Tableau Interpretation for a MILP Problem Using an Interactive Approach

by

Steven F. L. Yap

B.Sc., University of Windsor, 1983

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

SIMON FRASER UNIVERSITY
March 1989

Approval

Name: Steven F.L. Yap

Degree: Master of Science

Title of Thesis: Tableau Interpretation of a MILP Problem Using an Interactive Approach

Examining Commitee:

Chairman: Dr. R. Krishnamurti

Dr. Lou Hafer,
Senior Supervisor

Dr. Pavol Hell,
Supervisory Committee Member

Dr. Slawomir Pilarski,
External Committee Member

March 22, 1989
Date of Approval

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

Tableau Interpretation for a MILP Problem Using an Interactive Approach

Author:

(signature)

STEVEN YAP (INIT: FL)

(name)

17 APRIL 1989

(date)

Abstract

This thesis addresses an efficiency problem in a mixed-integer linear programming (MILP) approach to automated synthesis of RT-level digital logic. The approach to the problem is to add an interactive front-end, the Data Path Synthesis Interactive System (DPSIS), to the MILP package BandBX, so that a human designer can participate in guiding the progress of a branch-and-bound MILP algorithm.

The DPSIS extracts design information from the linear programming (LP) tableaux produced while solving the MILP problem. During the branch-and-bound process, the LP tableaux contain information about partial implementations. The DPSIS translates some of the design information in a tableau from its original mathematical formulation into a form which can be understood by a digital hardware designer (who often knows little or nothing of a constrained optimisation technique such as MILP).

Acknowledgements

I wish to express special appreciation and gratitude to Professor Lou Hafer, my senior supervisor, for his constant guidance, advice, support and availability. Without his support and supervision which I enjoyed during my study at Simon Fraser, the completion of this thesis would not have been possible. His insistence on precision and clarity in writing has been immensely helpful in the preparation of this thesis.

I am also indebted to Dr. Pavol Hell, not only for his reading and commenting on this thesis, but also for his constant advice and encouragement. Thanks are also due to Dr. Slawomir Pilarski, the external examiner of this thesis, for his valuable and stimulating suggestions and comments.

My special thanks to my friend and fellow student Michael Dyck who spent much of his precious time reading my thesis and correcting my grammatical mistakes. I would like to thank Elma Krbavac for giving me various kinds of advice and help.

My thanks go also to many of the graduate students in the School of Computing Science at Simon Fraser, who either read and commented parts of my thesis, or provided me with various kinds of help when they were most needed: Wuyi Wu, Mimi Kao, Frank Tong, Paul Wu, Pattabhiraman, Siu-Cheung Chau, Vincent Ng, Pierre Massi and many others.

Finally, this work is dedicated to my father, my late mother and Ada. They gave me love and support, and encouraged my endeavour to study.

Table of Contents

List of Tables

List of Figures

Chapter 1
Introduction

In [4], Hafer and Parker describe a method for formally modeling digital systems using algebraic relations at the Register-Transfer (RT) level. The model can be viewed as a system of linear constraints with an objective function specified by the designer. It is solved as a mixed-integer linear programming (MILP) problem.

This thesis describes research conducted in an attempt to extract information from the linear programming (LP) tableaux produced while solving the MILP problem - in other words, interpreting the tableau and the branch-and-bound tree maintained by the MILP package. During the branch-and-bound process, the LP tableaux contain information about partial implementations. The intent of the research is to translate information in a tableau from its original mathematical formulation into a form which can be understood by a digital hardware designer (who often knows little or nothing of a constrained optimisation technique such as MILP). Hopefully, this information will help the designer to guide the MILP package to quickly produce the most desirable implementation for a given behavioural specification. The approach to be used involves embedding an interactive module and a tableau interpretation module in the MILP program code. The designer will obtain the information extracted from the tableau through the interactive module.

1.1. Motivation

The general problem faced by a digital designer is to come up with a hardware implementation which is optimal with respect to some set of design goals or objectives. These goals or objectives often conflict with each other. In these situations, access to information about the partially completed implementation throughout the design process is very important, because with this information the designer will be able to select the set of hardware elements for the implementation which embodies the most acceptable tradeoff among the various objectives. Making the

mathematical formulation visible to the designer will be useful because the designer will have the capability to interact with the system at intermediate points during the design process to extract information about the partial solution and suggest design decisions. With this feature we hope to minimise the exploration of unnecessary partial solutions in the branch-and-bound tree maintained by the MILP package. Therefore, it is worth extracting such information and making it understandable to the designer.

At present, the greatest concern is for the development of CAD (Computer Aided Design) systems, covering from device to system levels, for VLSI, which is always increasing in scale and complexity and creating numerous design difficulties. One of the design difficulties is that a vast amount of design information must be kept track of during the design process. Unfortunately, most humans (designers) have difficulty handling large numbers of related details. In this case, some of the design information needed by a designer and already present in the tableau can help the designer if designer interaction is allowed. Therefore, an interactive system for the model mentioned above is very useful because the designer can use the design information already present in the tableaux.

Although strenuous efforts have been made to develop automated logic synthesis programs, CAD designers have always faced a great many difficulties in putting their programs into practical use because the automatically designed results were often of lower quality than manually designed ones. Stand-alone interactive design systems have been introduced in many IC manufacturing plants and laboratories, which have severe requirements for high packing density to hold the fabrication cost down as low as possible. In such situations, however, design remains primarily a manual activity, aided by interactive graphic systems and other bookkeeping tools.

1.2. Related Work

1.2.1. CAD/DA WORK

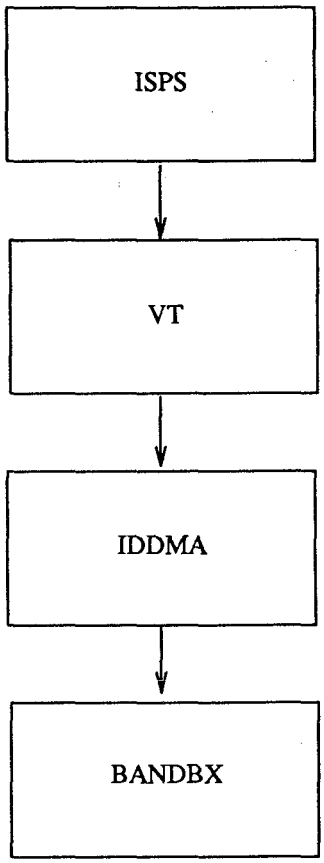

Figure 1-1: The DA system

The synthesis system to be used in this research is shown in Figure 1-1. The Instruction Set Processor Specification (ISPS) translator [1], Value Trace (VT) translator [11], [9] and IDDMA [6], were developed in the context of the Carnegie-Mellon University Design Automation (CMU-DA) project (refer to [2] for further information).

The register-transfer (RT) level logic synthesis method described in [4] expresses the design problem using an algebraic model and uses constrained optimisation techniques to solve the problem. The relations used in the model encompass the behaviour the design must support and the

performance constraints it must satisfy. Binary variables allow the inclusion of implementation decisions. These variables are used to represent the mapping of the operations and values of the data flow representation onto the operators and storage elements which will compose the implementation, and to specify how operation inputs are accessed. The relations are derived from a data flow representation (The Value Trace) [9] that expresses the original RT-level behavioural specification in terms of operations and values. A program, IDDMA, is used to automatically generate the model relations. The relations are then viewed as a system of constraints and solved as a mixed-integer linear programming (MILP) problem to optimise the objective function given by the designer for evaluating candidate implementations. The MILP software used in this research is BandBX [7] [8].

In [10], Prakash investigated the effectiveness of using simplified versions of design paradigms practised by human designers to guide the progress of the branch-and-bound MILP algorithm. Three simplified human design strategies were considered (storage elements first, operators first, and critical path first), as well as several artificial strategies. The choice of strategy was shown to have a significant effect on the solution time.

The approach adopted in [10] used the notion of heuristically capturing design knowledge by using static priorities to fix the order in which decision variables are selected for evaluation. This approach does not take into account the effect of previous design decisions, *i.e.*, it does not refine the importance of the rest of the variables depending on the form of the partial implementation. The idea of capturing design knowledge heuristically leads one to think about the notion of human (designer) interaction with the branch-and-bound MILP program, using the designer as the ultimate heuristic for assigning priorities to decision variables.

1.3. The Problem

From the above discussion, we know that the behavioural specifications for digital hardware designs can be expressed in a mathematical formulation. Unfortunately, such a mathematical formulation is often not transparent to the hardware designer.

In addition to difficulties understanding the mathematical formulation, the designer also has difficulties obtaining design information already present in the tableau.

The research will address these problems by trying to extract design information from the tableau of each partial solution and the branch-and-bound tree maintained by the MILP package.

1.4. The Approach

In order to extract design information from the tableau and branch-and-bound tree for a partial solution, it is necessary to incorporate into BandBX an interactive front-end that first translates between the variable names and concepts used by the designer and the mathematical representation used by BandBX.

Chapter 2 presents a brief discussion of the relationship between the synthesis model, the mathematical model and the designer. Chapter 3 will discuss some of the useful design information we can obtain from the mathematical model. Chapter 4 presents the implementation aspect of the thesis. Chapter 5 describes the interactive system called DPSIS and its commands. Chapter 6 demonstrates with examples how DPSIS helps the designer to guide the design process. Finally, Chapter 7 summarises the results of the thesis and suggests some directions for further research.

Chapter 2

The Synthesis Model and Mixed-Integer Linear Programming

In this chapter, we will present a brief discussion of the synthesis model and the mathematical representation used for linear and mixed-integer linear programming.

2.1. The Synthesis Model: Nonlinear Constraint Forms

This section[1] presents a brief summary of the constraint system for synthesis, using a set of nonlinear constraint forms. These forms are mathematically less tractable than the linear forms described in section 2.2, but it is much easier to see how they relate to the underlying algorithm and implementation. For a more complete exposition, the reader is referred to [4], or Chapter 4 of [3].

2.1.1. The Data Flow Representation and Labelling Convention

For the purpose of generating a system of constraints, a data flow description of the algorithm is used. The particular data flow description is the Value Trace (vt) form developed at Carnegie-Mellon by Snow [12] and augmented by McFarland [9]. Figure 2-1 shows a fragment of data flow with two activities, x_{a1} and x_{a2}, and a flow of data from output $o_{a1,c1}$ of activity x_{a1} to input $i_{a2,c2}$ of activity x_{a2}. This fragment will be used as an aid to explain the derivation of the nonlinear constraint forms.

To refer to the set of all outputs of activity x_a (there may, in general, be more than one) we will use $O_a = \{o_{a,c}\}$. Similarly, $I_a = \{i_{a,c}\}$ will refer to the set of all inputs of x_a.

For completeness, there must also be a convention for representing the outside world, as the logic

[1] Section 2.1 is excerpted from [6] with the permission of the author.

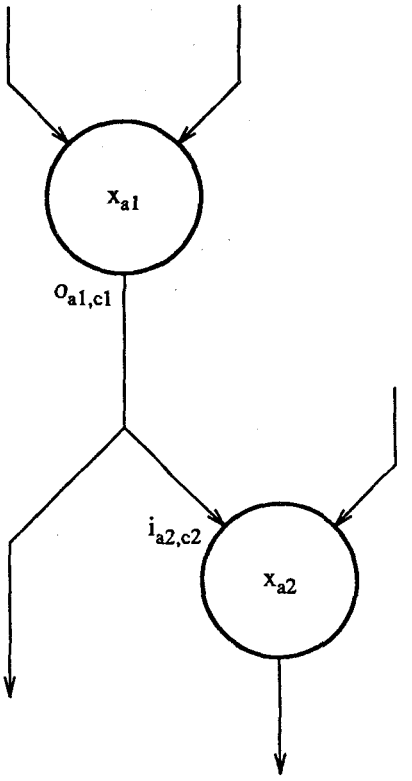

Figure 2-1: An illustrative vt fragment

being synthesized must communicate with it to perform useful work. The set of inputs to the data flow description will be denoted as $I_0=\{i_{0,c}\}$, and the set of outputs as $O_0=\{o_{0,c}\}$. The most consistent view is to look at the outside world as a large activity whose outputs become the inputs to the data flow description, and whose inputs are the values produced as outputs of the data flow. These values will be referred to as external inputs and outputs, to avoid confusion with the inputs and outputs of individual activities.

2.1.2. Hardware Components

It is assumed that there is a given set of operators and storage elements available with which to construct the implementation. Individual operators will be denoted by f_d, individual storage elements by s_e.

A further assumption is that each activity x_a in the data flow has been assigned a set of operators,

F_a, capable of implementing the activity. (*I.e.*, the operators are capable of performing the function required by the activity and also satisfy any other *a priori* constraints.) In general, the sets will not be disjoint; the constraint system ensures a non-conflicting scheduling of activities on operators. Similarly, it is assumed that a set of storage elements, $S_{a,c}$, has been associated with each value $o_{a,c}$.

Operators and storage elements require finite time to perform activities or store values. Table 2-1 describes the time delays which are incorporated in the synthesis model.

