavioral description of a design is written in the ISPS hardware description language and translated into a parse tree which is then converted into a data flow representation called the Value Trace (VT). The nodes of the VT correspond to operations on data values, and the edges represent the flow of these values between operations. After the design style selector chooses the most suitable

design style, the partitioner groups operations from the abstract design representation into control steps. Tradeoffs between the data and control parts are made at this level. A data/memory allocator decides the number and type of functional modules (operators, registers, data paths and switching functions) needed to implement the data part of a design. A control allocator generates a sequential state machine to control the data part produced by the DM allocator. The module binder then selects the physical modules from the module set library to implement the data and control parts. Finally, a physical DA subsystem handles the physical layout for the proposed implementation and prepares engineering documentation. The interested reader is referred to [Thomas 83] or [Thomas 86] for additional material and references.

The synthesis algorithm developed by Hafer [Hafer 81] uses MILP to solve a RT-level hardware synthesis model, which formally models digital systems by algebraic relations. The relations are generated from the Value Trace data flow representation. They explicitly express conditions and timing relations that must be satisfied by any correct implementation. When this system of relations is solved as a MILP problem, it generates an optimal data part implementation with respect to the objective function. In terms of the activities identified in Figure 1-1, solving the synthesis model performs data/memory allocation and control step partitioning simultaneously.

By applying the MILP algorithm to the synthesis model with different objectives and constraints, we can obtain a set of noninferior implementations of a data part design. This offers a great range of implementations to be evaluated by the designer. Based on these implementations, the designer can make the tradeoff by preferentially weighting the various objectives and choose the best implementation.

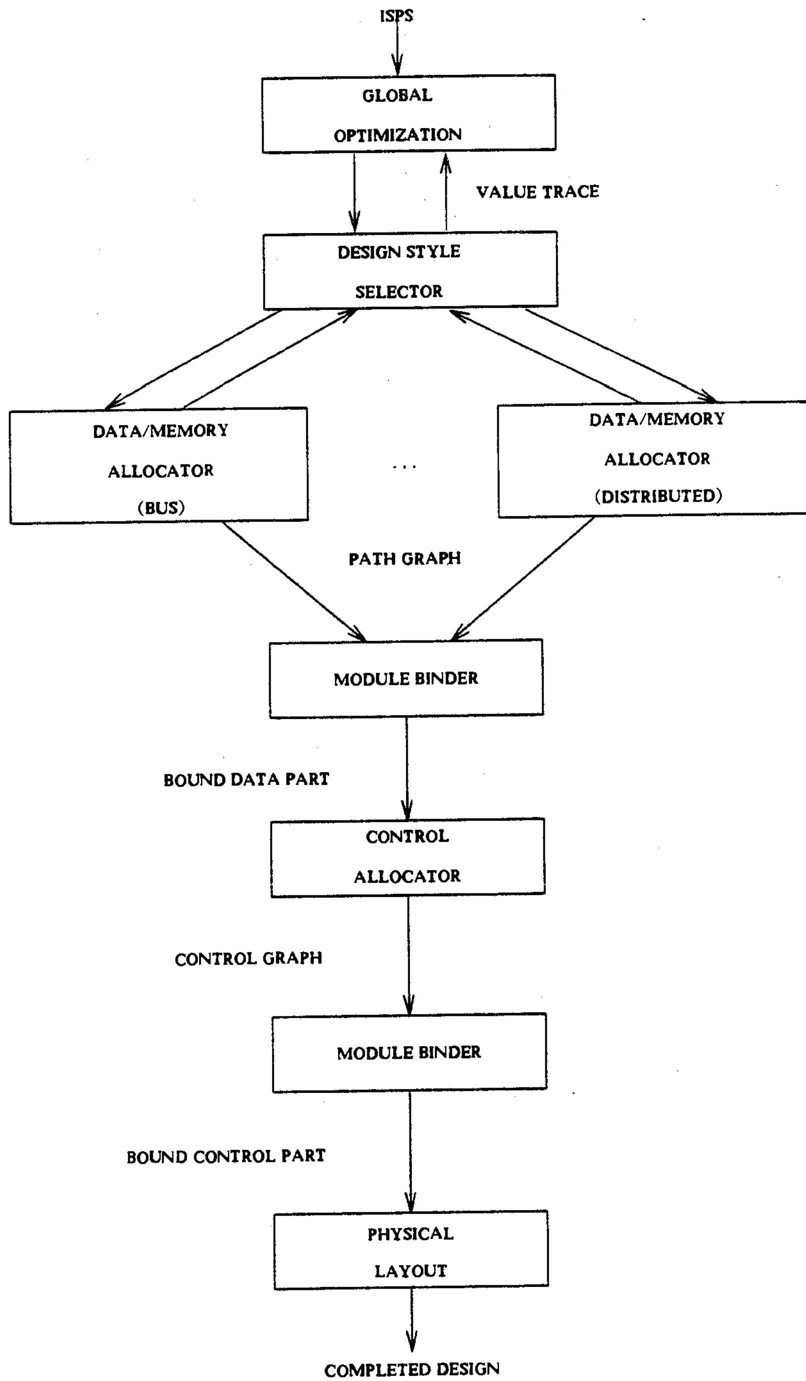

Figure 1-1: CMU-DA System Overview

1.3. The Problem

From the above analysis, we know that it is impossible to find a single optimal solution for a given design to meet multiple objectives which often compete with one another. The optimal solution is embodied in a set of noninferior solutions. But if every candidate solution is generated by solving the synthesis model, even a small problem needs an infeasible amount of computation time. This will limit the ability of the system to allow a designer to explore alternative implementations.

The synthesis model requires many relations and variables to formally specify the constraints and objectives for a design. The system generated for this synthesis model is a large, sparse system. The branch and bound algorithm used by the BANDBX package [Martin 78] and applied in [Hafer 81] is not a specialized algorithm for the solution of the synthesis model, which is the most time-consuming part of the synthesis procedure. So far, no suitable algorithms are available for efficiently handling large, sparse, logic synthesis systems with a reasonably low computation time. But if we can bound the solution space to effectively eliminate non-best-candidate alternatives and reduce the variables incorporated in a synthesis model, we will greatly reduce the computation time.

The research described in this thesis addresses such problems and investigates how to tighten bounds on the solution space with lower complexity algorithms. It also studies how to extract guidance information from two extremal implementations to guide the designer toward noninferior implementations from an early stage in the design.

1.4. The Approach

In order to reduce the time required to solve the MILP problem, it is mandatory to estimate the bounds of the solution space from information that is available before a synthesis model is generated. Improved estimates will aid in pruning the branch-and-bound tree, reducing the computation time of BANDBX; a bounded solution space will limit the choices incorporated into a synthesis model and reduce the number of candidates explored during the design process, again reducing the computation time.

The heuristics introduced in this research intend to accomplish the above goals by generating two extremal implementations. A set of bounds and guidance information is derived from these initial implementations. The time and cost bounds narrow the solution space so that we can generate a noninferior design without a large amount of search. The guidance information can help a designer to make a better choice of parameters and design decisions congruent with his objectives, which will, in turn, guide the synthesis model generator to simplify the equation system.

1.5. Assumptions about the Underlying Hardware Model

The synthesis model uses algebraic relations to model the timing and sequence properties required for correct behavior, under the assumption that all timing requirements can be related to transitions of a global clock signal. Due to the computational complexity of BANDBX, the controller implementation is not considered in the synthesis model and all relations are applied to the data part alone. To ensure that hardware resources are shared correctly among behavioral actions, it is assumed that the timing sequence generated by solving the synthesis model can be implemented

by integral multiples of the clock period of a global clock (or: by phases of a multiphase clock). In order to limit the size of a synthesis model, data paths and switching logic are not explicitly modeled. As a result, data paths between RT level components are considered point-to-point and switching costs and delays do not appear in the model.

Since the approach for improving the efficiency of solving a synthesis model is to use heuristics to generate two extremal implementations, the heuristics must match the above assumptions. The heuristic module will consider only the data part implementation and will evaluate the two extremal implementations in accordance with the model. The cost estimated by the heuristics, thus, will be the cost of operators and storage elements; and the execution time will be the propagation delay of operators plus the setup time, hold time and propagation delay of storage elements.

1.6. Outline of the Thesis

The thesis consists of seven chapters. Chapter 2 introduces two methodologies: M-optimal scheduling, utilized in our research for scheduling the serial execution sequence, and critical path analysis, used for estimating the parallel execution time. In Chapter 3 and Chapter 4, we describe the heuristics developed for generating the serial and parallel implementations. Chapter 5 presents and analyzes the guidance information extracted from the two extremal implementations. Chapter 6 demonstrates how to generate the extremal implementations for a given design with two examples and discusses how to utilize the guidance information to reduce the synthesis time. Finally, Chapter 7 summarizes the results of the thesis and suggests some directions for further research.

Chapter 2

THE M-OPTIMAL SCHEDULE METHOD AND THE CPA TECHNIQUE

In this chapter, we will introduce two methodologies that we apply in the research. One is the M-optimal schedule method developed in [Abdel 76]. It is applied while generating the serial implementation to schedule operations in parallel execution sequences in an optimal serial execution sequence. The other is critical path analysis. We use this technique to estimate the execution time of the parallel implementation and analyze the possibility of further optimization for the implementation.

2.1. The M-optimal Schedule Method

This section describes an algorithm developed to solve one of the scheduling problems investigated in [Abdel 76]. The investigation considers a set of events which are constrained by precedence relations and represented as a directed graph. Each event is associated with a cost. The method intends to order the events into a single sequence, called a schedule, in such a way that the maximum cumulative cost encountered during the execution of the schedule is the minimum one over all schedules. Such a minimax-cost schedule is called an M-optimal schedule.

Obtaining an M-optimal schedule for an arbitrary graph is an NP-complete

problem. In [Abdel 76], special classes of precedence graphs are studied, for which polynomial algorithms have been found. In our application, we are interested in Serial-Parallel (SP) graphs. A polynomial algorithm for finding an M-optimal schedule for SP graphs is developed in [Abdel 76] and is introduced in following subsections.

2.1.1. Definitions And Lemmas

Definitions of PC graph and SP graph

A precedence graph in the form of parallel chains is called a PC graph G as shown in Figure 2-1(a). Figure 2-1(b) illustrates a serial-parallel (SP) graph. Defining the indegree, $d_i(x)$, (outdegree, $d_o(x)$,) of a node x as the number of incoming (outgoing) arcs to (from) x, we have the following recursive definition for testing whether a given graph G is SP or not.

Testing definition for SP graph

G is an SP graph if it can be reduced to a graph consisting of two nodes with an arc between them by a sequence of the following operations:

1. Replace two arcs (u,v) and (v,w) and the node v by a single arc (u,w) if $d_i(v) = d_o(v) = 1$.
2. Delete an arc in parallel with another arc.