$D_{fp}(f_d)$	Propagation delay time of operator f_d from the appearance of the input value(s) at the operator input(s) to the appearance of the output value(s) at the operator output(s).
$D_{ss}(s_e)$	Setup time at the data input of storage element s_e; data at the input to storage element s_e must be valid for at least this long prior to the transition at the clock input of the storage element.
$D_{sh}(s_e)$	Hold time at the data input of storage element s_e; data at the input of storage element s_e must remain valid for at least this long after the transition at the clock input of the storage element.
$D_{sp}(s_e)$	Propagation delay time of storage element s_e from the transition at the clock input to the appearance of the value at the storage element data output.

Table 2-1: Hardware timing values

2.1.3. The Variables

The variables used in the synthesis model can be divided into two classes, continuous variables which represent time, and binary variables which represent design decisions. Table 2-2 describes the continuous variables, and Table 2-3 describes the binary variables.

2.1.4. The Constraint Forms

Roughly speaking, the constraint forms can be divided into three groups: constraints which ensure that components are assigned when needed, constraints which enforce the timing relationships implied by the data flow, and constraints which make sure that no component is scheduled to do more than one thing simultaneously.

$T_{ia}(i_{a,c})$	Time when the value required by input $i_{a,c}$ of activity x_a is available for use in the computation.
$T_{xs}(x_a)$	Time when the computation of activity x_a actually starts.
$T_{oa}(O_a)$	Time when the output values O_a computed by activity x_a are available at the outputs of the operator performing the activity.
$T_{xr}(x_a)$	Time when all output values O_a of activity x_a are no longer required, and thus the time that execution of the activity can cease.
$T_{ir}(I_a)$	Time when the input values for activity x_a are no longer required.
$T_{or}(o_{a,c})$	Time when output value $o_{a,c}$ is no longer required, considering all uses (whether for creating a stored copy of the value, or directly as an input to another activity).
$T_{ss}(o_{a,c})$	Time when the storage element assigned to store a copy of output value $o_{a,c}$ is clocked.
$T_{sa}(o_{a,c})$	Time when the value $o_{a,c}$ is available at the output of the storage element assigned to store it.
$T_{sr}(o_{a,c})$	Time when the stored copy of value $o_{a,c}$ is no longer required as an input to another activity.

Table 2-2: Timing variables for the synthesis model

$\sigma_{d,a}$	Specifies the activity to operator mapping. $\sigma_{d,a}=1$ indicates that operator f_d will implement activity x_a.
$\rho_{e,a,c}$	Specifies the output value to storage element mapping. $\rho_{e,a,c}=1$ indicates that storage element s_e will be used to store output value $o_{a,c}$.
$\gamma_{a,c}$	$\gamma_{a,c}=1$ indicates that a stored copy of output value $o_{a,c}$ exists.
$\delta_{a,c}$	Specifies how input $i_{a,c}$ accesses the output value which is the source for the input. $\delta_{a,c}=1$ indicates input $i_{a,c}$ accesses the stored copy of the value.
$\omega_{a,c}$	Specifies how input $i_{a,c}$ accesses the output value which is the source for the input. $\omega_{a,c}=1$ indicates input $i_{a,c}$ accesses the value directly from the output of the operator producing the value.

Table 2-3: Binary variables for the synthesis model

2.1.4.1. Assigning Components

Each activity x_a must be assigned one and only one operator $f_d \in F_a$ to perform the activity. This can be expressed by a summation over the binary variables $\sigma_{d,a}$:

$$\sum_{\{d | f_d \in F_a\}} \sigma_{d,a} = 1 \tag{2.1}$$

Note that the constraint works because the $\sigma_{d,a}$ are binary variables. At most one of them can take on the value 1, so that the summation is really enforcing a selection of one and only one operator. This notion of "summation as selection" is used frequently in the constraint relations.

For values, the situation is somewhat different - storing a value $o_{a,c}$ is an optional operation, and a storage element $s_e \in S_{a,c}$ is needed only if the value is actually stored. This is expressed by a summation over the variables $\rho_{e,a,c}$ which is set equal to the binary variable $\gamma_{a,c}$ which specifies whether the store actually occurs:

$$\gamma_{a,c} = \sum_{\{e | s_e \in S_{a,c}\}} \rho_{e,a,c} \tag{2.2}$$

It is worth noting that the synthesis constraint system does not enforce an if and only if relation for storage. There is nothing in the constraints to prevent the storage of a value, even though the stored copy is never accessed. An objective function which minimises cost will, however, remove unnecessary storage elements from the implementation.

2.1.4.2. Enforcing Timing Relationships

Referring to Figure 2-1, consider the conditions which must hold at the inputs I_{a2} of activity x_{a2} before execution of the activity can start.

First, a unique source for the input must be selected. The value produced at output $o_{a1,c1}$ can be obtained either directly from the output of the operator which is executing activity x_{a1}, or from a stored copy of the value. A constraint which expresses this is

$$\omega_{a2,c2} + \delta_{a2,c2} = 1 \tag{2.3}$$

Again, the constraint selects one of the two alternatives because the variables are binary.

A further restriction is that the value cannot be obtained from a stored copy if no stored copy exists. The constraint

$$\delta_{a2,c2} \leq \gamma_{a1,c1} \tag{2.4}$$

captures this requirement.

With (2.3) ensuring that one of $\omega_{a2,c2}$ and $\delta_{a2,c2}$ is set to 1 and the other is 0, the time that the value from output $o_{a1,c1}$ will be available at input $i_{a2,c2}$ can be expressed as

$$T_{ia}(i_{a2,c2}) = \omega_{a2,c2} T_{oa}(O_{a1}) + \delta_{a2,c2} T_{sa}(o_{a1,c1}) \tag{2.5}$$

Before the execution of activity x_{a2} can start, all its inputs must be available. This condition is expressed by the constraint

$$T_{xs}(x_{a2}) \geq \underset{\{c2 \mid i_{a2,c2} \in I_{a2}\}}{\mathbf{MAX}} T_{ia}(i_{a2,c2}) \tag{2.6}$$

Once the activity has started, the outputs will be available at a time

$$T_{oa}(O_{a2}) = T_{xs}(x_{a2}) + \sum_{\{d \mid f_d \in F_{a2}\}} \sigma_{d,a2} D_{fp}(f_d) \tag{2.7}$$

The summation essentially selects the proper propagation delay, depending on the operator assigned to implement the activity.

Consider now the lifetime of the output value $o_{a1,c1}$. One possible use of the value is to store it in a storage element, making a stored copy. In this case, the value produced by $o_{a1,c1}$ must satisfy the setup and hold time requirements of the storage element. This is expressed by the constraint

$$T_{or}(o_{a1,c1}) \geq \gamma_{a1,c1}\left(T_{ss}(o_{a1,c1}) + \sum_{\{e \mid s_e \in S_{a1,c1}\}} \rho_{e,a1,c1} D_{sh}(s_e)\right) \tag{2.8}$$

($T_{ss}(o_{a1,c1})$ is constrained by (2.12) to satisfy the setup time, so that only the hold time appears in (2.8).) Note that, in the case that the value is not stored, $\gamma_{a1,c1}=0$ and the constraint reduces to $T_{or}(o_{a1,c1}) \geq 0$, which is trivially satisfied.

The value produced by $o_{a1,c1}$ may also be used directly by the inputs of any number of other activities in the data flow description. A constraint is necessary to ensure that the value lasts as long as it is required at these inputs:

$$T_{or}(o_{a1,c1}) \geq \underset{\{(a2,c2)\,|\,o_{a1,c1}=\mathrm{src}(i_{a2,c2})\}}{\mathbf{MAX}} \omega_{a2,b2} T_{ir}(I_{a2}) \tag{2.9}$$

Note that the set of inputs $i_{a2,c2}$ in (2.9) is the exact same set of inputs which appears in (2.14), but an input only contributes to one of the constraints, because of (2.3).

When all the outputs of an activity are no longer needed, then the execution of the activity can cease:

$$T_{xr}(x_{a1}) \geq \underset{\{(a1,c1)\,|\,o_{a1,c1}\in O_{a1}\}}{\mathbf{MAX}} T_{or}(o_{a1,c1}) \tag{2.10}$$

When the activity is no longer executing, the values at its inputs are no longer needed:

$$T_{ir}(I_{a1}) = T_{xr}(x_{a1}) - \varepsilon \sum_{\{d\,|\,f_d\in F_{a1}\}} \sigma_{d,a1} D_{fp}(f_d) \qquad 0\leq\varepsilon\leq 1 \tag{2.11}$$

The parameter ε allows the constraint to take into account the fact that the outputs of the activity will not go away until some minimum propagation delay has elapsed after the inputs are removed.

To store a value in a storage element, the clock to the storage element must not occur until the value has been present long enough to satisfy the setup time of the storage element. This is expressed by the constraint

$$T_{ss}(o_{a1,c1}) \geq \gamma_{a1,c1}\left(T_{oa}(O_{a1}) + \sum_{\{e\,|\,s_e\in S_{a1,c1}\}} \rho_{e,a1,c1} D_{ss}(s_e)\right) \tag{2.12}$$

(The hold time of the storage element must also be satisfied; this is ensured by (2.8).)

Once the storage element has been clocked, the value will be available at its output at a time

$$T_{sa}(o_{a1,c1}) = T_{ss}(o_{a1,c1}) + \sum_{\{e\,|\,s_e\in S_{a1,c1}\}} \rho_{e,a1,c1} D_{sp}(s_e) \tag{2.13}$$

Finally, when the inputs which are using the stored value no longer need it, the storage element can be freed to store some other value:

$$T_{sr}(o_{a1,c1}) \geq \underset{\{(a2,c2)\,|\,o_{a1,c1}=\mathrm{src}(i_{a2,c2})\}}{\mathbf{MAX}} \delta_{a2,b2} T_{ir}(I_{a2}) \tag{2.14}$$

2.1.4.3. Avoiding Component Usage Conflicts

To make sure that no operator is ever assigned to execute two activities at once, one of two conditions must hold for each unique pair of activities which could use the operator: either the two activities are assigned to different components, or the execution intervals of the two activities do not overlap.

Define an overlap function $\mathbf{L}$ over pairs of closed intervals $[T_1,T_2]$, $T_1 \leq T_2$, and $[T_3,T_4]$, $T_3 \leq T_4$, such that

$$\mathbf{L}([T_1,T_2],[T_3,T_4]) = \begin{cases} 1 & \text{if the intervals overlap} \\ 0 & \text{otherwise} \end{cases}$$

With the overlap function just defined, the constraint

$$\sum_{\{a1,a2\,|\,f_d \in F_{a1}, f_d \in F_{a2}\}} \sigma_{d,a1}\sigma_{d,a2}\mathbf{L}([T_{xs}(x_{a1}),T_{xr}(x_{a1})],[T_{xs}(x_{a2}),T_{xr}(x_{a2})]) = 0 \tag{2.15}$$

ensures that one of the two conditions will hold. There is one term in the summation for each unique pair of potential uses of an operator. If the activities are assigned to different operators, then the product $\sigma_{d,a1}\sigma_{d,a2}$ will be 0, and the execution intervals can overlap. Otherwise, $\sigma_{d,a1}\sigma_{d,a2}$ will be 1, and the execution intervals must be disjoint. Since no term of the sum can be negative, all terms must be 0 for the sum to be 0.

The same reasoning, applied to uses of storage elements to store values, produces the constraint

$$\sum_{\substack{\{(a1,c1),(a2,c2)\,|\\ s_e \in S_{a1,c1},\, s_e \in S_{a2,c2}\}}} \rho_{e,a1,c1}\rho_{e,a2,c2}\,\mathbf{L}\left([T_{ss}(o_{a1,c1}),T_{sr}(o_{a1,c1})],\,[T_{ss}(o_{a2,c2}),T_{sr}(o_{a2,c2})]\right) = 0 \tag{2.16}$$

to prevent the assignment of a storage element to store two values simultaneously. Table 2-4 summarises the nonlinear constraint forms presented in this section.

$$\sum_{\{d \mid f_d \in F_a\}} \sigma_{d,a} = 1 \tag{2.17}$$

$$\gamma_{a,c} = \sum_{\{e \mid s_e \in S_{a,c}\}} \rho_{e,a,c} \tag{2.18}$$

$$\omega_{a,c} + \delta_{a,c} = 1 \tag{2.19}$$

$$\delta_{a2,c2} = \gamma_{a1,c1} \tag{2.20}$$

$$T_{ia}(i_{a2,c2}) = \omega_{a2,c2} T_{oa}(O_{a1}) + \delta_{a2,c2} T_{sa}(o_{a1,c1}) \tag{2.21}$$

$$T_{xs}(x_a) \geq \underset{\{c \mid i_{a,c} \in I_a\}}{\mathbf{MAX}} T_{ia}(i_{a,c}) \tag{2.22}$$

$$T_{oa}(O_a) = T_{xs}(x_a) + \sum_{\{d \mid f_d \in F_a\}} \sigma_{d,a} D_{fp}(f_d) \tag{2.23}$$

$$T_{or}(o_{a,c}) \geq \gamma_{a,c} \left(T_{ss}(o_{a,c}) + \sum_{\{e \mid s_e \in S_{a,c}\}} \rho_{e,a,c} D_{sh}(s_e) \right) \tag{2.24}$$

$$T_{or}(o_{a1,c1}) \geq \underset{\{(a2,c2) \mid o_{a1,c1} = \mathrm{src}(i_{a2,c2})\}}{\mathbf{MAX}} \omega_{a2,b2} T_{ir}(I_{a2}) \tag{2.25}$$

$$T_{xr}(x_a) \geq \underset{\{(a,c) \mid o_{a,c} \in O_a\}}{\mathbf{MAX}} T_{or}(o_{a,c}) \tag{2.26}$$

$$T_{ir}(I_a) = T_{xr}(x_a) - (1-\varepsilon) \sum_{\{d \mid f_d \in F_a\}} \sigma_{d,a} D_{fp}(f_d) \qquad 0 \leq \varepsilon \leq 1 \tag{2.27}$$

$$T_{ss}(o_{a,c}) \geq \gamma_{a,c} \left(T_{oa}(O_a) + \sum_{\{e \mid s_e \in S_{a,c}\}} \rho_{e,a,c} D_{ss}(s_e) \right) \tag{2.28}$$

Table 2-4: Nonlinear constraint forms

$$T_{sa}(o_{a,c}) = T_{ss}(o_{a,c}) + \sum_{\{e \mid s_e \in S_{a,c}\}} \rho_{e,a,c} D_{sp}(s_e) \tag{2.29}$$

$$T_{sr}(o_{a1,c1}) \geq \underset{\{(a2,c2) \mid o_{a1,c1} = \mathrm{src}(i_{a2,c2})\}}{\mathbf{MAX}} \delta_{a2,b2} T_{ir}(I_{a2}) \tag{2.30}$$

$$\sum_{\{a1,a2 \mid f_d \in F_{a1},\, f_d \in F_{a2}\}} \sigma_{d,a1}\sigma_{d,a2}\, \mathbf{L}\left([T_{xs}(x_{a1}), T_{xr}(x_{a1})],\, [T_{xs}(x_{a2}), T_{xr}(x_{a2})] \right) = 0 \tag{2.31}$$

$$\sum_{\substack{\{(a1,c1),(a2,c2) \mid \\ s_e \in S_{a1,c1},\, s_e \in S_{a2,c2}\}}} \rho_{e,a1,c1}\rho_{e,a2,c2}\, \mathbf{L}\left([T_{ss}(o_{a1,c1}), T_{sr}(o_{a1,c1})],\, [T_{ss}(o_{a2,c2}), T_{sr}(o_{a2,c2})] \right) = 0 \tag{2.32}$$

Table 2-4, continued

2.2. Linearising the Model Relations

The constraints outlined in Section 2.1 must be linearised before they can be solved as a MILP problem. In [6], Hafer gives the necessary modifications to linearise the relations shown in Table 2-4. For the purpose of linearising we will assume that, for any given synthesis problem, we can determine a time, $\hat{T}$, which is greater than the largest value attained by any of the timing variables in the model. To illustrate the procedure, we will use equation (2.31) again, which is written to prevent usage overlaps for an operator f_d. Since $\sigma_{d,a1}$, $\sigma_{d,a2}$, and the function **L** can take on only positive values, it is equivalent to write

$$\sigma_{d,a1}\sigma_{d,a2}\mathbf{L}\left([T_{xs}(x_{a1}), T_{xr}(x_{a1})], [T_{xs}(x_{a2}), T_{xr}(x_{a2})]\right) = 0 \tag{2.33}$$

for each term of the summation. The development of linear forms for constraints (2.33) is quite complex. To motivate the development, it is helpful to restate the constraint as follows:

$$\text{if } \sigma_{d,a1}=1 \text{ and } \sigma_{d,a2}=1 \;\Rightarrow\; T_{xs}(x_{a2}) \geq T_{xr}(x_{a1}) \text{ or } T_{xs}(x_{a1}) \geq T_{xr}(x_{a2}) \tag{2.34}$$

If the binary variable $\alpha_{a1,a2}$ is introduced and defined so that

$$\text{if } T_{xs}(x_{a2}) \geq T_{xs}(x_{a1}) \;\Rightarrow\; \alpha_{a1,a2}=0 \tag{2.35}$$

(2.34) can be restated as

$$\text{if } \sigma_{d,a1}=1 \text{ and } \sigma_{d,a2}=1 \text{ and } \alpha_{a1,a2}=0 \Rightarrow T_{xs}(x_{a2}) \geq T_{xr}(x_{a1}) \tag{2.36}$$

$$\text{if } \sigma_{d,a1}=1 \text{ and } \sigma_{d,a2}=1 \text{ and } \alpha_{a1,a2}=1 \Rightarrow T_{xs}(x_{a1}) \geq T_{xr}(x_{a2})$$

The linear forms for (2.36) are

$$\begin{aligned} &T_{xs}(x_{a2})-T_{xr}(x_{a1})-\hat{T}\sigma_{d,a1}-\hat{T}\sigma_{d,a2}+\hat{T}\alpha_{a1,a2} \geq -2\hat{T} \\ &T_{xs}(x_{a2})-T_{xr}(x_{a1})-\hat{T}\sigma_{d,a1}-\hat{T}\sigma_{d,a2}-\hat{T}\alpha_{a1,a2} \geq -3\hat{T} \end{aligned} \tag{2.37}$$

The linear constraint forms for the relations presented in Table 2-4 are shown in Table 2-5.

To indicate the presence or absence of a particular hardware component in an implementation, two more types of binary variables are required. Let β_d represent the presence ($\beta_d = 1$) or absence ($\beta_d = 0$) of operator f_d. Similarly, χ_e represents the presence or absence of storage element s_e. For each activity x_a which could potentially use f_d, there is a constraint,

$$\sum_{\{d \mid f_d \in F_a\}} \sigma_{d,a} \leq N\beta_d$$

where N is the number of variables in the summation. The corresponding constraint form for storage elements is

$$\sum_{\{e \mid s_e \in S_{a,c}\}} \rho_{e,a,c} \leq N\chi_e$$

The variables β_d and χ_e are used to construct objective functions which include the cost of the implementation.

2.3. Linear Programming and Mixed-Integer Linear Programming

This section discusses models in the context of constrained optimisation, and explains the notation used in the thesis for LP and MILP problems. It is assumed that the reader is familiar with the theory of linear programming.

When a technical person discusses a model of a situation which is being studied, he/she is referring to an idealized representation of an actual system. Typically, mathematical models for

$$\sum_{\{d|f_d \in F_a\}} \sigma_{d,a} = 1 \tag{2.38}$$

$$\sum_{\{e|s_e \in S_{a,c}\}} \rho_{e,a,c} \leq 1 \tag{2.39}$$

$$\delta_{a2,c2} - \sum_{\{e|s_e \in S_{a1,c1}\}} \rho_{e,a1,c1} \leq 0 \tag{2.40}$$

$$T_{ia}(I_{a2}) - T_{oa}(O_{a1}) \geq 0 \tag{2.41}a$$

$$T_{ia}(I_{a2}) - T_{sa}(o_{a1,c1}) - \hat{T}\delta_{a2,c2} \geq -\hat{T} \tag{2.41}b$$

$$T_{xs}(x_a) - T_{ia}(I_a) \geq 0 \tag{2.42}$$

$$T_{or}(o_{a,c}) - T_{ss}(o_{a,c}) - \sum_{\{e|s_e \in S_{a,c}\}} (D_{sh}(s_e) + \hat{T})\rho_{e,a,c} \geq -\hat{T} \tag{2.43}a$$

$$T_{or}(o_{a1,c1}) - T_{ir}(x_{a2}) + \hat{T}\delta_{a2,c2} \geq 0 \tag{2.43}b$$

$$T_{xr}(x_a) - T_{or}(o_{a,c}) \geq 0 \tag{2.44}$$

$$T_{ss}(o_{a,c}) - T_{oa}(O_a) - \sum_{\{e|s_e \in S_{a,c}\}} (D_{ss}(s_e) + \hat{T})\rho_{e,a,c} \geq -\hat{T} \tag{2.45}$$

$$T_{sr}(o_{a1,c1}) - T_{ir}(x_{a2}) - \hat{T}\delta_{a2,c2} - \sum_{\{e|s_e \in S_{a,c}\}} \hat{T}\rho_{e,a,c} \geq -2\hat{T} \tag{2.46}$$

$$T_{xs}(x_{a2}) - T_{xr}(x_{a1}) - \hat{T}\sigma_{d,a1} - \hat{T}\sigma_{d,a2} + \hat{T}\alpha_{a1,a2} \geq -2\hat{T} \tag{2.47}a$$

$$T_{xs}(x_{a2}) - T_{xr}(x_{a1}) - \hat{T}\sigma_{d,a1} - \hat{T}\sigma_{d,a2} - \hat{T}\alpha_{a1,a2} \geq -3\hat{T}$$

$$T_{xs}(x_{a2}) - T_{xr}(x_{a1}) - \hat{T}\sigma_{d,a1} - \hat{T}\sigma_{d,a2} \geq -2\hat{T} \tag{2.47}b$$

$$\sigma_{d,a1} + \sigma_{d,a2} \leq 1 \tag{2.47}c$$

Table 2-5: Linear constraint forms

$$T_{ss}(o_{a2,c2}) - T_{sr}(o_{a1,c1}) - \hat{T}\rho_{e,a1,c1} - \hat{T}\rho_{e,a2,c2} + \hat{T}\alpha_{a1,c1,a2,c2} \geq -2\hat{T} \quad (2.48)a$$

$$T_{ss}(o_{a2,c2}) - T_{sr}(o_{a1,c1}) - \hat{T}\rho_{e,a1,c1} - \hat{T}\rho_{e,a2,c2} - \hat{T}\alpha_{a1,c1,a2,c2} \geq -3\hat{T}$$

$$T_{ss}(o_{a2,c2}) - T_{sr}(o_{a1,c1}) - \hat{T}\rho_{e,a1,c1} - \hat{T}\rho_{e,a2,c2} \geq -2\hat{T} \quad (2.48)b$$

$$\rho_{e,a1,c1} + \rho_{e,a2,c2} \leq 1 \quad (2.48)c$$

Table 2-5, continued

constrained optimisation are structured to include four components: variables, parameters, constraints, and an objective function.

The variables in a model represent the decision alternatives or items which can be varied in the real-life situation. Typically, we are seeking values for these variables which are feasible with respect to the system of constraints, and optimal with respect to the objective function. In the synthesis model described in Section 2.1, the variables represent design choices and execution timings.

Parameters are inputs which may or may not be adjustable, but are known either exactly or approximately. In the synthesis model, propagation delays for components, or interface timing specifications, represent fixed parameters. Performance or cost limitations could be either fixed or adjustable, depending on the application.

Constraints are the conditions which limit the values that the variables can assume. They are derived from the physical properties of the system being modelled. Some constraints correspond to global properties. For synthesis, variables which represent decisions are restricted to values of 0 or 1, corresponding to the yes-no nature of a decision, while variables which represent time are constrained to be positive. Other constraints represent the interrelations between individual elements of the system. The constraints described in Section 2.1 fall into this class.

The objective function measures the effectiveness of a solution as a function of the variables in the model. It provides a goal for the optimisation procedure. In working with the synthesis model, the cost and execution time of an implementation will be the objectives that are considered.

The area of *mathematical programming* plays a prominent role in our research. It consists of a variety of techniques and algorithms for solving certain kinds of mathematical models. Mathematical programming has as its goal the solution of the model by finding values of the variables which maximise or minimise the objective function subject to a system of inequality and equality constraints. Mathematical programming is divided into several areas depending on the nature of the constraints, the objective function, and the variables. Linear programming (LP) deals with those models in which the constraints and the objective are linear expressions in the variables. Integer linear programming (ILP) deals with the special case in which the variables are constrained to take integer values. As we have discussed, the mathematical programming model used in this research is MILP, where some of the variables are restricted to be integral and others may take on any real value. In the particular case of the synthesis model used in this thesis, the integer variables are restricted to 0 or 1, and the other variables must be positive real values. In the next few sections, we will examine the relationship between the synthesis model and the MILP mathematical programming model in more detail.

2.3.1. MILP Models

In this section, we will explain how to set up a MILP mathematical model of our problem.

Let

Π denote the set of decision variables,
Φ denote the set of continuous variables.

Then the MILP model can be stated as follows:

$$\text{maximise } c_1x_1 + c_2x_2 + \ldots\ldots + c_nx_n$$

subject to constraints

$$a_{11}x_1 + a_{12}x_2 + \ldots\ldots + a_{1n}x_n \leq (=)\, b_1$$

$$a_{21}x_1 + a_{22}x_2 + \ldots\ldots + a_{2n}x_n \leq (=)\, b_2$$

•

•

$$a_{m1}x_1 + a_{m2}x_2 + \ldots\ldots + a_{mn}x_n \leq (=)\, b_m \qquad (2.49)$$

$$x_i = 1 \text{ or } 0, \text{ for } x_i \in \Pi \text{ and}$$

$$x_i \geq 0, \text{ for } x_i \in \Phi.$$

(2.49) is called the **standard** form. The **linear relaxation** of (2.49) retains the same constraints, with the exception that the condition of integrality for the variables Π is relaxed to $0 \leq x_i \leq 1$, $x_i \in \Pi$.