Construction of an M-optimal schedule for PC graph

For a PC graph G, let H = 1 2 ... l denote an arbitrary chain in G, where 1,2, ..., l are the nodes of H; let σ = u (u+1) ... v, where $1 \leqslant u \leqslant v \leqslant l$, be any string of consecutive nodes in H; and let a cost (c) associate with each node. We define the cumulative cost (C) of σ at node k $(u \leqslant k \leqslant v)$ as:

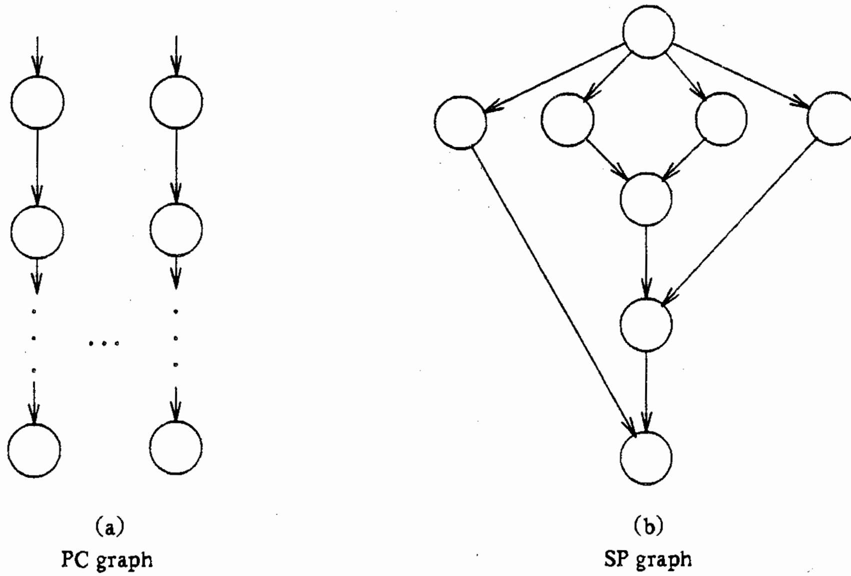

Figure 2-1: PC Graph and SP Graph

$$C(\sigma,\ k) = \Sigma_{j=u}^{k} c(j)$$

So we can write C(σ) for C(σ, v). The maximum cumulative cost in σ is given by:

$$M(\sigma) = \text{Max}\ \{\ C(\sigma,\ k) \mid (u \leqslant k \leqslant v)\ \}.$$

A peak node p is a node in σ such that M(σ) = C(σ, p).

To construct an M-optimal schedule S for G, we must determine two things. One is how to divide H into strings such that the nodes in each string will appear consecutively in S. The other is how to determine the order of the strings in S such that S will not violate the precedence relation defined in each chain H and will have the minimax cumulative cost over all schedules.

Define the Consecutive Nodes

Lemma 1: If σ satisfies the following two conditions :

1. $C(\sigma,i) \geq 0$ for all i, $u \leq i \leq p$;

2. $C(\sigma,i) \geq C(\sigma)$ for all i, $p \leq i \leq v$;

then there exists an M-optimal schedule for G in which the nodes of σ should appear consecutively.

To find all the subchains in H which satisfy Lemma 1, the following notation is useful.

$$\overleftarrow{M}(H,i) = \mathrm{Max}\ \{\ C(H,\ j) \mid 0 \leq j \leq i\ \}$$

$$\overleftarrow{m}(H,i) = \mathrm{Min}\ \{\ C(H,\ j) \mid 0 \leq j \leq i\ \}$$

$$\overrightarrow{M}(H,i) = \mathrm{Max}\ \{\ C(H,\ j) \mid i \leq j \leq l\ \}$$

$$\overrightarrow{m}(H,i) = \mathrm{Min}\ \{\ C(H,\ j) \mid i \leq j \leq l\ \}$$

$$m(H) = \overleftarrow{m}(H,\ l) = \overrightarrow{m}(H,\ 0)$$

There are two special minimizing nodes in H defined as:

$$v_1 = \mathrm{Min}\ \{\ i \geq 0 \mid C(H,\ i) = m(H)\ \}$$

$$v_2 = \mathrm{Max}\ \{\ i \leq l \mid C(H,\ i) = m(H)\ \}$$

The desired subchains are constructively defined as follows:

1. Level M-hump :
   If $v_1 \neq v_2$, then $(v_1+1),(v_1+2),\ldots,v_2$ is called a level M-hump.

2. Negative M-hump :
   If $v_1 > 0$, the nodes $(u+1),(u+2),\ldots,v$ form a negative M-hump according to the following construction procedure:
   1). let $v = v_1$;
   2). $p = \mathrm{Min}\ \{\ i \geq 0 \mid C(H,\ i) = \overleftarrow{M}(H,\ v)\ \}$ and
   $u = \mathrm{Min}\ \{\ i \geq 0 \mid C(H,\ i) = \overleftarrow{m}(H,\ p)\ \}$.
   3). If $u > 0$, there is more than one negative M-hump. Let $v = u$ and recalculate p and u to find another negative M-hump.

3. Positive M-hump :
   If $v_2 < l$, the nodes $v, v{+}1, \ldots, u$ form a positive M-hump as follows:
   1). let $v = v_2 + 1$;
   2). $p = \text{Max}\ \{ i \leqslant l \mid C(H, i) = \vec{M}(H, v) \}$ and
   $u = \text{Max}\ \{ i \leqslant l \mid C(H, i) = \vec{m}(H, p) \}$.
   3). If $u < l$, another positive M-hump will be formed by letting $v = u + 1$ and repeating the above calculations.

Determine the Precedence Relation of Subchains in S

We use M-rank to determine the precedence relations of subchains in S. The M-rank of any string σ, denoted $r(\sigma)$, is defined as follows:

$$r(\sigma) = \begin{cases} -1 / M(\sigma) & \text{if } C(\sigma) < 0 \\ 0 & \text{if } C(\sigma) = 0 \\ 1 / (M(\sigma) - C(\sigma)) & \text{if } C(\sigma) > 0 \end{cases}$$

Figure 2-2: M-rank

The criteria for an M-optimal schedule are specified in Theorem 2:

Theorem 2: A schedule S for G is M-optimal if:

1. The nodes of any M-hump are clustered together in S.
2. $r(h_1) \leqslant r(h_2)$ for any two adjacent M-humps, h_1 followed by h_2, in S.

2.1.2. M-optimal Schedule Algorithm

Based on Theorem 2, we can define the following algorithm for constructing an M-optimal schedule for a PC graph G.

1. Determine the M-humps for all chains of G and calculate their M-ranks.

   a. Find v_1 and v_2.

b. Determine the M-humps:

c. Calculate M-rank for each M-hump.

2. Merge the M-humps of all chains in the non-decreasing order of their M-ranks.

2.1.3. Example

The following is an example for the application of the M-optimal schedule algorithm. Consider the PC graph in Figure 2-3 which has three chains. The M-humps of each chain and their M-ranks, as a result of applying step 1, are indicated on each chain.

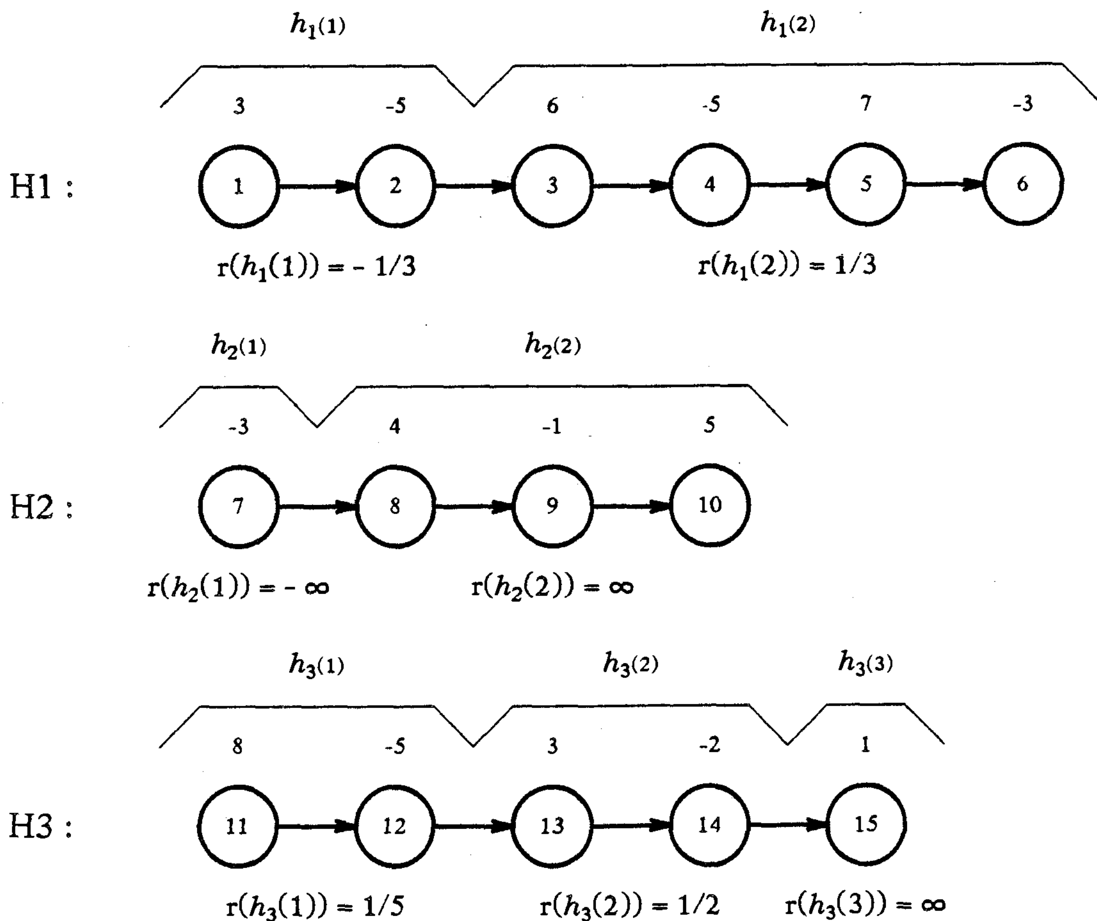

Figure 2-3: Construction of an M-optimal Schedule

The result of step 2 is the following M-optimal schedule S for the PC graph:

S = 7 1 2 11 12 3 4 5 6 13 14 15 8 9 10

2.1.4. Construction of an M-optimal schedule for SP graph

It is easy to see that an SP graph G embeds PC subgraph(s). We can decompose an SP graph into a set of subgraphs recursively, each of which consists of a start node, followed by two parallel chains, followed by a terminal node. Abdel calls such a subgraph a **simple loop** of G.

The approach to find an M-optimal schedule for an SP graph, therefore, is to apply the M-optimal schedule algorithm introduced in section 2.1.2 to the two parallel chains of each simple loop and replace them by a single chain. Then we obtain another simpler SP graph. Repeating this operation as many times as needed, we end up with a single chain which corresponds to the M-optimal schedule for the SP graph. The procedure of changing the SP graph of Figure 2-1(b) into a single chain is shown in Figure 2-4.

2.2. The Critical Path Analysis Method

The standard technique **Critical Path Analysis** has proved very valuable in analyzing networks for planning and arranging schedules. If we represent a project as a network, the minimum time to complete the project is the length of the longest path (*i.e.*, the critical path) of the network. In other words, the activities in the critical path determine the time needed to complete the project. By applying the CPA technique to identify the critical path in the network, we can estimate the completion time of the project. In evaluating the parallel implementation we use this technique to estimate the execution time.

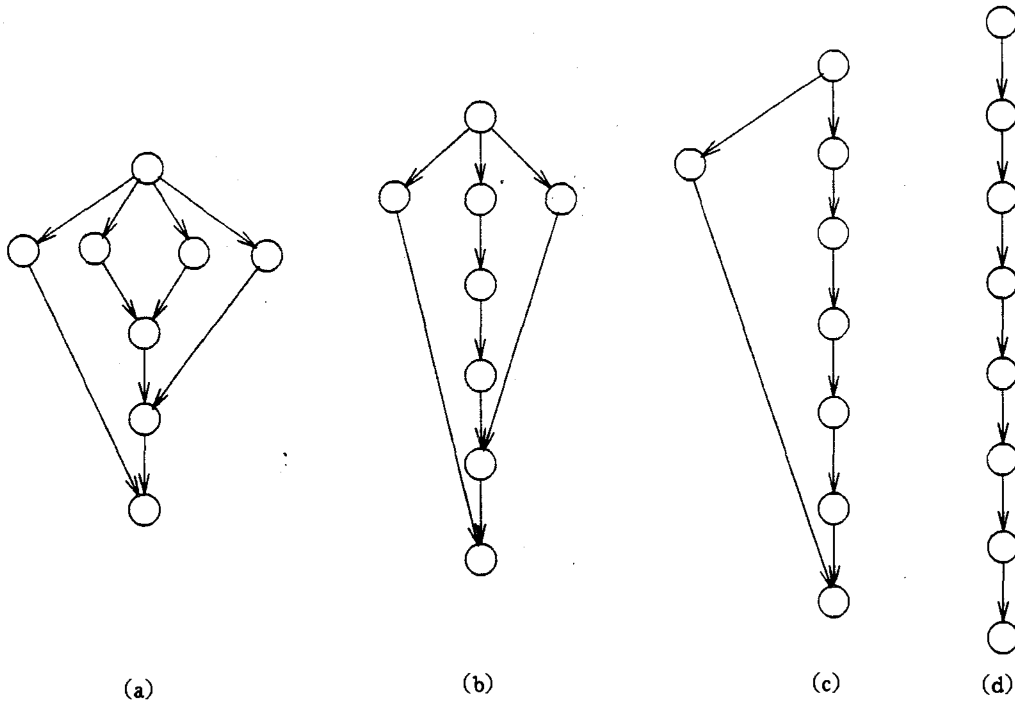

Figure 2-4: Procedure of Scheduling an SP Graph

2.2.1. Construct A Network

To apply CPA, a project must be broken down into its constituent activities, and the duration of each activity must be estimated. In the network representation, each activity is represented as a directed arc, linking two nodes. There is only one start node and one finish node. The finish node indicates the completion of all the activities in the network. No loop is allowed in a network. The nodes in a network are numbered in topological order (*i.e.*, increasing order from the start node to the finish node according to the rule that a node can be numbered only when all its preceding activities have a numbered beginning node).

2.2.2. CPA Algorithm

To determine the critical path, the CPA computation procedure performs a forward pass and a backward pass through the network representing the project. The forward pass computes the early start and early finish times for each activity, proceeding from the start node to the finish node. An activity's early start time is the earliest time when all its preceding activities finish. Its early finish time is its early start time plus its duration. The backward pass computes the late start and late finish times for each activity. It starts by setting the late start time of the finish node equal to its early start time obtained from the forward pass. Other late start and finish times can then be computed recursively from the finish node back to the start node. An activity's late start time is the latest time when the activity can start without delaying its succeeding activities. Its late finish time is its late start time plus its duration, which should be equal to the early start time of its succeeding activity.

After completing the two passes, the total float can be computed for each activity to determine if it is critical. The total float of an activity is equal to its late finish time minus its early finish time. Obviously, the activities with zero total float are critical for the project. A path consisting of critical activities from the start node to the finish node is a critical path.

The above description can be stated as the following algorithm:

The CPA Algorithm

1. Sort the activities so that their beginning nodes are in topological order.

2. CPA computation :

```
For k = 1 to n do              /* n : the number of nodes */
    ees[k] = 0                 /* ees: the early start time of event k */
    els[k] = ∞                 /* els: the late  start time of event k */
End
For k = 1 to m do                           /* m : the number of arcs */
    if ees[j(k)] < ees[i(k)] + d(k)         /* i(k) : start event of activity k */
       ees[j(k)] = ees[i(k)] + d(k)         /* j(k) : finish event of activity k */
End
els[j(n)] = ees[j(n)]
For k = m to 1 do
    if els[i(k)] > els[j(k)] - d(k)
       els[i(k)] = els[j(k)] - d(k)
End
For k = 1 to m do
    ES(k) = ees[i(k)]     /* ES : the early start time of the activity k */
    LF(k) = els[j(k)]     /* LF: the late  finish time */
    EF(k) = ES(k) + d(k)   /* EF: the early finish time */
    TF(k) = LF(k) - EF(k)  /* TF: the total flow */
    if TF(k) = 0
       k is a critical activity
End
```

3. Pick up one critical path.

Chapter 3

SERIAL IMPLEMENTATION

Since it is impossible to satisfy conflicting objectives (maximum speed and minimum cost, for example) at the same time, the module generates two extremal implementations from the Value Trace specification by optimizing the cost and performance objectives respectively.

In this chapter, we will describe the serial implementation. Section 3.1 will discuss the objectives of this implementation and introduce the methodologies used to achieve the objectives. Section 3.2 will then present the algorithm for constructing the serial implementation.

3.1. Objectives and Methods

The objectives of the serial implementation for a given design are:

1. Minimize the design cost to provide an estimate of a lower bound on the design cost.
2. Calculate the execution time to provide an upper bound on the execution time.
3. Provide useful guidance information to reduce the logic synthesis time.

To achieve these objectives, we apply heuristics to efficiently generate a serial implementation and extract the solution bounds from it. By adding times for the

serialized execution sequence, we can easily estimate the execution time. A minimum cost estimation is harder to obtain. To minimize the design cost, we should find the minimum set of operators and registers for the implementation. By searching the set of operators available we can find the minimum set of operators for the serial implementation without much difficulty. But finding the minimum set of registers involves a scheduling problem, since different schedules of the operator execution sequence may have a different maximum number of simultaneously existing values. These problems will be discussed below and the methodologies developed to overcome the difficulties will be introduced.

3.1.1. How to Find the Minimum Set of Registers

The minimum number of registers required in a serial sequence is equal to the maximum number of values simultaneously existing in that sequence. To obtain the minimum set of registers for the serial implementation, we have to create the optimal serial execution sequence first. An optimal serial execution sequence is a serial sequence which has the smallest maximum number of simultaneously existing values among all the possible serial sequences. In general, this scheduling problem is an NP-complete problem (proved in [Abdel 76]). In the next section, we will show that our problem can be represented as an SP graph so that we can apply the M-optimal schedule method to solve the problem in polynomial time.

For now, let us assume that we have the optimal serial execution sequence. How can we assign a minimum set of registers to store all the values in the sequence? The serial execution sequence generated from the VT has a one-to-one correspondence

between an operator and the value it generates. To implement the design, we should assign values to registers so as to reuse a register to store different values generated during the execution without conflicting register use. The register folding algorithm is developed for this purpose.

Register Folding Algorithm

We use a matrix to represent the storage element assignment in the execution sequence. The columns of the table correspond to the input and output values. The rows correspond to the lifetimes of inputs and outputs of the operators. The values are divided into two sets (u set and v set). The u set contains the external input values, while the v set is the internally generated values. Initially, only the elements of the u set are put in the table. The storage elements chosen for the u set represent the initial allocation of storage elements for external input values. The storage elements chosen for the v set can either be replaced or combined with others. To avoid unnecessary comparisons in the folding process, the elements in the v set are only compared with the elements in the u set. If an element of the v set cannot be combined with any element in the u set, it will be added to the u set, forming a new column of the table. The folding algorithm sorts the elements first and then does the comparisons and folding. The result generated by Algorithm 1 will be a valid storage element assignment for the serial execution sequence, using the minimum number of registers.

Some notation introduced in Algorithm 1 :

f_i : the row number to indicate the beginning of the value's lifetime.

l_i : the row number to indicate the end of the value's lifetime.

The superscript v indicates the v set.

The superscript u indicates the u set.

Algorithm 1

1. Put the elements with a lifetime beginning at zero into the u set.
2. Sort the elements of the v set into increasing order according to the value of f_i^v so that no folding chance will be missed if the end of the lifetime of an element in the u set is less than the beginning of the lifetime of an element in the v set.
3. Fold the columns.

```
Begin
  k = m                  /* k : the last column in the table */
                         /* m : the number of elements in u set */
  For i = 1 to n         /* n : the number of elements in v set */
      x = 0              /* x : the folding flag */
          For j = 1 to k        /* j : column number;
                                   k is changeable */
              if l_j^u < f_i^v
                 Begin
                   l_j^u = l_i^v;
                   write(j, i);
                   x = 1;
                   exit loop j
                 End
          End
      if x = 0
          Begin
            k = k + 1;
            l_k^u = l_i^v;
            write(k "=" i)
          End
  End
End
```

The Proof

Lemma 1: The number of storage elements needed in the implementation of a serial sequence cannot be less than the maximum number of simultaneously existing values at any time during the execution of the sequence.

Proof: If the number of storage elements is less than the maximum number of simultaneously existing values, then some value will be lost when the number of values needed to be stored at the same time is more than the

number of storage elements. Therefore, the number of storage elements needed is at least equal to the maximum number of simultaneously existing values. Q.E.D.

Theorem 2: The storage element folding result generated by Algorithm 1 is an optimal storage assignment for a given serial sequence (*w.r.t.* the number of storage elements required).

Proof: By Lemma 1, we have $S_{min} = \text{Max} \{ |v_{t_i}|, 0 \leq t_i \leq n\}$, where S_{min} is the minimum number of storage elements required in the serial implementation, while v_{t_i} is the set of values existing at time t_i.

By showing that the number of storage elements used in the storage assignment generated by Algorithm 1 always equals S_{min}, the correctness of Theorem 2 follows.

Algorithm 1 sorts the values of the v set in non-decreasing order according to the beginning of their lifetimes. When a value's lifetime starts, the algorithm checks the folding table from the beginning to the end to see if there is a free storage element to fold this value into. A storage element is free whenever its old value dies (*i.e.*, reaches the end of its lifetime). If no storage element is free, that means the present number of assigned storage elements is less than the number of values needing to be stored at this time. The algorithm will then assign a new storage element for the value. Otherwise the value will always be folded into a free storage element.

Since the values are sorted and the columns in the folding table have an open end (*i.e.*, the value in a column can be replaced whenever it reaches the end of its lifetime), it guarantees that no new storage element is needed as long as there is a free storage element available. In other words, a new storage element is added into the folding table only when the number of simultaneously existing values is larger than the number of assigned storage elements.

After each folding iteration, the relation $|v_{t_i}| \leq S_{t_i} \leq S_{min}$ is always true, where S_{t_i} is the number of assigned storage elements at time t_i. So the number of storage elements required in the storage assignment generated by Algorithm 1 equals S_{min}, which is the optimal result with respect to the number of storage elements needed. Q.E.D.

Optimal Serial Execution Sequence Generation

We say two operations have data dependencies if we can trace a directed path from an output of one of the two operations to an input of the other. In the Value Trace representation, only essential orderings (*i.e.*, data dependencies) are specified. It does not specify the ordering among those operations which do not have data dependencies (parallel operations, for example). To obtain an optimal serial execution sequence, we must take this into account and reschedule the data-independent operations.

The optimal serial execution sequence is generated from the VT bodies by excluding the non-operator operations and rescheduling the parallel operations as follows:

1. For the serial operations in a VT body, we retain the original serial order for the operator operations to form the serial execution sequence.
2. For a "SELECT" operation, we explore all its branches and select the longest branch to form the sequence.
3. For a "DIVERGE" operation, we apply the M-optimal schedule method to reschedule the parallel operations into an optimal serial execution sequence which contains the minimum number of simultaneously existing values among all the possible serial sequences for the parallel operations.

A VT is an SP graph. To apply the M-optimal schedule method to the VT, we have to serialize the operator operations in each branch starting from the deepest simple loop. For each operator operation in the parallel branches, we assign a cost C(op) = birth(op) - death(op) to it. The function birth(op) is the number of values created by op. The function death(op) represents the number of input values reaching the end of their lifetime at the input of the operator op. The cost C(op) is the net

increase or decrease in the number of storage elements required to hold values due to executing the operator. It can be negative since the execution may release more registers than its output values require (*i.e.*, death > birth). In so doing, we can apply the M-optimal schedule method to our problem and generate a serial sequence which has the smallest maximum cumulative cost over all serial sequences of the parallel operations. In our application, the cumulative cost refers to the number of new registers needed in the sequence. After obtaining such an optimal serial sequence, we can guarantee that the minimum set of registers assigned to this sequence is the minimum set of registers needed in the serial implementation.