Many LP algorithms require the constraint system to be stated in **canonical** form. This requires that all constraints be converted to equality constraints. Consider the constraint

$$a_{i1}x_1 + a_{i2}x_2 + \ldots\ldots + a_{in}x_n \leq b_i$$

We may convert an inequality constraint to an equality constraint by introducing a new variable x_{n+i} and writing

$$a_{i1}x_1 + a_{i2}x_2 + \ldots\ldots + a_{in}x_n + x_{n+i} = b_i$$

The variable $x_{n+i} \geq 0$ is called a **slack variable** because it "takes up the slack" between the left side of the constraint and the right side.

Corresponding to the way most mathematical programming algorithms are presented, we will use the abbreviated **Tucker** tableau in this thesis. The tableau can be stated as follows:

$$x_{B0} = b_{00j} - a_{01j}x_{1j} - a_{02j}x_{2j} - \ldots\ldots - a_{0nj}x_{nj}$$

$$x_{B1} = b_{10j} - a_{11j}x_{1j} - a_{12j}x_{2j} - \ldots\ldots - a_{1nj}x_{nj}$$
$$x_{B2} = b_{20j} - a_{21j}x_{1j} - a_{22j}x_{2j} - \ldots\ldots - a_{2nj}x_{nj}$$

•

•

$$x_{Bm} = b_{m0j} - a_{m1j}x_{1j} - a_{m2j}x_{2j} - \ldots\ldots - a_{mnj}x_{nj} \qquad (2.50)$$

The variables x_{Bi} on the left side of the tableau are the **basic variables** and the constant values b_{i0j} are the **basic variable values.** The subscript i indicates the *ith* row of the tableau. The subscript j indicates the final tableau of subproblem j. The variables on the right side of the tableau are the **nonbasic variables**, each of which has the value of 0.

(2.50) can be expressed in matrix and vector form as

$$\mathbf{x_{basic}} = \mathbf{b} - \mathbf{A}\mathbf{x_{non\text{-}basic}} \qquad (2.51)$$

where A is the coefficient matrix, $\mathbf{x_{non\text{-}basic}}$ is the vector of non-basic variables, **b** is the vector of basic variable values, and $\mathbf{x_{basic}}$ is the vector of basic variables.

2.3.2. The Concept of Branch and Bound in MILP

This section describes the branch-and-bound technique in MILP. Branch-and-bound is an optimisation technique that uses binary tree enumeration. It involves calculating *upper bounds* and *lower bounds* on the objective function, in order to accelerate the process of reaching the optimal solution and thereby curtail the enumeration. The technique starts by solving the LP relaxation of the problem (*i.e.*, solving the same constraint system, but without the integrality constraints). It then selects a non-integer binary variable[2] $x_i \in \Pi$ and fixes it at values of 0 or 1, thereby creating two subproblems. Thus a sequence of subproblems is considered. Eventually, no further exploration from a subproblem is possible. Such a subproblem is said to be *fathomed.* Subproblems can be fathomed due to infeasibility, integrality, or by bounds.

---

[2]The phrase "a non-integer binary variable", while something of an oxymoron, is more convenient than "a binary variable which has a fractional value in the optimal solution to the linear relaxation of the current subproblem".

If a subproblem is fathomed due to *integrality*, the subproblem has reached an integer solution and no further exploration is necessary.

If a subproblem is fathomed due to *infeasibility*, there is no solution of the relaxation problem which satisfies all the constraints.

If a subproblem is fathomed by *bounds*, it means that the current bound on the objective function for the subproblem is less than or equal to the value of the objective function for the best integer solution already know (assuming that we are maximising the objective function).

If the subproblem is not fathomed, one or more non-integer binary variables are selected and restricted to have values of 0 or 1. Such variables are appropriately called *fixed* or *specified variables*. Similarly, variables which are not specified are called *free* or *unspecified variables*.

Each node v_j in the binary tree represents a subproblem; node v_0 (*i.e.*, the root) represents the original problem. Each edge fixes a binary variable at 0 or 1, and the path v_0 to v_j denoted by P_j represents a set of specified variables which describes the subproblem v_j. v_j will be referred to as a partial solution of the design.

At this point, we going to define some notation that will be used in the rest of this thesis.
Let

- T_j denote the final tableau resulting from the solution of the linear relaxation of subproblem v_j.
- B_j denote the set of basic variables in T_j.
- NB_j denote the set of nonbasic variables in T_j.
- BT_j denote the set of timing variables in B_j.
- BD_j denote the set of decision variables in B_j.
- Π_j denote the set of decision variables that are fixed along the path P_j.

2.4. The BandBX Branch-and-Bound Algorithm

2.4.1. Upward and Downward Penalties

After the linear relaxation problem defined at a given subproblem has been solved, the solution is checked for possible fathoming. If the subproblem still cannot be fathomed, the question of which non-integer binary variable should be selected to branch is still open. This problem will be considered after we show how to calculate a better bound. From (2.50), the value of each basic variable in the optimal solution to the linear relaxation at subproblem v_j is

$$x_{Bi} = b_{i0j} - \sum_k a_{ikj} x_{kj}, \quad x_{kj} \in NB_j$$

Since v_j was not fathomed due to integrality, there are non-integer binary variables among the basic variables. Since these variables must be forced to 0 or 1 to satisfy the integrality constraints, it is useful to calculate the penalty to the objective function which results when this is done.

For each non-integer binary variable, the downward penalty D_i which results when the variable is forced to 1 is defined as

$$D_i = \underset{k}{\text{MIN}} \frac{b_{i0j} a_{0kj}}{a_{ikj}}, \quad a_{ikj} > 0$$

D_i is a lower bound on the deterioration of the objective function value which will result if x_{Bi} is forced to 0. Similarly,

$$U_i = \underset{k}{\text{MIN}} \frac{(b_{i0j} - 1) a_{0kj}}{a_{ikj}}, \quad a_{ikj} < 0$$

is defined to be the upward penalty which results when x_{Bi} is forced to 1.

Now, if the subproblem still cannot be fathomed by bounds after applying the penalties, then the algorithm proceeds to to either fix all monotone variables (non-integer basic binary variables that can assume only one value in successor subproblems due to penalties) or, if no monotone variables are present, to branch on a non-integer basic binary variable. The choice of the branching variable and the direction is made by selecting the largest among all up and down penalties and branching on the corresponding variable in the direction opposite that yielding the maximum penalty.

2.4.2. Selecting a New Subproblem

When a subproblem is fathomed, backtracking to retrieve a subproblem is flexible and the choice is made according to two options. The first option is to choose the subproblem with the most promising bound (*i.e.*, the largest bound on the objective function). The second option is to use the best projection criterion based on the sum of integer infeasibilities, defined as

$$\sum_{i \in BD_j} min\ \{b_{i0j}, 1-b_{i0j}\}$$

Thus if we let s_j represent the sum of integer infeasibilities for the subproblem v_j, the subproblem selected by this option is the one yielding the largest value of E_j where

$$E_j = z_j - \frac{z_{LP} - z_s}{s_1} s_j$$

In the above expression, z_j is the value of the objective function for subproblem v_j and z_s is the value of the current incumbent zero-one solution, unless there is no incumbent, in which case z_s is the smallest objective function value encountered among feasible subproblems. z_{LP} is the objective function value for the solution to the linear programming relaxation of the original problem, and s_1 is its integer infeasibility.

2.4.3. Monotone Variables

Let

$$suf_p = z_j - z_s$$

be the difference between the objective function value for the current incumbent solution and the objective function value for the linear relaxation at subproblem v_j. If $D_i \geq suf_p$ then x_{Bi} is a *monotone increasing* variable. It must be set to 1; setting it to 0 will cause the value of the objective function to become worse than the objective function value for the current incumbent solution. Similarly, if $U_i \geq suf_p$ then x_{Bi} is a *monotone decreasing* variable and must be set to 0.

2.4.4. Branching Rules

After the linear relaxation problem defined for a given subproblem has been solved, the solution is checked for possible fathoming by infeasibility, bounds, and integrality. If the subproblem still cannot be fathomed, the algorithm proceeds to either fix all the monotone variables or, if no monotone variables are present, to branch on a non-integer basic zero-one variable.

The choice of branching variable and direction is made by selecting the largest among all up and down penalties and branching on the corresponding variable in the direction opposite that yielding the maximum penalty. Let

$$pen_{max} = \underset{i}{\mathbf{MAX}}\ (D_i, U_i)$$

be the maximum penalty over all non-integer binary variables in the solution to the linear relaxation.

If pen_{max} is less than suf_p and the value of pen_{max} is from an upward penalty (downward penalty) then the selected binary variable is fixed to 0 (1) and the algorithm stores the subproblem in which the variable is fixed to 1 (0).

2.4.5. Fixing Variables

Zero-one variables that are to be fixed at 0 or 1, either by branching or by being identified as monotone variables, are handled by assigning them an upper bound of 0 which forces them to remain at 0 in successor problems. The mechanism is straightforward for variables which are to be fixed at 0. Variables which are to be fixed at one are *complemented* by rewriting the constraints in which they occur and adding a correction to the value of the objective function.

To complement a variable x_{kj}, let

$$x_{kj} = 1 - x'_{kj}$$

where x'_{kj} is the complementing variable of x_{kj}. Now, consider a constraint i such that

$$a_{i1j}x_{1j} + a_{i2j}x_{2j} + \ldots\ldots + a_{inj}x_{nj} = b_{i0j} \qquad (2.52)$$

Suppose x_{1j} in the above constraint is to be fixed at 1. Constraint (2.52) is modified to

$$-a_{i1j}x'_{1j} + a_{i2j}x_{2j} + ... + a_{inj}x_{nj} = b_{i0j} - a_{i1j} \tag{2.53}$$

and the new variable x'_{kj} is fixed at 0 by giving it an upper bound of 0. The net effect is the same as fixing the original variable x_{1j} at 1.

2.4.6. The Branch-and-Bound Algorithm

In this section, we will discuss the technique used by BandBX. BandBX uses a branch-and-bound algorithm that involves the following major steps:

1. Solve the linear programming relaxation problem (LP) of the mixed-integer linear programming problem (MILP) to provide a bound on the objective.

2. Try to fathom the current subproblem by using bounds on the value of the objective function, infeasibility, and integrality. If a subproblem is fathomed then go to step 5.

3. Try rounding tests if the LP problem remains unfathomed. These tests are optional. The system will do natural rounding first (*i.e.*, rounding all non-integer binary variables to the nearest integer value). If this does not produce a feasible zero-one solution, then directed rounding is attempted. By directed rounding we mean rounding all non-integer binary variables in one direction (*i.e.*, either to one or to zero). If either type of rounding produces a better incumbent zero-one solution the present integer solution will be replaced. Otherwise the system will continue solving the subproblem.

4. If there are no monotone variables[3], the system will pick a non-integer binary variable and force it to 1 or 0, creating two subproblems, and go to step 1. One problem is stored and the other becomes the active problem in step 1. Otherwise the system will force the monotone variables and go to step 1.

---

[3]Non-integer variables for which only one value (0 or 1) is possible. Monotone variables are identified based on penalty and feasibility tests.

5. There are two options to select a subproblem when the current subproblem has been fathomed. The first option is to choose the subproblem with the largest bound on the objective function. The second is to use the best projection criterion based on the sum of integer infeasibilities. If there are no subproblems remaining then STOP otherwise select a new subproblem and go to step 1.

Chapter 3
Useful Information

The main reason why we need an interactive system is to cut down the amount of computation. With the branch-and-bound method, if no clever heuristic or guideline is provided, the complete tree search has a size exponential in the number of decision variables. The size will become prohibitive once we have 20 or more decision variables. However, if we have a good heuristic, which in our case will come from the designer through interaction, we can greatly reduce the amount of computation because of quick fathoms or pruning of the branch-and-bound tree. In order to supply good guidelines, the user has to know what is going on in the middle of the computation.

In this chapter we will discuss the available design information and suggest ways to interpret useful design information from the branch-and-bound tree and the partial solution tableau. We will explain the information in terms of the continuous variables (or timing variables) which represent time, and the binary variables (or decision variables) which represent design decisions.

In the next chapter, we shall describe the interactive system, including the queries and commands a user can submit to find out information in the middle of the branch-and-bound process.

3.1. BandBX Technique versus Designer Technique

Before we discuss how to interpret design information from the tableau and the branch-and-bound tree, it is useful to describe the relationship between the branch-and-bound technique used in BandBX (described in section 2.4) and the human designer. This, in turn, will show why designer interaction is appropriate in certain steps of the branch-and-bound process.

The partial solution to the MILP problem is analogous to a partially completed implementation of

a design. When a human designer is at a certain stage of designing an implementation, he/she has to make various design decisions based on design information that he/she knows, such as :

- Whether storage elements and operators have been assigned.
- The propagation delay time of operators and storage elements.
- The setup time and hold time of storage elements.
- The time constraints between two operators that have been assigned in the partial design.
- Whether the active periods of operators are independent or overlap each other.
- Whether operators and storage elements can be released after a certain time (*i.e.*, trying to reuse components that have been released).