3.1.2. Hardware Allocation

Register Allocation

Applying the register folding algorithm to each individual VT body we can generate the minimum set of registers for it. But from the global viewpoint, it can be further optimized since the interface between VT bodies contains redundant values and the registers which do not store external values can be reused by succeeding VT bodies.

To optimize the register allocation, we introduce the "dummy register" and "spare register". A "dummy register" has zero cost and zero delay time, which is used to indicate an external input value from the output of another VT body. The spare registers are those registers storing the internal values. They are collected during the processing of each VT and can be reused in register allocations for succeeding VT bodies, where these registers have zero cost.

In our implementation, after folding the registers for a VT, the module assigns dummy registers for the redundant external input values and searches the spare register list to reuse the registers for storing the values generated in the sequence. If the spare register list is empty or no suitable register is found, the module will explore the register array to allocate new registers.

Operator Allocation

For the operator allocation, we do not know exactly how many and which of the operators will be used until we process all the VT bodies. Therefore, we have to postpone the calculation of bounds and keep recording each operator's usage. For each VT, we search the operator array and find the minimum set of operators to perform all the operations in that VT. After processing all the VT bodies, we can merge the operator sets into a minimum set of operators for the implementation and calculate the cost as well as the execution time.

3.2. The Serial Implementation Algorithm

After last section's discussion, we can summarize the serial implementation algorithm as follows:

Algorithm 2:

1. Generate an optimal serial execution sequence for the given VT.

    a. Connect the operator operations according to the precedence sequence defined in the VT.

    b. Select a longest branch among the "SELECT" branches.

    c. Apply the M-optimal schedule method to reschedule the parallel operations.

2. Search the operator array to find the minimum set of candidate operator(s) for the operations to satisfy functional requirements of the design and record each operator's usage.

3. Apply Algorithm 1 to the optimal serial execution sequence to find the minimum set of registers for the VT.

4. Assign the registers and calculate their cost and delay time T_r.

$$T_r = \sum_{j=0}^{m} (D_{SS_j} + D_{SP_j})$$

where m is the number of operations, D_{SS} is the setup time at the data input of a register, and D_{SP} is the propagation delay of a register.

5. Collect spare registers.

6. If there is a succeeding VT, goto step 1; otherwise step 7.

7. Select the minimum set of operators for the implementation and calculate their cost and delay time T_{op}.

$$T_{op} = \sum_{i=1}^{n} (m_i D_{FP_i})$$

where n is the number of operators; m_i is the number of operations performed by the operator i in the implementation; and D_{FP} is the operator propagation delay time.

Chapter 4

PARALLEL IMPLEMENTATION

The other extremal implementation is the parallel implementation. It improves execution speed by paying the price of increased design cost. The methods used in generating and analyzing this implementation will be discussed in this chapter.

4.1. The Hardware Allocation

The objectives of the parallel implementation are:

1. Minimize the execution time to provide a lower bound on the execution time.
2. Calculate the cost to provide an upper bound on the design cost.
3. Provide useful guidance information and analysis results.

To accomplish the objective of minimizing the execution time, we assign each operator operation an operator to let the operations be executed as concurrently as possible. The operator elements in the operator array are sorted increasingly according to the number of functions they perform. The operators with the same functions in the array are sorted increasingly according to their delay time. Searching this array, we can easily assign an operator for each operator operation by a one-to-one mapping function. The register allocation for parallel implementation is much easier than that for serial implementation. We do not need to place a register between two

operations, since operations are implemented by different operators. The only place where registers are needed is the interface between two VT bodies. Due to the redundant specification of the external values in VT bodies' interfaces, we can remove some redundant registers from the interfaces.

After the initial hardware allocation, the cost can be calculated. Some opportunities for optimizing the allocation may remain, and these will be explicitly displayed by the result of the critical path analysis for the parallel implementation. This will be discussed in section 4.3.

4.2. Estimate the Execution Time

The execution time of the parallel implementation is that of the longest (timewise) branch. This longest timing path can be found by using the CPA technique introduced in Chapter 2. Before we can apply CPA to solve our problem, we must represent the parallel implementation as a network. The problems will be discussed one by one in the following subsections.

4.2.1. Build a Network to Represent the Parallel Implementation

As introduced in Chapter 2, a network suitable for CPA has only one start node and one finish node. Its internal arcs and nodes are linked according to their precedence relations. The precedence relation in our problem is that an event which produces a value must happen before any event which makes use of the value.

To construct a network representation for the parallel implementation, we use nodes to represent registers and operators, and arcs to represent the activities of value

transmission. Usually in a VT, there is more than one external input and more than one external output. So we cannot specify them as the start node or finish node of the network. To solve the problem, we introduce two dummy nodes, one for the start node and one for the finish node, which have zero time duration. The start node will be linked to all the external inputs, and all the external outputs will be linked to the finish node. The network of internal activities will be formed based on the data dependencies.

4.2.2. Find the Critical Path

After checking the correctness of the network, we can apply CPA to find the critical path in the network. By completing the forward pass and backward pass computations through the network as described in CPA algorithm, we can find all the critical activities which have zero total float. These critical activities may form more than one critical path. We only need to pick up one of them to estimate the execution time.

4.3. Further Optimization

The CPA result explicitly specifies the timing flexibility of each activity by its total float. There is a slack (*i.e.*, non-zero total float) in each non-critical activity, which is the amount of time that the activity can be delayed without affecting the total execution time. This characteristic provides us with room to attempt to trade fast but expensive hardware elements for slow but cheaper ones for the non-critical activities. But before we can make the replacement, we must make sure that these elements are not used in any critical activities and that they will not delay their activities more than their slack time.

In our implementation, the cost estimation module checks for optimization possibilities as described above. If an optimization is possible, it will modify the initial hardware allocation and recalculate the design cost. This optimization reduces the design cost without delaying the total execution time.

Chapter 5

GUIDANCE INFORMATION FROM THE SERIAL AND PARALLEL IMPLEMENTATIONS

The heuristics for generating the serial and parallel implementations not only give a useful set of bounds for reducing the solution space but also provide some valuable guidance information for accelerating the logic synthesis process. This chapter will analyze and summarize the guidance information extracted from these two extremal implementations.

5.1. The Estimation Results

The estimation parameters that we get from the initial implementations described in the previous two chapters are shown in Figure 5-1.

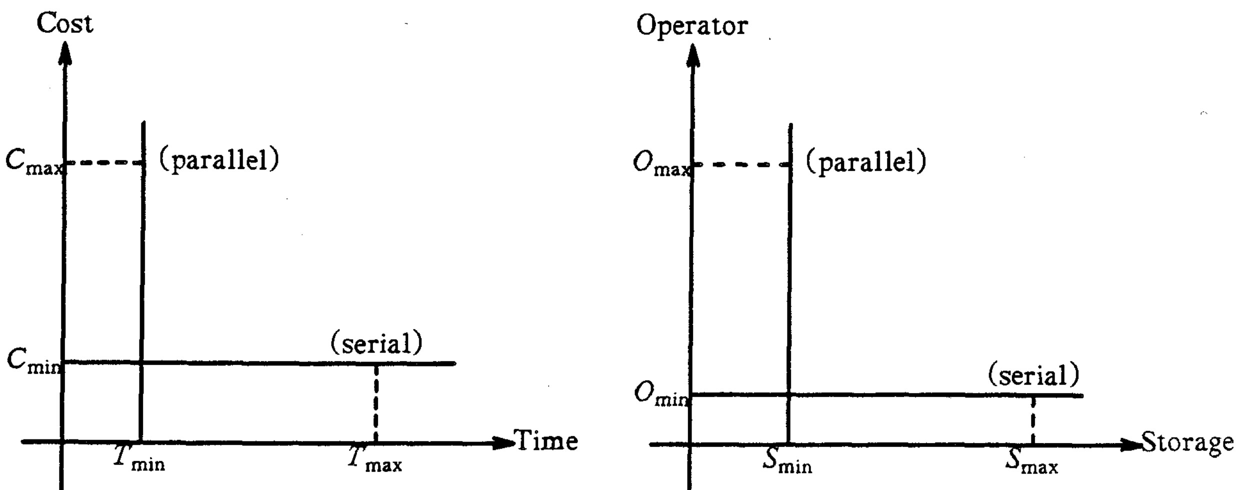

Figure 5-1: The Parameter Bound Analysis

The serial implementation generates estimates for an approximate on the design cost, C_{min}; an upper bound on the execution time, T_{max}; a lower bound on the number of operators required, O_{min}; and an upper bound on the number of registers, S_{max}.

The parallel implementation generates estimates for an upper bound on the design cost, C_{max}; a lower bound on the execution time, T_{min}; an upper bound on the number of operators required, O_{max}; and an approximate on the number of registers, S_{min}, required by any noninferior implementation.

These parameters bound the solution space and help the MILP algorithm to prune inferior candidate implementations. They also explicitly specify the boundary of tradeoffs between the design cost and the execution time, which will help the designer to make a satisfactory tradeoff to realize the design objectives.

5.2. The Guidance Information

5.2.1. The Guidance Information From the Serial Implementation

T_{max} gives an upper bound for the execution time. Thus the designer can set the output valid time, $T_{IA}(O_0)$, less than or equal to T_{max}, rather than an arbitrarily large value.

C_{min} gives a lower bound estimate for the cost. It helps to determine whether there is a feasible solution to satisfy the cost constraint for a given problem. As we pointed out that this lower bound is approximate in Chapter 3, we should ease this bound a little bit in the practical design.

O_{min} specifies the minimum number of operators needed to implement a given design.

S_{max} provides an upper bound on the number of registers required.

The storage element assignment generated in the serial implementation provides useful guidance information for reducing the number of registers specified to the synthesis model generator as alternatives for storing a given value.

5.2.2. The Guidance Information From the Parallel Implementation

T_{min} gives a lower bound on the execution time, which helps the designer to specify $T_{IA}(O_0) \geqslant T_{min}$ and rejects infeasible designs which require that the execution time be less than T_{min}.

C_{max} gives an upper bound for the cost. If the objective function is to minimize the execution time, C_{max} can be a constraint.

S_{min} is an approximate minimum number of registers required by a given design. Some of the variables representing these registers can be forced to the extreme value 1, pruning the branch-and-bound tree and reducing the number of subproblems to be explored.

O_{max} specifies the upper bound on the number of operators for a given design.

The slack times calculated by the CPA timing analysis procedure reveal possibilities for further refining the design. The network representation for the

parallel implementation reveals all possible execution orders. This makes the S set specification to the synthesis model generator much easier for the designer.

5.2.3. The General Guidance Information From Both Implementations

The guidance information extracted from the two extremal implementations makes the tradeoffs more predictable and gives the designer a general estimation of the limits for a given design. Based on this information, the designer can make some compromise between these two extremal implementations.

By analyzing the guidance information, the designer can more precisely specify the output valid time, $T_{IA}(O_0)$, the cost constraint, the sets of operators, F, and the sets of registers, S, which are the inputs to the synthesis model generator program IDDMA.

The equation system generated by IDDMA is partly based on the F and S sets, which describe what hardware elements are candidates to implement each behavioral action. Reducing the size of the F and S sets decreases the number of integer variables required to specify design decisions and constraints, thus reducing the time required to solve the MILP problem. Note that we cannot delete the elements of the sets blindly since this might eliminate noninferior implementations. To guarantee optimality each set of hardware elements should contain all hardware elements which are capable of performing the behavioral action. But this approach is overly conservative. It expands the size of the synthesis model and increases the synthesis process time. It is desirable to obtain guidance information which can lead the designer to properly specify small sets of registers and operators to limit the freedom of choice incorporated into a synthesis model without eliminating noninferior solutions.

The register allocation generated from the serial implementation and the network created for the parallel implementation provide the necessary information for specifying small sets of registers for a given design, suitable for synthesizing noninferior implementations with different execution sequences. The storage element assignment of the serial implementation is a best assignment for the implementation with respect to the number of registers used and it also satisfies the storage assignment for the parallel implementation. We can, therefore, use the storage assignment of the serial implementation as an initial specification of the S sets and explore all mixed serial/parallel execution orderings to complete the specification. Any execution sequence which is neither serial nor parallel is called a mixed ordering (or sequence), *i.e.*, it contains at least two simultaneously executed operations and is not a fully parallel sequence. The arcs in the network specify the precedence relations of the nodes. If two operation nodes do not have any precedence relation (*i.e.*, no path connecting them), we call them data-independent nodes. A data-independent set consists of all operation nodes which are data-independent with one another. For each two data-independent nodes x and y, there are three possible orderings: x -> y; y -> x and x,y (parallel). To generate mixed orderings, we can fix one node in each data-independent set, and then move each one of the other nodes in the same data-independent set ahead, behind, and parallel with the fixed node. Repeating the same steps for each data-independent set and excluding fully serial or parallel orderings, we end up with all the mixed orderings. (It is not a practical way to generate the mixed orderings when the number of data-independent nodes is large.) For each mixed ordering, we number the operation nodes according to their execution order from 1 to n. The simultaneously executed operations receive identical numbers. By

exploring the network with attached S sets, we can discern potential storage usage conflicts, which may occur in some mixed execution sequence, and add the necessary storage elements to the S sets to eliminate any possible conflicts. The procedure for doing this is sketched as follows.

S set specification procedure

1. Specify the storage element assignment of the serial implementation as the initial S set specification in the network.
2. Examine each operator node, X_i, in the network and add a storage element to its S set, S_i, if its output uses the same storage element as one of its inputs and that input value is also used by other operator(s).
3. Explore mixed orderings for storage usage conflicts. (Since the initial S set is suitable for synthesizing the two extremal implementations, conflicts will only occur in some mixed orderings.)

   a. For each mixed ordering, number the operation nodes according to their execution order in that ordering.

   b. Operation nodes with the same number can not use the same storage elements for their outputs. If this does happen in the ordering under the current S set specification, additional storage element(s) must be added into the corresponding S set(s) to eliminate the conflict(s).

   c. No operator can use the same storage element to store its different input values. If this conflict occurs in the ordering, a storage element has to be added to one of its input S sets.

   d. An operator with number *i* can not use the same storage element for its output as that used for the inputs of data-independent nodes with a number larger than *i* if the inputs are from operators with a number smaller than *i*. If this conflict occurs in the mixed ordering, a storage element has to be added into the S set for the output of the operator.

   e. If there are no more mixed orderings to be explored, stop. Otherwise, check the next mixture ordering from step a.

When adding storage elements to an S set, we should try to use previously assigned registers first. If no old register can be used, we will assign a new one. To minimize the number of registers in the S sets, we must keep the number of new registers as small as possible.

Using the guidance information to optimize the S sets will simplify the synthesis model and reduce the synthesis solving time of BANDBX. It will be demonstrated by the examples in the next chapter.

Chapter 6

EXAMPLES

The methodologies and heuristics introduced in previous chapters will be illustrated with two examples of RT level data path design in this chapter. These examples are chosen from the examples used in [Hafer 81]. The presentation of the Logic example will emphasize the generation of the two extremal implementations, while the presentation of the Criss.Cross example will emphasize the extraction of guidance information. For both examples data comparing solution times for synthesis models will be presented to show the effects of incorporating information gleaned from the extremal implementations.

6.1. Generating Extremal Implementations

In the following examples, each design's behavior is described in the ISPS language and translated into the VT representation. Two extremal implementations are generated from the VT. The hardware elements available for the implementations of the two examples are shown in Table 6-1 (Logic) and Table 6-3 (Criss.Cross). The cost and delay figures are obtained by constructing the required operators and storage elements from 7400 series SSI/MSI IC's. The delay and cost might differ slightly from that of actual commercial components, (the basic 7400 series is an aging technology) but the functionality remains representative of the class of hardware elements assumed in constructing the synthesis model. The costs for each element are

based on the retail pricing for the component IC's, plus an estimate of manufacturing overhead cost of $3.00 per IC.

6.1.1. Logic Example

This section demonstrates how to generate two extremal implementations from the VT representation of a given design.

```
Logic:=

BEGIN

  ** Carriers **
  A<0:63>,
  B<0:15>,
  C<0:15>,

  ** Activity **
  action:=
  (B = (C AND (B OR A<48:63>)) XOR (B AND (C OR A<8:23>)))

END
```

Figure 6-1: Logic ISPS Behavioral Description

The ISPS behavioral description for this example is shown in Figure 6-1. The available hardware elements are shown in Table 6-1, in which the registers are sorted in non-decreasing order of their bits; the operators are sorted increasingly according to the number of functions that they can perform. Figure 6-2 presents the data flow representation of the design which is the input to our heuristic module for generating the two extremal implementations.

The Serial Implementation

In the serial implementation, the module chooses the operator f_6 (ALU) to perform the operations OR, AND and XOR, and generates the serial execution sequence shown

storage	bits	IC's	D_{SS}	D_{SH}	D_{SP}	cost
s_0	<8>	(1)SN74195	20 ns.	0 ns.	27 ns.	$5.00
s_1	<16>	(2)SN74273	20 ns.	0 ns.	27 ns.	$8.10
s_2	<16>	(2)SN74273	20 ns.	0 ns.	27 ns.	$8.10
s_3	<16>	(2)SN74273	20 ns.	0 ns.	27 ns.	$8.10
s_4	<64>	(8)SN74273	20 ns.	0 ns.	27 ns.	$32.40

operator	bits	IC's	function	D_{FP}	cost
f_1	<16>	(4)SN7432	OR	27 ns.	$14.72
f_2	<16>	(4)SN7432	OR	27 ns.	$14.72
f_3	<16>	(4)SN7408	AND	18 ns.	$14.72
f_4	<16>	(4)SN7408	AND	18 ns.	$14.72
f_5	<16>	(4)SN7486	XOR	30 ns.	$16.08
f_6	<16>	(4)SN74181	ALU	48 ns.	$19.00

Table 6-1: Hardware Elements Available for Logic Example

in Figure 6-3. In Table 6-2, we use a matrix to express how the register folding algorithm compacts the input and output values into a minimum set of registers. The columns of the table correspond to the storage elements chosen for the input and output values, while the row numbers represent the lifetimes of input and output values of the operators. Each pair of Xs in a column indicates the beginning and the end of the lifetime of a value. Whenever a value reaches its end of the lifetime, its storage element can be released for the use of other values. The Xs enclosed in an ellipse indicate that a value is compacted into a used register. s_1, s_4 and s_2 represent the initial allocation of registers for the external inputs. s_3 is the new register added when folding the internally generated values.

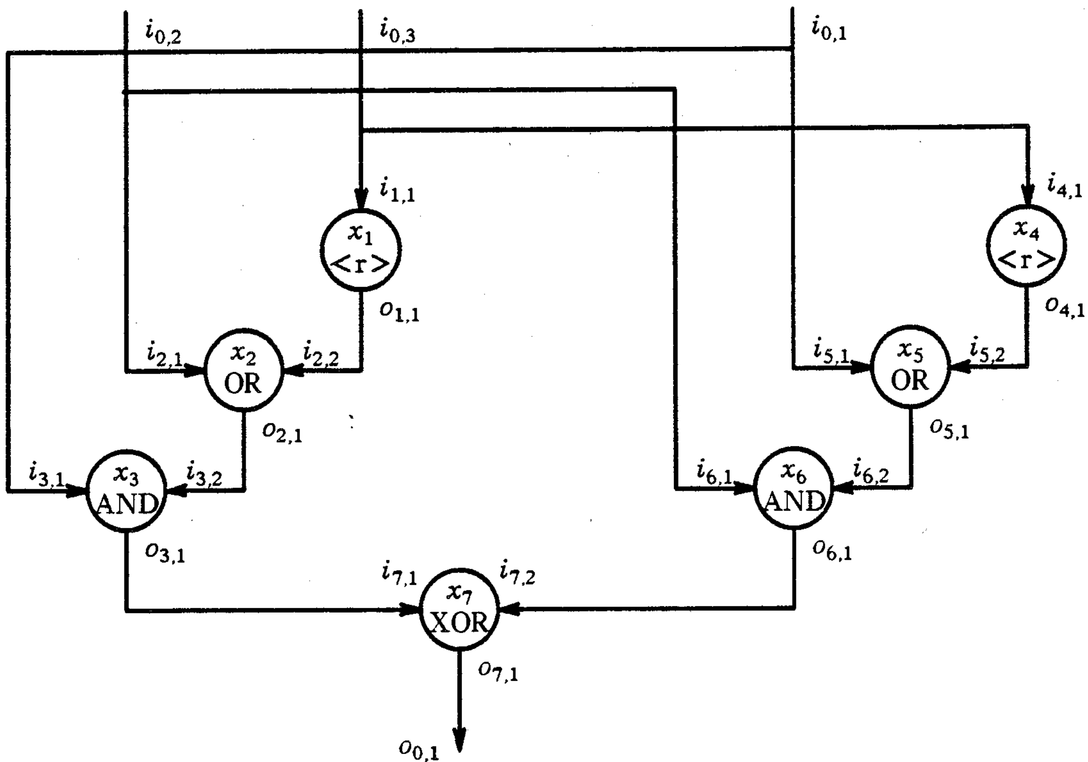

Figure 6-2: Logic Data Flow Representation

After folding the registers, we get following results:

The minimum number of storage elements needed is 4.

The storage elements chosen from Table 6-1 are assigned as follows:

s_1 : for $i_{0,1}$, $o_{5,1}$, $o_{6,1}$, $o_{7,1}$
s_2 : for $i_{0,2}$
s_4 : for $i_{0,3}$
s_3 : for $o_{2,1}$, $o_{3,1}$

The serial implementation provides an upper bound on execution time, T_{max}, of 522 ns, and a lower bound on cost, C_{min}, of \$76.90.