An examination of step 2 and step 3 of the branch-and-bound algorithm reveals that it searches for a better solution by computing the penalties and doing the rounding tests using the numerical values present in the tableau (*i.e.*, the objective-function coefficients, the technological coefficients and the solution column). As for a human designer, she/he knows certain design information (as mentioned above) and based on this information, she/he tries to improve the partially designed circuit. There are certain similarities betweens step 2 and 3 of the branch-and-bound algorithm and the stage where a designer is trying to improve a partially designed circuit, and these provide the motivation for this research. It will examine how the information extracted from the tableau could help a designer gain a better understanding of a partially designed circuit. On the other hand, it will also examine how the knowledge that expert designers use to complete a partial design could help to guide the branch-and-bound search.

3.2. Decision Variables

Decision information is contained in the subproblem's final tableau T_j and its place in the binary branch-and-bound tree. In this thesis, branch-and-bound trees are binary, because we're using binary integer variables. But also note that this is not always so, nor is material in the thesis restricted to this case. Each node v_j of the binary tree represents a different subproblem. Using the notation given in the previous chapter, this section explains the design information we can obtain.

There are three types of subproblem found at the leaves of the branch-and-bound tree. The first type are subproblems that were fathomed due to integrality. These are feasible, complete designs, and one of them is the current best solution. The second type are subproblems that were fathomed due to bounds or infeasibility. Such subproblems will either lead to no solution or will never lead to a solution better than the current best solution. The third type are the active subproblems (*i.e.*, subproblems that can possibly lead to a better solution than the current best solution).

With each subproblem, the user will be able to look at the the MILP information, which includes the branch-and-bound information such as the current bound on the value of the objective function and the current values of the decision variables and timing variables. The order with which registers and operators were allocated for the design can be obtained by looking at the decision variables that have been fixed along the path from the root of the branch-and-bound tree to the subproblem.

As described in section 2.3.2, decision variables can be fixed or free in a partial solution. In each case, the decision variables give different design information. The allocation of registers to output values, and operators to operations, is specified by the values of the $\rho_{e,a,c}$ and $\sigma_{d,a}$ decision variables. If the variable $\sigma_{d,a}$ has a value of 1, it indicates that operator f_d will be used to implement activity x_a in the current partial design; if it has a value 0, it indicates operator f_d will not be used to implement (or operator f_d is *excluded* from implementing) activity x_a. In either case, if $\sigma_{d,a}$ is not fixed, then the decision might change in subsequent partial implementations developed from the current one. But if $\sigma_{d,a}$ is fixed, the decision is permanent and will be present in every implementation derived from the current one. If the variable $\sigma_{d,a}$ is free in the current

partial design, then it indicates operator f_d is still available to implement activity x_a. This includes a free $\sigma_{d,a}$ variable which has a value of 0 or 1. The decision variables $\rho_{e,a,c}$ provide the same information about the use of storage elements to hold values.

The ordering between two activities (or operations) can be inferred from the $\alpha_{a1,a2}$ decision variable. If $\alpha_{a1,a2}$ has a value of 0 (1), then activity x_1 (x_2) starts ahead of activity x_2 (x_1) for the current partial design. If $\alpha_{a1,a2}$ is not fixed, then the decision might change in later partial designs. Similarly, the variables $\alpha_{a1,c1,a2,c2}$ specify the relative order in which values are stored.

To determine if an input value for an operation comes directly from an operator, or from a stored copy, we have to interpret the information from the $\delta_{a,c}$ binary variables. If a variable $\delta_{a,c}$ has a value of 1, then $i_{a,c}$ will access the stored copy, otherwise it will be connected directly to the operator producing the value.

A user can make use of the above information in any way he/she sees fit. In particular, the following are some suggestions:

(1) Setting decision variables.

An expert designer may know from his/her previous experience which operators or storage elements are good choices for certain activities or output values. We know he/she is capable of doing this because this is how a designer designs without the help of a CAD system. In other words, the designer can supply his/her expertise in setting some decision variables, saving a lot of computation in the branch-and-bound execution. If possible, the designer can even supply a complete set of values for all the decision variables to give a good initial integer solution (*i.e.*, a good completed design), which has a good chance to fathom a lot of other subproblems. We can see that without an initial solution, the branch-and-bound algorithm needs to run 2^n subproblems in the worst case to get the first integer solution, where n is the number of decision variables.

At the moment, it is difficult to formalise what rules a designer follows in making his/her design choices. We hope that through the interactive system, we can gain more understanding about the reasoning process of a designer and hence be able to formalise some rules. This is the first requirement for building any expert CAD system, and will be a good topic for future research.

(2) Controlling exploration of partial designs.

As described in section 2.4, each subproblem can be fathomed due to bounds, infeasibility, or integrality. In each case, BandBX uses the largest bound or best projection criterion to select another subproblem to continue. The algorithm does not allow BandBX to abandon the current subproblem at any time before fathoming occurs. If the designer is allowed to look at each active subproblem then the designer has the alternative to examine non-fathomed partial designs (partial implementations which can be improved with respect to the present bound) and to select one to replace the present one at any time, if he/she is not content with the present one. If he/she makes good choices, it will save a lot of branch-and-bound computation since it will lead to earlier fathoming of active branches.

(3) Choosing between alternatives.

A designer can find out the effect of different choices for a particular element in the middle of the design. For example, the designer may want to know what difference it will make if he/she chooses one operator instead of another for some activity. He/she can set the decision variables for choosing the operators and run the corresponding subproblems. The resulting bounds will give an idea of how well each choice performs.

(4) Find good or bad decisions.

The user can look at the solutions along a path in the branch-and-bound tree. Each step in the path represents one (or more) design decisions. Suppose that in a path, a step S which fixes a decision variable x links subproblem A to subproblem B. If step S leads to a large change in the bound value from A to B, then the designer may conclude that the decision represented by the variable x is a significant one.

The designer can also examine all partial solutions which cannot be improved (*i.e.*, the inferior partial designs), corresponding to the subproblems which were fathomed due to bounds or penalties. He/she could then attempt to deduce the choices which caused the fathom, and avoid them in later decision making.

(5) Supply a strategy in the search.

The designer may guide the search by supplying a limited number of alternative partial designs, each with some significant features, and the branch-and-bound search is activated with these starting points. The results will show the performance of each different feature, and the designer can pick one alternative to be refined.

3.3. Time Information

Beside decision information, we can interpret timing information from the continuous variables (*i.e.*, timing variables). We distinguish between two types of values for timing variables based on the values of the decision variables that are related to each timing variable. In the synthesis constraint systems, timing relationships are described by equalities and inequalities and transformed into a set of linear relations. For example, the output time of an activity is the input start time plus the propagation delay through the operator assigned to implement the activity. The equation is written as:

$$T_{oa}(O_a) = T_{xs}(x_a) + \sum_{\{d \mid f_d \in F_a\}} \sigma_{d,a} D_{fp}(f_d) \tag{3.1}$$

The summation essentially selects the proper propagation delay, depending on the operator assigned to implement the activity. When an operator f_1 is chosen, the value of $\sigma_{1,a}$ will be one while the values of $\sigma_{d,a}$ for all other operators f_d will be zero. We define a "proper value" of a timing variable recursively as follows: if a value is a constant, *e.g.*, the initial time, then the value is proper. For equations like the above, if the timing variables on the right hand side (*e.g.*, $T_{xs}(x_a)$ in the above) are proper, and the decision variables on the right hand side are either zero or one, then the value of the timing variable on the left hand side (*e.g.*, the value of $T_{oa}(O_a)$ in the above) is a "proper value".

However, in a partial solution, the choice of an operator for an activity may be undecided. In this case, some, or all, the values of $\sigma_{d,a}$ in the equation will be fractions between 0 and 1. In this case, the value of $T_{oa}(O_a)$ obtained by equation (3.1) is not the exact time, and we call such values "improper values". Since $\sum \sigma_{d,a} = 1$, the sum $\sum \sigma_{d,a} D_{fp}(f_d)$ effectively calculates a weighted

average of the propagation delays for all available operators. If the timing variable $T_{xs}(x_a)$ is proper then the improper value of variable $T_{oa}(O_a)$ is the average output availability time for activity x_a.

The linear form that expresses the time a value can be stored is

$$T_{ss}(o_{a,c}) - T_{oa}(O_a) - \sum_{\{e \mid s_e \in S_{a,c}\}} (D_{ss}(s_e) + \hat{T})\rho_{e,a,c} \geq -\hat{T} \tag{3.2}$$

If $\delta_{a2,c2}$ has a value of 1 for some input $i_{a2,c2}$ which uses output value $o_{a,c}$ then by (2.40) $\sum \rho_{e,a,c} = 1$. In this case, the interpretation of the constraint is similar to that given for (3.1). However, if the values of the variables $\delta_{a2,c2}$ are 0 or fractional for all inputs $i_{a2,c2}$ which access the value $o_{a,c}$, then it may be that $\sum \rho_{e,a,c} \leq 1$. In this case, the value assigned to $T_{ss}(o_{a,c})$ is likely to be misleading and may be completely meaningless. Suppose that $\sum \rho_{e,a,c} = 0.5$, and note that $\hat{T}$, as defined in section 2.2, is generally much greater than $D_{ss}(s_e)$. Then constraint (3.2) reduces to

$$T_{ss}(o_{a,c}) - T_{oa}(O_a) \geq -0.5\hat{T}$$

implying that the value $o_{a,c}$ can be stored before it is available.

The linear relations below express the requirement that no operator is ever assigned to execute two activities at once.

$$T_{xs}(x_{a2}) - T_{xr}(x_{a1}) - \hat{T}\sigma_{d,a1} - \hat{T}\sigma_{d,a2} + \hat{T}\alpha_{a1,a2} \geq -2\hat{T}$$

$$T_{xs}(x_{a2}) - T_{xr}(x_{a1}) - \hat{T}\sigma_{d,a1} - \hat{T}\sigma_{d,a2} - \hat{T}\alpha_{a1,a2} \geq -3\hat{T} \tag{3.3}$$

If one or more of the decision variables $\alpha_{a1,a2}$, $\sigma_{d,a1}$ or $\sigma_{d,a2}$ have fractional values, then $T_{xs}(x_{a2})$ and $T_{xr}(x_{a1})$ are improper and their values are potentially misleading.

From these examples, we can see that the user must be careful to interpret the values calculated for timing variables in the context of the decisions made to that point in the development of the partial implementation. Even in an integer solution (*i.e.*, a complete implementation), some timing variables may have meaningless values. (*E.g.*, when, in fact, a value is not stored in the implementation.)

In spite of these difficulties, much useful information can be obtained from the values of the timing variables in the partial implementations:

(1) Obtaining the maximum or minimum computation time

For each activity x_a, the set F_a contains the possible choices of operators to be used in the implementation. For each output value $o_{a,c}$, the set $S_{a,c}$ contains the possible choices of storage elements to be used in the implementation.

In the partial solution of a subproblem, some choices of operators and storage elements are made but some are not. For an activity or output value that has a chosen operator or storage element, we know the exact propagation delay. For an activity or output value that does not have a chosen operator or storage element, we can still get an upper bound and a lower bound on the propagation delay by picking components with the maximum and minimum propagation delay from among the sets F_a and $S_{a,c}$.

Therefore we shall be able to answer a query from the user when he/she wants to know the maximum or minimum computation time that may arise from the current partial solution. To get the maximum value, we use the exact time whenever a component has been decided, and we pick the component with the maximum delay for each undecided activity or value and calculate the total time. To get the minimum time bound, we use the exact time whenever a component has been decided, and we pick the component with the minimum delay for each undecided activity or value and calculate the total time. In the general case, specialised algorithms are required for this analysis, as described in [13].

(2) Reusing operators and storage elements

$T_{xs}(x_a)$ is the time when computation of an activity starts, and $T_{xr}(x_a)$ is the time when all output values of an activity are no longer required. For two activities x_{a1} and x_{a2}, if $T_{xr}(x_{a1})$ is less than $T_{xs}(x_{a2})$, and the times are proper, then the two activities are not overlapping. This means that the operator that implements activity x_{a1} can be reused by x_{a2}. This choice may lead to a good solution since it saves on the number of operators used and hence reduces the cost. The designer can set the decision variables to choose the same operator for both activities.

If some activity or value that x_{a1} or x_{a2} depends on has not been assigned a specific operator or storage element, or the α variables for those activities or values are non-integer (or a combination of these cases), then the times $T_{xr}(x_{a1})$ or $T_{xs}(x_{a2})$ are improper. In such cases we can still derive some useful information. First we calculate the maximum possible value MAX for $T_{xr}(x_{a1})$. This is found as outlined in (1) above. Similarly, we calculate the minimum possible value MIN for $T_{xs}(x_{a2})$. If MAX is less than MIN, then it is not possible for the two activities to overlap, and we can reuse the same operator.

A similar analysis for $T_{sr}(o_{a1,c1})$ and $T_{ss}(o_{a2,c2})$ can determine if it possible for the lifetimes of stored values to overlap.

Chapter 4
An Interactive System for Data Path Synthesis

In this chapter we have chosen to focus primarily on an interactive environment for synthesizing register-level data paths from a data-flow specification. The interactive system is called the Data Path Synthesis Interactive System (DPSIS). DPSIS extracts design information from the mathematical formulation in a form which can be understood by a digital hardware designer (who often has little knowledge about constrained optimisation techniques). The remainder of this chapter will discuss the control flow and data flow of the DPSIS system.