			S_1	S_4	S_2	S_3
External Input		0	X	X	X	
OR	I	1				
	O	2				X
AND	I	3				X
	O	4				X
OR	I	5	X	X		
	O	6	X			
AND	I	7	X		X	
	O	8	X			
XOR	I	9	X			X
	O	10	X			
External Output		11	X			

Table 6-2: The Matrix Table For The Storage Element Folding

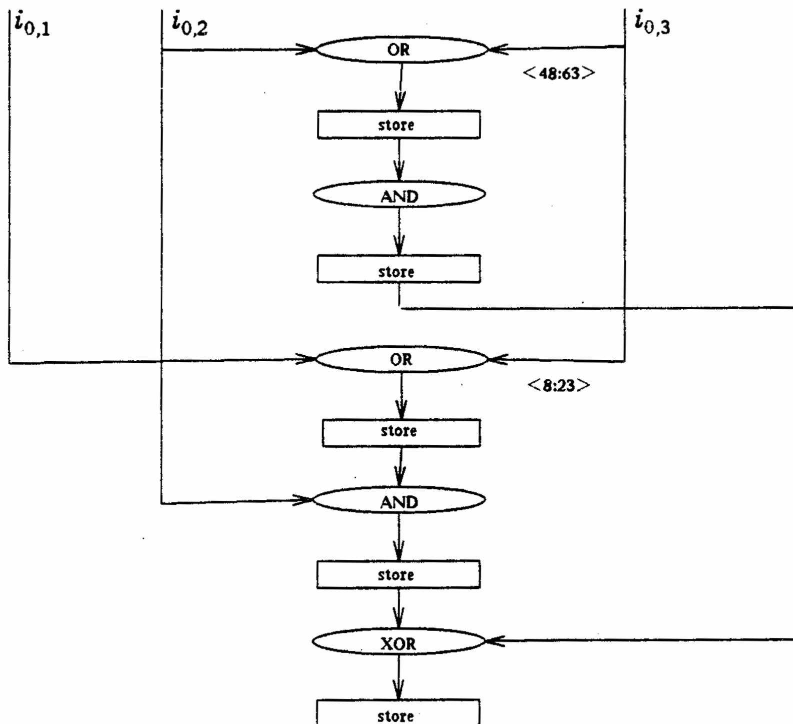

Figure 6-3: The Execution Sequence of Logic Example

The Parallel Implementation

In the parallel implementation, the module assigns each operation to an operator by a one-to-one mapping function. The operator assignment is shown below:

f_1 : for x_2, f_2 : for x_5
f_3 : for x_3, f_4 : for x_6
f_5 : for x_7

The registers assigned for the external inputs are s_1 for $i_{0,1}$, s_2 for $i_{0,2}$, and s_4 for $i_{0,3}$.

To apply CPA to estimate the execution time, the network shown in Figure 6-4 is created from the data flow of Figure 6-2. We introduce two dummy nodes, 0 and 9, to represent the start node and the finish node of the network. Nodes 1, 2 and 3 refer to the external inputs $i_{0,1}$, $i_{0,2}$ and $i_{0,3}$ respectively. Nodes 4, 5, 6, 7 and 8 correspond to the inputs of operators x_2, x_3, x_5, x_6 and x_7. The arcs represent the activities of value transmission from register inputs to operator inputs (for external inputs) or from operator inputs to inputs of other operators (for VT operations). Thus, the duration of an activity is the propagation delay time of a register or an operator.

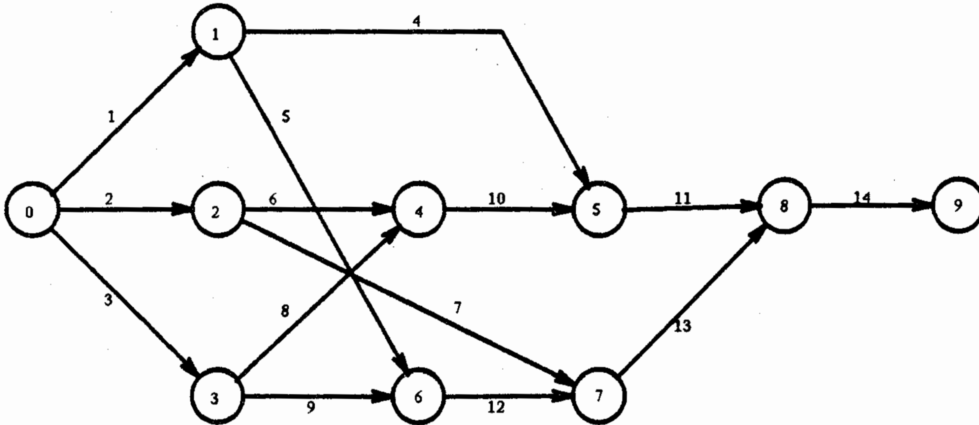

Figure 6-4: The Network For The Data Flow Of Logic

The CPA result is presented below:

*** CRITICAL PATH ANALYSIS FOR VT_5 ***

activity k	start i	finish j	duration d	op for j
1	0	1	0	
2	0	2	0	
3	0	3	0	
4	1	5	47	AND
5	1	6	47	OR
6	2	4	47	OR
7	2	7	47	AND
8	3	4	47	
9	3	6	47	
10	4	5	27	
11	5	8	18	XOR
12	6	7	27	
13	7	8	18	
14	8	9	30	

k	i	j	early start	early finish	late finish	total float	CPM
1	0	1	0	0	0	0	c
2	0	2	0	0	0	0	c
3	0	3	0	0	0	0	c
4	1	5	0	47	74	27	x
5	1	6	0	47	47	0	c
6	2	4	0	47	47	0	c
7	2	7	0	47	74	27	x
8	3	4	0	47	47	0	c
9	3	6	0	47	47	0	c
10	4	5	47	74	74	0	c
11	5	8	74	92	92	0	c
12	6	7	47	74	74	0	c
13	7	8	74	92	92	0	c
14	8	9	92	122	122	0	c

EXECUTION TIME AND COST OF THE PARALLEL IMPLEMENTATION

COST : $124.3600
TIME : 122 ns

After the timing analysis, the non-critical activities are examined to determine the possibility of further optimization. From the CPA result we can see that no further cost optimization can be made since every hardware element is assigned to at least one critical activity.

The parallel implementation provides the lower execution bound, 122ns, and the upper cost bound, $124.36.

6.1.2. Criss.Cross Example

The two extremal implementations of this example are generated by the same procedure as demonstrated in the last example. The implementation results are presented in this subsection. They will be utilized in the next two sections to show how to extract guidance information from them to simplify the equation system and how to use the bounds to prune down the solution space.

The ISPS behavioral description for the example is shown in Figure 6-5 and the hardware elements are shown in Table 6-3.

```
Criss.Cross :=

BEGIN

  ** Carriers **
  t1<0:15>,
  t2<0:15>,
   a<0:15>,
   b<0:15>,

  ** Activity **
  action:=
  ( t1 = a + b next
    t2 = a - b next
    a = t1 + t2 next
    b = t1 - t2 )

END
```

Figure 6-5: Criss.Cross ISPS Behavioral Description

Figure 6-6 is the data flow representation of the design. It has 4 RT level operations (+, +, -, -) and 6 values. The results of the two extremal implementations are show below.

storage	bits	IC's	D_{SS}	D_{SH}	D_{SP}	cost
s_1	<16>	(2)SN74273	20 ns.	0 ns.	27 ns.	$8.10
s_2	<16>	(2)SN74273	20 ns.	0 ns.	27 ns.	$8.10
s_3	<16>	(2)SN74273	20 ns.	0 ns.	27 ns.	$8.10
operator	**bits**	**IC's**	**function**		**D_{FP}**	**cost**
f_1	<16>	(4)SN7483	ADD		70 ns.	$14.20
f_2	<16>	(4)SN7483	ADD		70 ns.	$14.20
f_3	<16>	(2)81LS96 (4)SN7483	ALU		85 ns.	$24.18
f_4	<16>	(2)81LS96 (4)SN7483	ALU		85 ns.	$24.18
f_5	<16>	(4)SN74181	ALU		107 ns.	$19.00
f_6	<16>	(4)SN74181	ALU		107 ns.	$19.00

Table 6-3: Hardware Elements Available for Criss.Cross Example

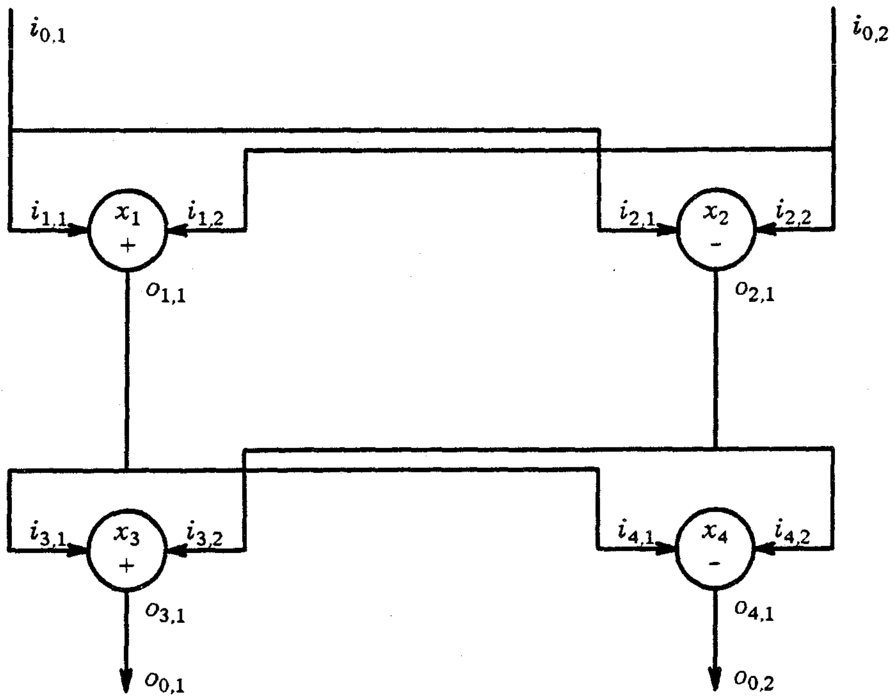

Figure 6-6: Criss.Cross Data Flow Representation

The serial implementation results:

1. storage element assignment:

   s_1 : for $i_{0,1}$, $o_{2,1}$, $o_{4,1}$
   s_2 : for $i_{0,2}$, $o_{3,1}$
   s_3 : for $o_{1,1}$

2. operator assignment:

   f_5 (ALU) for all operations x_1, x_2, x_3 and x_4

3. bounds:

   C_{min} : $43.30
   T_{max} : 663 ns

The parallel implementation results:

1. storage element assignment:

   s_1 : for $i_{0,1}$, $o_{4,1}$
   s_2 : for $i_{0,2}$, $o_{3,1}$

2. operator assignment:

   f_1 : for x_1, f_2 : for x_3
   f_3 : for x_2, f_4 : for x_4

3. bounds:

   C_{max} : $92.96
   T_{min} : 217 ns

The network generated from the data flow of Figure 6-6 is shown in Figure 6-7. The possibility of further cost optimization is shown in the CPA result of Figure 6-8.

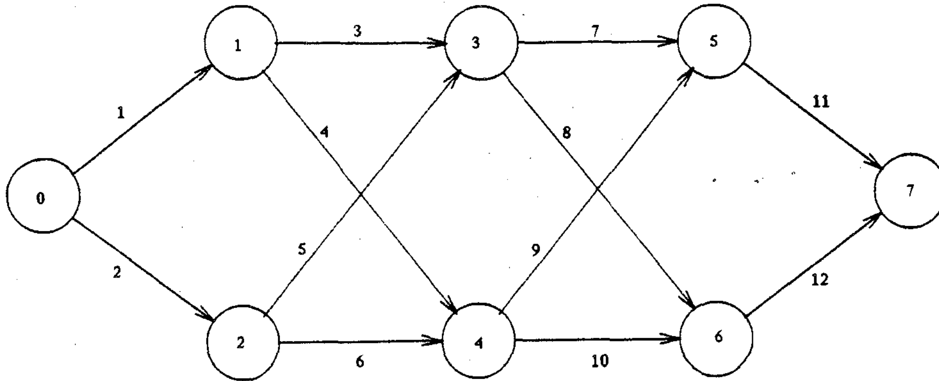

Figure 6-7: The Network For The Data Flow Of Criss.Cross

*** CRITICAL PATH ANALYSIS FOR VT_6 ***

k	i	j	early start	early finish	late finish	total float	CPM
1	0	1	0	0	0	0	c
2	0	2	0	0	0	0	c
3	1	3	0	47	62	15	x
4	1	4	0	47	47	0	c
5	2	3	0	47	62	15	x
6	2	4	0	47	47	0	c
7	3	5	47	117	147	30	x
8	3	6	47	117	132	15	x
9	4	5	47	132	147	15	x
10	4	6	47	132	132	0	c
11	5	7	132	202	217	15	x
12	6	7	132	217	217	0	c

Figure 6-8: CPA Result for Criss.Cross

From the above CPA result, we can see that event 3 (x_1) and event 5 (x_3) could be replaced by a cheaper adder with an extra delay time of 15ns (were such a part available), since they all appear in non-critical activities and the minimum slack is 15ns.

6.2. Computational Improvement Made by Bounds

The two extremal implementations bound the noninferior surface of the solution space by their cost and execution time bounds. Based on these bounds, the designer can make some compromise between cost and performance to achieve the global objective. We have examined different ways to improve the synthesis time by utilizing the bounds. The experimental results presented in the first two subsections show the effects of time bounds and cost bounds. In section 6.2.3, we will compare the computational cost of obtaining the bounds by the heuristic module *vs.* solving the synthesis model with BANDBX.

6.2.1. Effects of Time Bounds

When imposing the time constraint into the synthesis model, we set the objective function to minimize the design cost in order to evaluate the effect of the time bounds. Three examples are represented to examine the synthesis efficiency improvement made by the time bounds.

The first example uses the time bounds to reduce the number of solutions to be explored. By specifying variant values of $T_{IA}(O_0)$ between T_{min} and T_{max} to the synthesis model generator, a subset of noninferior solutions is generated. The designer can choose the best one from them without necessarily generating the complete set of noninferior solutions since the extreme points of the noninferior surface are known. For example, if the design objective of the Criss.Cross example is to obtain a medium speed implementation with reasonably low cost, the designer can specify $T_{IA}(O_0)$ less and equal to 440ns, which is the average time of T_{min} and T_{max}. This

implementation costs \$49.40. The last output value becomes valid at the time $t = 308ns$. If the designer is willing to pay a higher price for a better performance, he can restrict $T_{IA}(O_0) \leq 300ns$. A new implementation is then generated by solving the synthesis model under this higher performance requirement, with an increase in cost to \$54.58 and a decrease in execution time to 269ns.

This example shows that the time bounds narrow the solution space and provide useful guidance for the designer to make proper design decisions. It speeds up the process of generating a desired implementation for a given design with respect to the design objectives.

The second example is an experiment which examines the effect of tightening $T_{IA}(O_0)$ by the upper bound of the execution time, T_{max}, extracted from the serial implementation. Without this upper bound, the designer has to specify an enormous value of $T_{IA}(O_0)$ to avoid the omission of the least cost solution. By replacing the arbitrarily large time bound with the tighter bound T_{max} for both examples, the synthesis time is improved as shown in Table 6-4.

Example	Time Constraint ($T_{IA}(O_0)$)	Synthesis Time (sec.)
Criss.Cross	1000	338.1
Criss.Cross	670	331.0
Logic	1000	21265.0
Logic	250	3896.6

Table 6-4: Effect of Tightening Time Bounds

From Table 6-4, we can see that tightening the time constraint to T_{max} reduces the time for solving the synthesis model to generate the serial implementation.

The third example shows an inconclusive experiment result. The experiment was conducted to examine how TMAX will affect the synthesis time by varying its value. TMAX is a large timing variable used in conjunction with the binary variables to linearize the relations of the synthesis model. It must be greater than the largest value attained by any other timing variables in the model. On the other hand, it should be as small as possible in order to minimize the synthesis time used for linearization. The time bound, T_{max}, obtained from the serial implementation seems to provide the useful guidance to specify such a TMAX to satisfy above two conditions. If we set TMAX slightly larger than T_{max}, ideally the synthesis time should be less than that required by imposing a larger TMAX. But the experimental data presented in Table 6-5 display an inconsistent result. At this point, we cannot draw any conclusion out of it.

Example	TMAX (sec.)	Synthesis Time (sec.)
Criss.Cross	670	630.8
Criss.Cross	1000	861.6
Criss.Cross	2000	1376.8
Criss.Cross	3000	336.1
Criss.Cross	4000	1360.5
Logic	600	3403.9
Logic	1000	4350.9
Logic	2000	3679.9
Logic	3000	4002.3
Logic	4000	4990.0

Table 6-5: TMAX Experimental Results

6.2.2. Effects of Cost Bounds

To evaluate the effects of cost bounds, we add a cost constraint into the synthesis model and change the objective function to minimize the execution time. By making tradeoffs within the cost bounds, C_{min} and C_{max}, a subset of noninferior implementations can be generated. The designer can quickly approach the desired implementation by exploring a smaller set of noninferior solutions.

Using the Criss.Cross example again, by setting the cost constraint to \$55 and \$50 respectively, we obtain the same implementations as those generated by setting $T_{IA}(O_0)$ to 300ns and 440ns. This experiment displays a surprising improvement in synthesis time: by imposing cost bounds on the synthesis model, the synthesis time for the implementations is reduced about 50%, compared with the synthesis time using time bounds. The synthesis results indicate that the cost constraint allows inferior solutions to be pruned earlier and thus reduces the synthesis time for solving the synthesis model by BANDBX.

By setting the cost constraint to C_{max} rather than an arbitrarily large bound, the synthesis time for generating the parallel implementation is consistently reduced for both examples as shown in Table 6-6.

Example	Cost Constraint	Synthesis Time (sec.)
Criss.Cross	\$200	145.2
Criss.Cross	\$95	136.5
Logic	\$200	11307.8
Logic	\$70	11289.1

Table 6-6: Effect of Tightening Cost Bound

6.2.3. Computational Result Comparison for Bound Extraction

Instead of deriving the bounds by the heuristic module, the designer can specify an arbitrarily large output valid time to obtain the minimum cost implementation and then change the objective function to minimize the execution time to generate the parallel implementation. But this is an expensive way to extract bounds. As shown in previous chapters, solving a synthesis model by BANDBX requires considerable computation time. The synthesis time for generating the two extremal implementations for the Logic example by these two methods is given in Table 6-7.

Methodology	Syn. Time for the serial implementation (sec.)	Syn. Time for the parallel implementation (sec.)
Heuristics	1.3	1.0
BANDBX	21265.0	11307.8

Table 6-7: Comparison of Different Bound Extraction ways

Clearly, the heuristic module generates the extremal implementations much faster than BANDBX does.

6.3. Computational Improvement Using S Set Guidance

The guidance information for the S set specification extracted from the two extremal implementations makes a significant improvement in synthesis time, since the size of the synthesis model partly depends on the number of elements in the S set. In this section, we will demonstrate how to utilize the guidance information to minimize the S set, and then present the computational results.

6.3.1. How to Minimize the S Set

When specifying the S set without guidance, each set should include all available registers in order to guarantee optimality for a given design. This will result in a large synthesis model and long synthesis time, since each element in an S set will be present as a binary variable in the synthesis model and will appear in several equations. To improve the synthesis time, we need to minimize the size of each S set while ensuring that it remains suitable for synthesizing all noninferior implementations. The two extremal implementations provide useful guidance for this minimization, using the serial storage element assignment and the network representation.

S set specification

To ensure an optimal implementation for the Criss.Cross design, an easy way to specify each S set is to include all available storage elements capable of storing the value in each set. This introduces more variables than the synthesis process needs for synthesizing noninferior implementations. In a small design such as Criss.Cross, a designer's intuition and experience can be used to restrict the S sets to smaller sets, as, *e.g.*, shown in Figure 6-9. Even with these improved S set specifications, further optimization can still be made by utilizing the guidance information.

The optimization is based on the serial storage element assignment. In Figure 6-10, the leftmost element in each set represents this assignment. With this guidance, we only need to add storage elements to the S sets at the places where a storage usage conflict may occur in some mixed serial/parallel execution sequence. Following

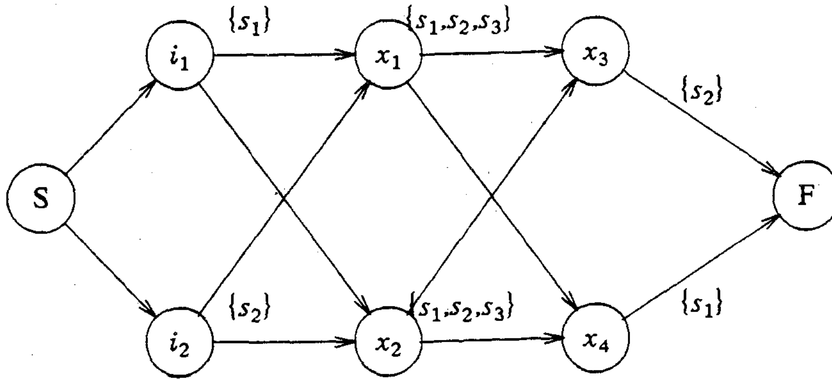

Figure 6-9: Initial S set Specification For Criss.Cross Example

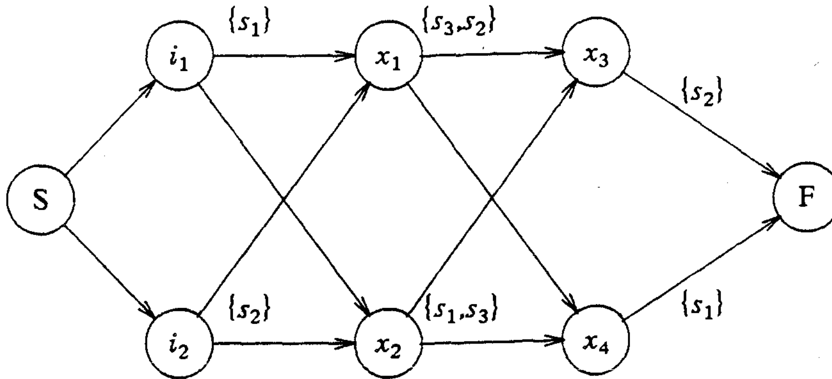

Figure 6-10: Revised S set Specification For Criss.Cross Example

the S set specification procedure described in last chapter, we can examine the operator nodes to find all potential input storage usage conflicts. One conflict is found when examining x_2, as the output of x_2 uses the same storage element, s_1, as one of its inputs, and this input value is also used by x_1. So we add s_3 to the S set for x_2, $S_{2,1}$. Now we need to explore all mixture orderings to find the rest of the conflicts. In Figure 6-10, there are four possible mixed orderings for operator nodes: a) x_1 -> x_2 -> x_3, x_4; b) x_2 -> x_1 -> x_3, x_4; c) x_1, x_2 -> x_3 -> x_4; d) x_1, x_2 -> x_4 -> x_3. Applying case c of step 3 in the S set specification procedure to the second mixed ordering, we find a conflict since the same storage

element, s_3, is used for both inputs of of x_4. So an additional register s_2 has to be added to the set $S_{1,1}$. It is obvious that if we add s_1 instead, the assignment will violate the rule stated in case b of step 3 of the procedure under the last two mixture orderings c) and d). In the current S set specification, as presented in Figure 6-10, no conflict will occur in any mixed ordering.

By specifying the S sets of Figure 6-10 to the synthesis model generator, we can avoid the dilemma of either supplying too large a set of registers, expanding the equation system by unnecessary variables and extending the synthesis time, or eliminating the elements of each S set blindly, potentially sacrificing optimality. This specification uses 2 elements less than the S set specified in Figure 6-9.

Similarly, applying the S set specification procedure to the network and serial S set specification for the Logic example, we obtain the S sets shown in Figure 6-12. This specification uses 3 elements less than the initial S set specification shown in Figure 6-11.