4.1. Overview of DPSIS

DPSIS assists the user to minimise cost or maximise speed subject to time and resource constraints, allocate values to registers and operations to operators, schedule operations and optimise by exploring alternative designs. Since most synthesis problems are NP-complete, programs cannot investigate all alternatives. Therefore, flexibility is important for a synthesis program because the designer must either make decisions in the "best" order, or must be able to select alternatives. DPSIS can provide the above features depending on the designer's knowledge about design.

The overview of the control flow and data flow of DPSIS is illustrated in Figure 4-1. The following major components comprise DPSIS:

- **IDDMA** [5] (The Model Generator Software) is an interactive aid for generating a synthesis model from a data flow representation of a desired behaviour. Two forms of output are provided, a human-oriented output in the form of symbolic relations, and a fixed format output for use by the BandBX module. Our research provides the ability to integrate IDDMA with BandBX by implementing an additional command mode called **"inter"** which we will discuss in more detail in the next chapter.

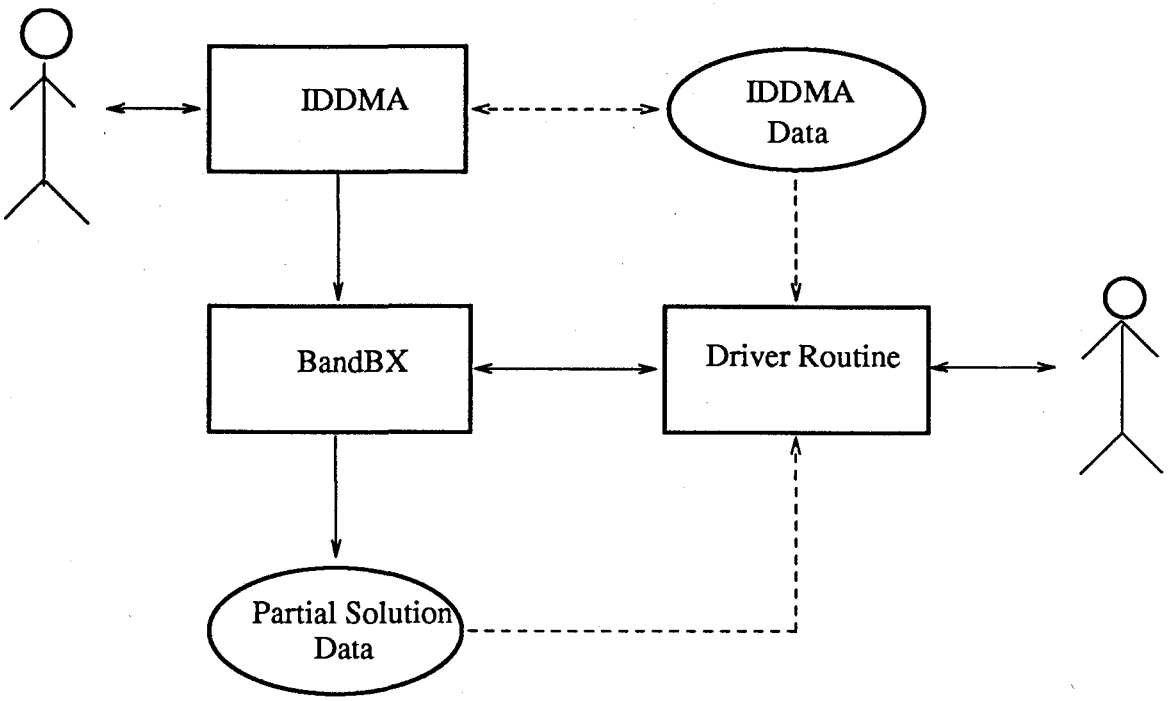

Figure 4-1: Overview of DPSIS

- **BandBX** [8] is the program used to solve the synthesis model as a MILP problem (described in sections 2.3 and 2.4). BandBX uses certain criteria to select the decision variables on which to branch (*i.e.*, make a design decision) and to select another subproblem (*i.e.*, partial implementation) at which to continue when the present subproblem has been fathomed. DPSIS provides the flexibility to allow the user (designer) to select the decision variables and choose subproblems.

- **Driver Routine** is the driver that processes commands by calling the proper routines. The user enters the command he/she needs, then the driver routine checks the command and calls the respective routine. The details of the commands will be discussed in the following sections and chapters.

4.2. Representation of the Synthesis Model's Variables

Table 4-1 shows the representation of variables in DPSIS, and DPSIS uses this format to display to the user. The notation in the first column has already been described in chapter 2. The construct "v*a" in the DPSIS format for the variable names is an artifact of the data-flow representation. The vt translator numbers activities sequentially within larger subdivisions called vt bodies (roughly equivalent to subroutines). Hence both a vt body index and an activity index are necessary to uniquely specify an activity. The details of this representation can be obtained from [5].

$\sigma_{d,a}$	Gs<d,v*a>
$\rho_{e,a,c}$	Gr<e,v*a,c>
$\delta_{a,c}$	Gd<v*a,c>
$T_{ia}(i_{a,c})$	Tia<v*a,c>
$T_{xs}(x_a)$	Txs<v*a>
$T_{oa}(O_a)$	Toa<v*a>
$T_{xr}(x_a)$	Txr<v*a>
$T_{ir}(I_a)$	Tir<v*a>
$T_{or}(o_{a,c})$	Tor<v*a,c>
$T_{ss}(o_{a,c})$	Tss<v*a,c>
$T_{sa}(o_{a,c})$	Tsa<v*a,c>
$T_{sr}(o_{a,c})$	Tsr<v*a,c>

Table 4-1: DPSIS representation of the synthesis model's variables

4.3. Data Structures

This section describes the data structures used in DPSIS and the information that these data structures contain. The dashed lines in Figure 4-1 represent the data flow in DPSIS. From Figure 4-1 the data information in DPSIS is primarily contained in two major groups of data structures: IDDMA's data structures and partial solution data structures.

4.3.1. IDDMA's Data Structures

The next few paragraphs will describe some of IDDMA's data structures that DPSIS uses to supply information to the user.

There is a data structure that provides information about the hardware elements that can be used during the design process. The information that DPSIS can obtain about a hardware element are its actual name, the list of its functions, its size in bits, the setup time prior to clocking it, the hold time after clocking it, its propagation delay, and its cost. The designer usually needs this information when he/she is working on a partially implemented design.

There are two data structures (matrices) that give information about the decision variables of type σ and ρ. The columns in the matrices correspond to storage elements or operators, while the rows of the matrices correspond to outputs or activities. Columns of these matrices record which activities (values) are able to use a particular operator (storage element) for the design. The rows give information about which operators (storage elements) can be used to implement (store) a particular activity (value) during the design process.

The constraint system (in the form (2.49)), is stored in a two dimensional sparse matrix. Each b_i value is kept in a row header, each variable index is kept in a column header, and each value of a_{ij} is stored in an entry of the sparse matrix. With this data structure the designer can examine the original constraints.

The data structure called the *variable translation tree* is an n-ary tree in IDDMA which is used to convert between the indexed variable associated with the vt components (as described in Table 4-1) and a uniform numbering scheme for the variables as required by BandBX. This data structure can provide a human-readable name for any variable by traversing the tree to obtain the variable type and subscripts.

The value trace data structure represents the algorithm to be implemented. This data structure is a directed acyclic graph (DAG) and provides a data-flow specification of the design, as written by the designer. With this data structure the designer knows the data-dependence information of the

activities. Activity x_{a2} is *data-dependent* on activity x_{a1} if we can trace a directed path in the data flow graph from an output of activity x_{a1} to an input of activity x_{a2}.

4.3.2. Partial Solution Data Structures

As already described in the previous two chapters, BandBX solves the linear relaxation of the current subproblem, then acts on the results by fixing monotonic variables, branching, or fathoming the subproblem. But DPSIS interrupts BandBX after it solves the linear relaxation, at which point there is intermediate information regarding the partial implementation of the design. DPSIS and BandBX use this information to maintain the "partial solution data structures" as shown in Figure 4-1. The partial solution data structures keep complete information only for the current subproblem. The next few paragraphs will discuss these data structures.

The information of (2.51) (*i.e.*, $\mathbf{x_{basic}} = \mathbf{b} - \mathbf{A}\mathbf{x_{non\text{-}basic}}$) is stored in a two-dimensional sparse matrix maintained by DPSIS. The values of **b** (basic values) are stored in the row header, the vector $\mathbf{x_{basic}}$ is stored in the row headers, and the values of A are stored in the entries of the sparse matrix. The first row stores the objective function of the subproblem, and the row header of this row contains the bound (*i.e.*, the optimum value) of this subproblem. Each entry of the sparse matrix points to the column header which in turn points to the variable translation tree.

In BandBX each decision variable and timing variable is represented by a numeric identifier. This number can be used to index into an array of structures maintained by DPSIS. Each structure in this array points back to the variable translation tree. With this feature, given any numeric variable identifier, DPSIS can traverse the variable translation tree to obtain the name (as described in Table 4-1) of the design variable or to locate it in the IDDMA data structures. Each structure (corresponding to a variable) in this array also points to a row (if the variable is a basic variable) or a column (if the variable is a non-basic variable) of the two-dimensional sparse matrix that represents the tableau (2.51). In this case, the value of each variable can be obtained by looking up the corresponding array structure. If the variable is a basic variable then value is obtained from the corresponding row in the sparse matrix. If it is a non-basic variable then the variable has the value of 0. Together with the basic values and the corresponding variables in the variable translation tree DPSIS manages to display the the values of the partial design in a human-readable form.

DPSIS also maintains its own copy of the branch-and-bound tree. As mentioned, each node of the tree represents a subproblem. Each node is represented by a structure in the actual implementation. This structure stores the subproblem number used by BandBX. If there are no monotonic variables the structure also stores the decision variable to be branched on and the values of the two subproblems resulting from the branch. In the implementation we defined these two subproblems as the right and left subproblems with respect to the current subproblem. The subproblem that is stored is called the left branch, indicated by the character "l" in the data structure. The subproblem that the system will continue to work on is called the right branch and is indicated by the character "r". If there are monotonic variables, the structure stores the monotonic variables and their values. For the case of monotonic variables there is no left branch because (by definition) a monotonic variable can take only one value, therefore the structure stores the character "c" indicating a forced branch to the next subproblem. With information stored in the structure, the designer can obtain the design decisions made between two subproblems by traversing the path between the two subproblems.

There is an array structure that stores the address of each active subproblem in the branch-and-bound tree (*i.e.*, the nodes in the tree structure mentioned in the previous paragraph). When a subproblem is fathomed, BandBX will update this array by removing the address of the subproblem and compacting the array. The use of this structure will be shown when we describe the command **subpro** in chapter 5.

We have discussed the two matrices which describe the variables of type σ and ρ in IDDMA's data structure. This paragraph will discuss a similar matrix, maintained by DPSIS for timing variables. The columns of the matrix correspond to the types of timing variables, while the rows correspond to outputs or activities. Each entry of the matrix represents an individual timing variable. Each entry points to the corresponding entry in the main DPSIS array used to link the IDDMA and BandBX variable representations. Therefore this structure can be used to obtain the value of all the timing variables in T_j. The main advantage of having this matrix is the flexibility of accessing the timing variables. With this data structure, DPSIS can access a particular type of timing variable by traversing down the column for the variable type. By accessing the rows DPSIS can find out all the timing values corresponding to a particular activity or value.

4.4. Interface Software

DPSIS uses a number of subprograms driven by a main driver to accomplish its tasks. Each of these subprograms performs one or more functions. Figure 4-2 gives the overview picture of functional flows of the driver routine.

The driver calls the respective subprograms based on the command issued by the user. Table 4-2 lists the commands (*i.e.*, the functions) that have been implemented in DPSIS, with a capsule description. The commands will be discussed in detail in the next chapter. The subprograms display the requested design information to the user by extracting the partial solution data generated by the MILP package and translating it into human-readable form.

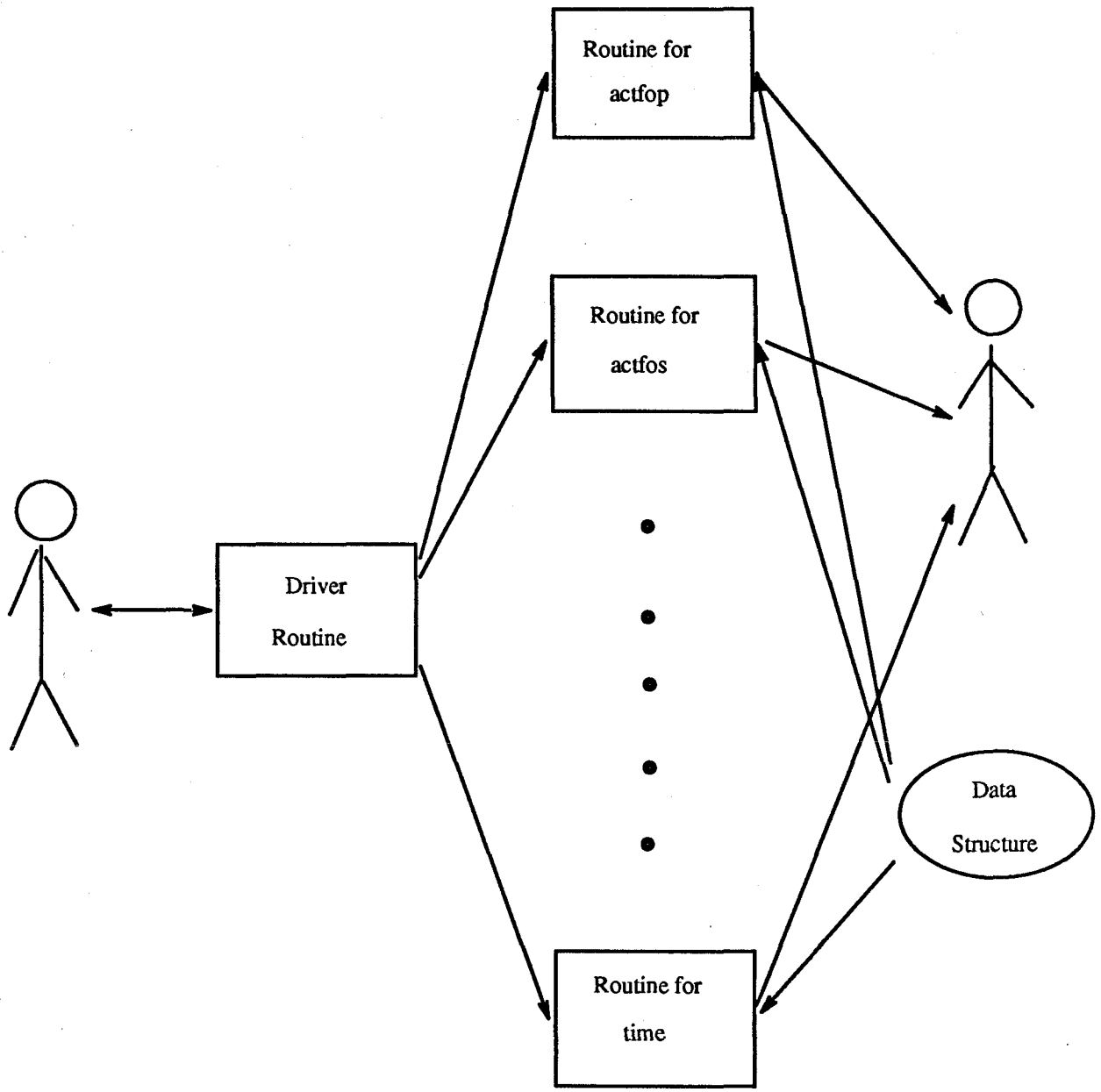

Figure 4-2: Functional flows

actfop	show what operators are available, excluded, and used by an activity.
actfos	show what storage elements are available, excluded, and used by a value.
cont	allow the user to specify a number of subproblems for BandBX to execute before interaction resumes.
fixvar	allow the user to fix decision variables.
flist	show the decision variables that have been fixed, and their values.
help	show the user a list of legal commands.
oper	show what activities are assigned, available and excluded from an operator.
pshoweqn	show an equation (*i.e.*, a row) of the partial solution tableau.
quit	allow the user to quit executing DPSIS at any time.
select	allow the user to select another partial solution.
setfat	allow the user to select another partial solution when the present partial solution has been fathomed.
showeqn	show the user an original constraint relation.
stor	show what activities are assigned, available to, and excluded from a storage element.
subpro	show the active subproblems.
time	show the values of all timing variables.
var	show the value of all variables of a particular type.

Table 4-2: DPSIS commands

Chapter 5
DPSIS Commands

DPSIS has two levels of interaction with the user. The first level of interaction has 18 commands mainly used for generating the constraint relations [5] and displaying human-oriented information about IDDMA internal data structures. The second level, which has been implemented for this thesis, allows the user to interact with the MILP software and obtain design information at any selected subproblem (*i.e.*, partially implemented solution), thus providing flexibility to DPSIS.

5.1. Usage

In [5], Hafer has provided a command interface which can be used interactively or supplied with input from a command file. To interact with the MILP software, place the "inter" command at the end of the command file. Then, simply type

```
% DPSIS < file.cmd
```

During the course of execution of the first level commands, there will be messages indicating the execution of the commands. Then the "inter" command is executed, which starts the second level of interaction by prompting

```
INTER>
```

In this level there are 16 commands.

5.2. Command Descriptions

In this section we will describe each second level command of DPSIS in detail. The description for each command will include:

- a summary of the command's function in terms of the designer's point of view.
- a summary of the command's function in terms of the mathematical formulation.

- an example.

5.2.1. cont

The **cont** command allows the user to specify the number of partial implementations that he/she wants DPSIS to explore before resuming the interaction. In other words, the **cont** command provides the designer the option of whether to participate in the design process or to allow DPSIS to make design decisions automatically.

From a mathematical point of view, interaction is suppressed until BandBX has solved the specified number of subproblems. If the current subproblem is fathomed before the specified number is reached BandBX will select another subproblem and continue. But DPSIS provides another command called **setfat** that will set a flag to "off" or "on". If the flag is on then control is returned to the user after fathoming regardless of the subproblem count.

The example below shows the user asking the system to solve five subproblems before he/she wants to interact with the system again. The user again asks the system to solve another five subproblems. The system abandons the partial implementation after solving two subproblems, because the current partial implementation will not produce a better design. In this case the system will select a new partial implementation to work with. Interaction resumes after solving another three subproblems.

Example :

```
INTER > cont 5 ;

5 more partial implementation
4 more partial implementation
          •
          •
1 more partial implementation
```

```
INTER > cont 5 ;

5 more partial implementation
4 more partial implementation

SUBPROBLEM FATHOMED DUE TO BOUND

3 more partial implementation
2 more partial implementation
1 more partial implementation

INTER >
```

5.2.2. flist

The **flist** command allows the designer to check the design decisions that have been made for the partial design he/she is working on. This command does not show the order of these decisions.

From a mathematical point of view, the branch-and-bound tree is traversed from node v_j (*i.e.*, the current subproblem) to node v_0 (*i.e.*, the root), recording the values assigned to variables fixed due to branching and variables fixed when they were found be monotone. In other words, this command will show the designer the set Π_j.

The example shows the designer asking DPSIS to show the design decisions that have been made. The output shows that hardware component f_5 is not going to be used to implement activity x_5 and hardware component s_1 will be used to implement activity x_4. The designer also knows that input $i_{1,1}$ of activity x_1 and inputs $i_{3,1}$ and $i_{3,2}$ of activity x_3 will be obtained from values held in storage elements.

Example:

```
INTER> flist

Gs<5,7*5>=0  Gr<1,7*4,1>=1  Gd<7*1,1>=1
Gd<7*3,1>=1  Gd<7*3,2>=1

INTER>
```

5.2.3. help

The **help** command gives the designer a table (with a capsule description) of the legal commands that are present in DPSIS.

Example :

```
INTER > help

Table of Commands
-----------------
help      a show a list of legal commands.
•
•
select    select another partial solution.

INTER >
```

5.2.4. quit

The **quit** command allows the designer to stop the whole design process. The designer may wish to do this if cost or performance does not meet design specifications, if the designer is content with the design, or if there is no noninferior design.

Example :

```
          •
          •
INTER > quit ;

the design process is terminated intentionally

 % (back to unix prompt)
```

5.2.5. subpro

The **subpro** command allows the designer to look at any active partial designs, and the information they contain. With this command the designer can determine, for each active partial implementation: 1) the order of design decisions that were made to reach it; 2) its objective value; and; 3) the next decision that BandBX will take;

From the mathematical point of view, the designer is actually looking at the subproblems which are not fathomed (that is, the bound for these subproblems is still better than the current best integer

solution). As already discussed, BandBX selects a binary variable to branch on. If it branches to a value of 1 (or 0) it will continue working on that subproblem and save the the other subproblem that branches to the value of 0 (or 1). In this case, the saved subproblem will contain the information specifying which binary variable it is going to branch on when BandBX recovers this subproblem to continue.

With the addition of the interactive interface to BandBX, a second type of active subproblem is required. When the designer uses the **select** command (described in section 5.2.6) to choose another subproblem to work with and store the subproblem he/she is working on, a branching variable is not selected, hence the stored subproblem cannot specify one.

The path P_j specifies which of the decision variables have been fixed, as described in the **flist** command, but the **subpro** command also shows the order in which the decision variables were fixed.

There are two options in the command. First, the designer can specify the range of partial designs he/she wants to see. Second, the designer can randomly select the partial designs he/she wants.
Example :

a) `INTER > subpro range 3,4 ;`

```
Number = 3    Current Bound = -27.516879
Node # = 3    Variable Gs<1,7*1> to be fixed = 1
Variables fixed information in order :
Gd<7*1,2> = 1r  Gd<7*5,1> = 0r

Number = 4    Current Bound = -31.415799
Node # = 5    Variable Gb<5> to be fixed = 0
Variables fixed information in order :
Gs<5,7*5> = 0c  Gs<1,7*1> = 0r  Gd<7*1,2> = 1r
Gd<7*5,1> = 0r
```

b) `INTER > subpro random 7 ;`

```
Number = 7    Current Bound = -59.165001
Node # = 8    Variable Gr<1,7*0,1> to be fixed = 0
Variables fixed information in order :
Gb<6> = 0r          Gb<1> = 0r      Gb<5> = 1r
Gs<5,7*5> = 0c  Gs<1,7*1> = 0r  Gd<7*1,2> = 1r
Gd<7*5,1> = 0r
```

The examples show the designer asking DPSIS to display a) the subproblems ranging from 3 to 4, and b) the 7th subproblem. The command is set up this way because there is an array storing the actual location of each subproblem, as described in section 4.3.2. The number used in the command is not the actual subproblem number. Rather, it is the location in the array that contains the address of subproblem v_j. The examples presented show that the designer can find out the current bound (*i.e.*, the optimum value of the linear relaxation), the partial implementation he/she is working on (*i.e.*, the node number), the order in which the fixed decision variables were specified, their value, and the direction these decision variables branch to. The letters "r", "l", and "c" indicate the branching direction of each subproblem, as discussed in section 4.3.2.

5.2.6. select

The **select** command provides the designer with the ability to control the order in which BandBX explores the branch-and-bound search tree.

As explained in section 2.4, BandBX uses bounds and projection criteria to select a new subproblem when the current subproblem is fathomed. The **select** command allows the designer to intervene and select the partial implementation he/she considers most promising. The **setfat** command, described in section 5.2.10, is used to force BandBX to allow designer intervention after a subproblem is fathomed.

It may also happen that the designer will wish to abandon a partial implementation before BandBX can fathom the subproblem. This will occur when the designer's experience leads him/her to believe that the design choices already fixed in the partial implementation will preclude a good final implementation.

From the mathematical point of view, some care is necessary in the second case. When the select command is used to force BandBX to another subproblem before the current one is fathomed, the current subproblem must be stored. This is necessary to ensure no potential optimal solution is missed due to premature pruning of a branch in the search tree. As mentioned in section 5.2.5, the subproblem which is stored differs from those created by BandBX during the normal branching process in that no branching variable has been selected; this choice is postponed until the

subproblem is reactivated.

Example :

```
INTER > subpro random 4 ;

Number = 4    Current Bound = -31.41579
Node # = 5    Variable Gb<5> to be fixed = 0
Variables fixed information in order :
Gs<5,7*5> = 0c  Gs<1,7*1> = 0r  Gd<7*1,2> = 1r
Gd<7*5,1> = 0r

INTER > select 4 ;
```

The example shows the designer examining partial design number 4 with the **subpro** command, then asking DPSIS to store the current partial design and to activate partial design number 4.

5.2.7. showeqn

The **showeqn** command allows the designer to look at the original constraints of the synthesis model in a (more-or-less) human-readable format.

The example below shows the designer asking DPSIS to display the first constraint. And the first constraint tells the designer that activity x_7 can use hardware components f_5, f_6 or f_7 during the design process.

Example:

```
INTER> showeqn 1 ;

1: Gs<5,7*7>+Gs<6,7*7>+Gs<7,7*7> = 1

INTER>
```

5.2.8. pshoweqn

The function of **pshoweqn** is similar to **showeqn**, but it shows the constraints of the current partial solution, rather than those of the original model.

From the mathematical point of view, we are looking at the rows of the tableau, $\mathbf{x}_{\mathbf{basic}} = \mathbf{b} - \mathbf{A}\mathbf{x}_{\mathbf{non\text{-}basic}}$.

The example shows the designer asking DPSIS to display row number 4 of current tableau.

Example:

```
INTER> pshoweqn 4 ;

S230 = 1 -1e+03Gd<7*1,2>-1S107+65Gd<7*1,1>-1S116

INTER>
```

The "S" character followed by a number is the name generated for a slack variable. The number "230" means variable x_{230} in BandBX. Therefore, S230 is the slack variable for constraint number "230 - number of variables". The equation shows that the slack variable S230 is a member of B_j and has a value of 1 since $b_{40j} = 1$. The slack variable S116 has a value of 0 since S116 $\in NB_j$. As for variables Gd<7*1,2> and Gd<7*1,1>, they each have a value of 0 (since both are elements of NB_j), unless they have been fixed at 1.