Effect on the size of the synthesis model

By specifying the revised S sets as shown above, we can simplify 14 equations and delete 4 from the hand-written synthesis model for the Criss.Cross example based on the initial S set specification of Figure 6-9; for the Logic example we can simplify 8 equations and delete 18 from the synthesis model based on Figure 6-11. The reduction in size for each model is summarized in Table 6-8.

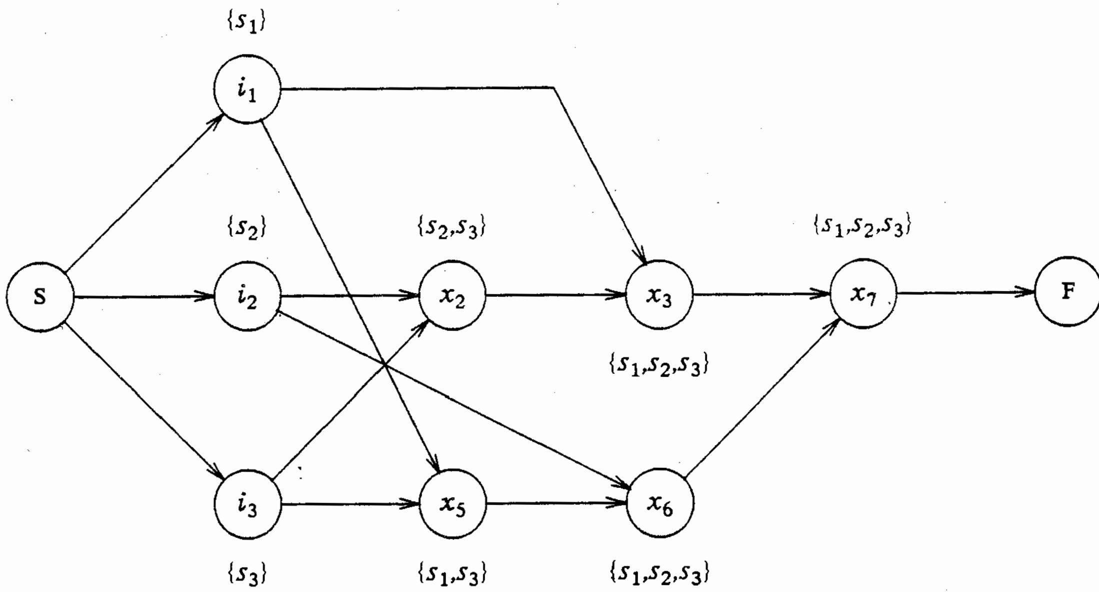

Figure 6-11: Initial S set Specification For Logic Example

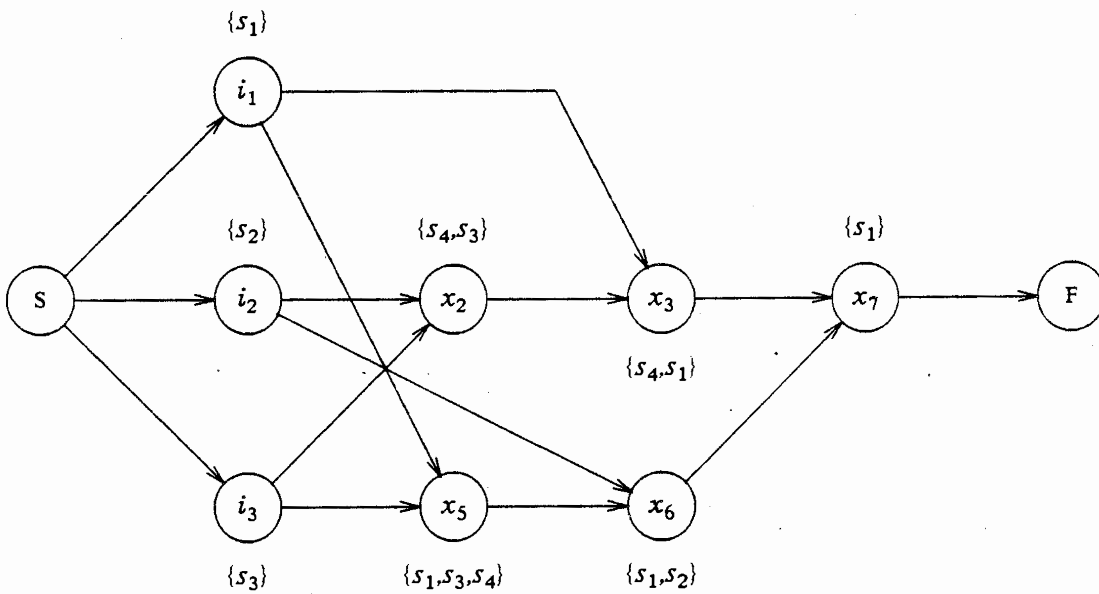

Figure 6-12: Revised S set Specification For Logic Example

Model	Equations	Variables	Model Generation
Criss.Cross	124	87	IDDMA
Criss.Cross	112	72	hand-written
Criss.Cross	108	70	guided
Logic	148	96	IDDMA
Logic	150	85	hand-written
Logic	132	83	guided

Table 6-8: Model Size Comparison

6.3.2. Synthesis Time Improvement

In our experiment, we use BANDBX to solve the equations generated with guidance and without guidance, respectively, and use the timing routine available in the Unix system to measure the synthesis time for each example. To obtain a representative timing measure, each example is run 10 times to obtain an average synthesis time. The synthesis results in Table 6-9 indicate that with guidance the number of subproblems solved during the solution of a problem is less than that without guidance, and hence the optimal solution is found much earlier.

On average, the synthesis time for generating implementations using the guided model is reduced 21% for Criss.Cross, and 79% for Logic, comparing with the synthesis time for solving the hand-written models. Since the Logic example has more options for storage elements, the guidance information is more effective in improving the synthesis time for it than for the Criss.Cross example.

Model	$T_{IA}(O_0)$ (specified)	Optimal Solution Found at Node	Number of Subproblems	Synthesis Time (sec.)
Criss.Cross	440	163	201	1041.91
Criss.Cross(G.)	440	70	159	745.42
Criss.Cross	300	243	260	1105.41
Criss.Cross(G.)	300	53	225	902.35
Logic	500	2930	3208	21265.0
Logic(G.)	500	163	772	4426.3

Table 6-9: Model Synthesis Result Comparison

Chapter 7

OBSERVATIONS AND CONCLUSIONS

This chapter characterizes the heuristic module described in previous chapters and summarizes the work done in this thesis research. During the implementation of the module and the testing of some synthesis examples, several ideas for improving the synthesis time have been tried and a number of experiments have been done to search for promising approaches. The observations are described in section 7.1. Section 7.2 outlines the contributions of the research and its remaining shortcomings. Two directions for further research are suggested in section 7.3.

7.1. Observations

In our experiment, we evaluated the synthesis process for two simple examples (described in last chapter) using hand-written equations. The experimental results reveal that the heuristic module plays an important role in simplifying the equation system and improving the synthesis time. Due to the absence of an automated synthesis model generator in our system, we are unable to complete more complicated design examples, though we can use our heuristic module to generate extremal implementations for them.

The bounds and guidance information extracted from the two extremal implementations improve the synthesis time in two different ways as we have shown.

The bounds limit the solution space and reduce the number of alternative noninferior solutions to be explored, while the guidance information simplifies the synthesis model and reduces the synthesis time for obtaining each noninferior solution. Tightening the cost or time constraints by bounds extracted from the extremal implementations also reduces the synthesis time. The cost constraints seem to have more effect in fathoming subproblems and guiding BANDBX toward the optimal solution.

However, we have to point out that the cost bounds are not precise if the VT representation of a given design consists of more than one VT body. In this case, the registers used for the outputs of a VT body are not used by any other VT bodies since the end of their lifetime is unpredictable due to the isolation of the data relations among VT bodies. When specifying S sets, the number of possible orderings of the nodes grows combinatorially as the number of independent nodes increases. How to apply the guidance locally to restrict the size of data-independent sets is a subject for further study.

The flexibility of the heuristic module can be easily improved by providing different sets of hardware elements (with different cost and speed features) as options to accommodate different logic technologies. This will provide more choices for the designer to decide which technology is best suited to the design.

7.2. Contributions

We have developed and implemented a heuristic module to quickly generate two extremal implementations. Since the module is integrated into the VT translator, no additional data translation is needed to derive the extremal implementations from the initial behavioral specification.

In the research, we have examined different approaches to improve the synthesis time. The major result is that the extremal implementations appear to give a useful set of bounds for the solution space and guidance information for simplifying the synthesis model. The bounds constitute a rather close estimate of the boundary of the noninferior solution space and therefore we can quickly approach the desired solution without generating every point of the solution space. Tightening constraints with the bounds also reduces the synthesis model solution time for generating implementations. The guidance for the S set specification helps a designer to specify a smaller set of storage elements which, in turn, leads the synthesis model generator to simplify the equation system. As a result, the number of subproblems explored by the BANDBX program is reduced and the synthesis time is significantly decreased, compared with the synthesis model solving time without guidance.

There are still some limitations in our module. The storage element assignment has been restricted to local allocation. The cost bound is, therefore, not precise if there is more than one VT body in a design. To obtain more accurate bounds, global data flow analysis is required. In the serial implementation, every value is forced to be stored, while in a practical design interface values might not be stored. This

problem can be solved by taking interface timing constraints into account, *i.e.*, checking the time the inputs will remain available when assigning storage elements for these values.

7.3. Suggestion for Further Research

There are two directions for further improving the synthesis time. The first is to consider global optimization to generate better bounds for the solution space. By global data flow analysis, we can explore all VT bodies to determine the end of the lifetime of every value such that all redundant storage elements can be eliminated. Since most difficulties in synthesizing large designs stem from the large number of variables, constraints and objective functions required for constructing synthesis models, the second direction should steer toward developing more intelligent heuristics to guide the synthesis model generator to efficiently generate a minimum set of equations. In our research we have only dealt with the minimization of storage elements for small examples. We believe that guidance for operator allocation can also be extracted to improve the synthesis time.

References

[Abdel 76] Abdel-Wahab, M.
Scheduling with Application to Register Allocation and Deadlock Problems.
PhD thesis, Dept. of Electrical Engineering, Waterloo University, Waterloo, Ontaric, 1976.

[Christofides 79] Christofides, N., Mingozzi, A., Toth, P., Sandi, C., eds.
Combinatorial Optimization.
John Wiley & Sons, New York, N.Y., 1979.

[Darringer 80] Darringer, J., Joyner, W.
A New Look at Logic Synthesis.
In *17th Design Automation Conference Proceedings*, pages 543-549. ACM SIGDA, IEEE Computer Society-DATC, June, 1980.

[Hafer 81] Hafer, L.
Automated Data-Memory Synthesis : A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic.
PhD thesis, Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., May, 1981.
Also available from the Design Research Center, Carnegie-Mellon University, as Technical Report DRC-02-05-81.

[Martin 78] Martin, C.
BANDBX: An Enumeration Code for Pure and Mixed Zero-One Programming Problems
Industrial and Systems Engineering Dept., Ohio State University, 1978.

[Smith 82] Smith, D.K.
Network Optimisation Practice.
Ellis Horwood Limited, West Sussex, England, 1982.

[Thomas 83] Thomas, D., Hitchcock, C., Kowalski, T., Rajan, J., Walker, R.
Automatic Data Path Synthesis.
Computer 16(12):59-70, December, 1983.

[Thomas 86] Thomas, D.
Automatic Data Path Synthesis.
Advances in CAD for VLSI. Volume 6.*Design Methodologies.*
North-Holland, Amsterdam, 1986, Chapter 13.