5.2.9. var

The syntax of the **var** command is "**var**<type of variable>". This command will display the values of all the variables of a particular type.

From the mathematical point of view, for any type of timing variable DPSIS will search the set BT_j to find the particular type of variable, then it will obtain the value of these variables from the basic value vector **b**. If the requested variable type is a decision variable, then DPSIS will search the set BD_j. DPSIS also searches the set NB_j to find variables of the particular type which are nonbasic and hence 0 (or possibly fixed at 1).

The example shows the designer successively asking DPSIS to show the values of the σ, ρ and T_{xs} variables. The "B"s beside the values mean that variables are in the set BT_j (for timing variables) or in the set BD_j for decision variables. The "*" can only apply to decision variables, and it means that the variable is fixed and is in the set Π_j. Variables after the "Nonbasic" are the nonbasic variables that have a value of 0. In the **vars** command, the variables Gs<5,7*5> and Gs<1,7*1> are displayed as variables that are fixed to the value of 0 and also nonbasic.

Example :

```
INTER> vars

Gs<1,7*1>=0*      Gs<1,7*5>=1.00B  Gs<3,7*7>=0.00B
Gs<5,7*1>=1.00B   Gs<5,7*3>=1.00B  Gs<5,7*5>=0*
Gs<5,7*7>=0.90B   Gs<6,7*7>=0.10B

Nonbasic variables:

Gs<6,7*3>=0       Gs<6,7*5>=0       Gs<5,7*5>=0
Gs<4,7*7>=0       Gs<2,7*5>=0       Gs<3,7*3>=0
Gs<1,7*1>=0
INTER> varr

Gr<1,7*0,1>=0.25B Gr<1,7*2,1>=0.03B Gr<1,7*4,1>=0.79B

Nonbasic variables:

Gr<2,7*0,2>=0     Gr<2,7*2,1>=0     Gr<2,7*4,1>=0
Gr<3,7*2,1>=0     Gr<3,7*4,1>=0
INTER> vartxs

Txs<7*7>=174.22B  Txs<7*5>=172.00B  Txs<7*3>=16.06B

Nonbasic variables:

Txs<7*1>=0
```

5.2.10. setfat

The **setfat** command acts as a switch to control the actions BandBX takes when a subproblem is fathomed. When this switch is on, it allows the designer to participate in selecting another partial implementation when the current partial implementation is fathomed. When the switch is off, BandBX will select a new subproblem and proceed automatically.

This command does not have a mathematical explanation. But we could describe it as an additional feature for BandBX, allowing human expertise to aid in selecting the next subproblem. Normally, BandBX uses bounds and best integer projection criteria to select a new subproblem.

In order to see the benefit of the **setfat** command, we give a clear example. The example shows

the designer initially setting the switch "on", then asking DPSIS to solve five subproblems before resuming interaction. After solving three subproblems, the subproblem was fathomed. Since the switch is "on" interaction is resumed, but when the designer tries to continue without selecting a new subproblem, DPSIS reminds the designer that there is no active subproblem. So the designer looks up the active subproblems in the tree and selects one to continue.

Example :

```
INTER> setfat on ;

INTER> cont 5 ;

5 more partial implementation
4 more partial implementation
3 more partial implementation

SUBPROBLEM FATHOMED DUE TO INFEASIBILITY

Enter Interaction System Due to Fathom

INTER > cont 2 ;

No active subproblem to continue

INTER > subpro range 5,6 ;

Number = 5      Current Bound = -59.165001
Node # = 8      Variable Gr<1,7*0,1> to be fixed = 0

      •

      •

INTER > select 5 ;
```

5.2.11. fixvar

The **fixvar** command allows the designer to participate in making design decisions. The designer uses other commands to obtain design information, which can help him/her to identify the next feasible move in decision making (*i.e.*, fixing the binary variables).

In section 2.4 we learned that BandBX used upward and downward penalties to select a decision variable to fix in the next partial implementation. The **fixvar** command provides human interaction (*i.e.*, human experience) in decision making, instead of basing it solely on mathematical criteria.

From the mathematical point of view, the designer is creating another subproblem by adding another binary variable to the set Π_j of fixed variables. The other subproblem (formed using the opposite value of the binary variable) will be stored by BandBX.

The example shows the designer making a design decision by assigning hardware component f_5 to implement activity x_7. After telling DPSIS what variable to fix, the designer has to issue the **cont** command to ask BandBX to do the actual calculation.

Example:

```
INTER> vars

Gs<5,7*1>=1.00B  Gs<1,7*5>=1.00B  Gs<5,7*3>=1.00B
Gs<5,7*7>=0.90B

Nonbasic variables:

Gs<1,7*1>=0      Gs<3,7*7>=0      Gs<5,7*5>=0
Gs<6,7*3>=0      Gs<6,7*5>=0      Gs<6,7*7>=0
Gs<2,7*5>=0      Gs<3,7*3>=0      Gs<4,7*7>=0

INTER> fixvar one Gs<5,7*7> ;

INTER> cont 1 ;
```

5.2.12. actfop and oper

The **actfop** and **oper** commands both yield selected information about the assignment of operators to activities in the current partial implementation. The difference between the two commands lies in the method of selection: **actfop** yields information about a given list of activities, whereas **oper** yields information about a given list of operators.

During the design process, the designer needs to recall which operators have been assigned to a set of activities. Occasionally, the designer would like to know what operators are still available for assignment to a set of activities and what operators are excluded from being used to implement those activities. The **actfop** command provides this information. Conversely, the designer may need to know where a given set of operators are used, available, and excluded in the current partial design. The **oper** command provides this information.

The queries supported by the **actfop** and **oper** commands are very useful to a designer. Knowing the distribution of operators and the timing information (mentioned in sections 3.2 and 3.3), a designer might be able to swap operators to save cost or improve the performance of the design.

From the mathematical point of view, the above queries can be answered by looking at the current values of σ variables. Therefore, the command

actfop scans all $\sigma_{d,a}$ for a given activity x_a and

oper scans all $\sigma_{d,a}$ for a given operator f_d.

Each command takes an additional argument specifying the class of information to be selected:

- "used" : An operator f_d is used by an activity x_a if and only if $\sigma_{d,a} = 1$.
- "exclude" : An operator f_d is excluded from an activity x_a if and only if $\sigma_{d,a} = 0$.
- "avail " : An operator f_d is available for an activity x_a if and only if $\sigma_{d,a}$ is not fixed to 0.

If $\sigma_{d,a} \in \Pi_j$, then its use is fixed in the partial implementation and in all implementations derived from it. This is indicated by a "*" in the command response. Operator f_d is used by activity x_a if $\sigma_{d,a} = 1$; for consistency, the operator is also listed as being available for the activity. If $\sigma_{d,a}$ is fixed at 0, then it is excluded from use by activity x_a.

If $\sigma_{d,a} \notin \Pi_j$, the value of $\sigma_{d,a}$ gives the usage of the operator in this partial implementation. However, the use is not fixed, and may be completely different in implementations derived from the current one. Because the usage may change, f_d will always be listed as available. If $\sigma_{d,a}$ has a value of 0 or 1, f_d is will also be listed as used or excluded, respectively. A variable $\sigma_{d,a} \in NBD_j$ which has a value of 0 and is not fixed, is indicated by a "~" in the command response. Table 5-1 gives a summary of the above explanation.

	$\sigma_{d,a} \in BD_j$	$\sigma_{d,a} \in \Pi_j$	$\sigma_{d,a} \in NBD_j$
used	1	1	n/a
available	$0 \leq \sigma_{d,a} \leq 1$	1	0
excluded	0	0	0

Table 5-1: Design interpretation *vs.* values for σ variables

Example :

```
INTER > vars

Gs<1,7*1>=0*      Gs<1,7*5>=0.00B  Gs<2,7*5>=1.00B
Gs<5,7*1>=1.00B   Gs<5,7*3>=1.00B  Gs<5,7*5>=0*
Gs<5,7*7>=1.00B   Gs<6,7*5>=0.00B

Nonbasic variables:

Gs<3,7*7>=0       Gs<6,7*3>=0      Gs<6,7*7>=0
Gs<3,7*3>=0       Gs<4,7*7>=0      Gs<5,7*5>=0
Gs<1,7*1>=0

INTER > actfop x1,x3,x5,x7 used ;

activity 1 used 5
activity 3 used 5
activity 5 used 2
activity 7 used 5
INTER > actfop x1,x3,x5,x7 avail ;

activity 1 available 5
activity 3 available 6~,5,3~
activity 5 available 6,2,1
activity 7 available 6~,5,4~,3~

INTER > actfop x1,x3,x5,x7 exclude ;

activity 1 excluded 1*
activity 3 excluded 6~,3~
activity 5 excluded 6,5*,1
activity 7 excluded 6~,4~,3~
```

The above example shows the designer asking DPSIS to display the current operator allocation information for activities x_1, x_3, x_5 and x_7. In the fourth query, the designer asks DPSIS to display the information about operators that are excluded by these activities. Consider the display for activity x_5. Currently, none of the operators f_1, f_5 or f_6 are used to implement x_5. For f_5 this decision is fixed for all continuations of the the current partial implementation, whereas f_1 or f_6 might later be used to to implement activity x_5. Consider the display for activity x_7. Currently, none of the operators f_6, f_4 or f_3 are used to implement x_7.

Example :

```
INTER> vars

Gs<1,7*5>=0*     Gs<2,7*5>=0.13B Gs<3,7*7>=0.00B
Gs<5,7*1>=1.00B Gs<5,7*3>=1.00B Gs<5,7*5>=0.87B
Gs<5,7*7>=1*     Gs<6,7*5>=0*     Gs<6,7*7>=0*

Nonbasic variables:

Gs<1,7*1>=0      Gs<6,7*3>=0      Gs<1,7*5>=0
Gs<5,7*7>=0      Gs<6,7*5>=0      Gs<6,7*7>=0
Gs<3,7*3>=0      Gs<4,7*7>=0
INTER> oper f1,f2,f3,f4,f5,f6 used ;

Operator 1 used in activities nowhere
Operator 2 used in activities nowhere
Operator 3 used in activities nowhere
Operator 4 used in activities nowhere
Operator 5 used in activities 7*,3,1
Operator 6 used in activities nowhere
INTER> oper f1,f2,f3,f4,f5,f6 avail ;

Operator 1 available in activities 1~
Operator 2 available in activities 5
Operator 3 available in activities 7,3~
Operator 4 available in activities 7~
Operator 5 available in activities 7*,5,3,1
Operator 6 available in activities 6~
INTER> oper f1,f2,f3,f4,f5,f6 exclude ;

Operator 1 excluded in activities 5*,1~
Operator 2 excluded in activities nowhere
Operator 3 excluded in activities 7,3~
Operator 4 excluded in activities 7~
Operator 5 excluded in activities nowhere
```

The example above shows DPSIS answering queries about operators f_1 through f_6. In the third query, the designer asks DPSIS to display where these operators are still available. Consider the operators f_1 and f_5. For operator f_1, "1~" means that operator f_1 is currently not used to implement activity x_1, but might be in some continuation of the current partial design. Operator f_5 is currently available to implement activities x_7, x_5, x_3 and x_1. It will be used for x_7 in all continuations of the current partial implementation, but possibly not for activities x_5, x_3 and x_1.

5.2.13. actfos and stor

The **actfos** and **stor** commands are similar to the **actfop** and **oper** commands respectively. The difference is that they yield selected information about the assignment of storage elements to output values in the current partial implementation.

From the mathematical point of view, the queries can be answered by looking at the current values of the variables of type ρ, instead variables of type σ. Therefore, the command

actfos scans all $\rho_{e,a,c}$ for a given value $o_{a,c}$, and

oper scans all $\rho_{e,a,c}$ for a given storage element s_e.

The relationship between the mathematical explanation and the design interpretation is similar to that for the commands **actfop** and the **oper**, using the $\rho_{e,a,c}$ variables instead of the $\sigma_{d,a}$ variables. Table 5-2 gives a summary. Due to a design oversight in the implementation, these two commands

	$\rho_{e,a} \in BD_j$	$\rho_{e,a,c} \in \Pi_j$	$\rho_{e,a,c} \in NBD_j$
used	1	1	n/a
available	$0 \leq \sigma_{d,a} \leq 1$	1	0
excluded	0	0	0

Table 5-2: Design interpretation *vs.* values for ρ variables

do not provide complete information in some cases. The flaws are easy to patch up and will be illustrated in the examples.

Example :

```
INTER > varr

Gr<1,7*0,1>=1*     Gr<1,7*2,1>=0.93B Gr<1,7*4,1>=1*
Gr<2,7*4,1>=0.00B Gr<3,7*2,1>=0.00B

Nonbasic variables:

Gr<2,7*0,2>=0      Gr<3,7*4,1>=0      Gr<2,7*2,1>=0
Gr<1,7*0,1>=0      Gr<1,7*4,1>=0
```

```
INTER > actfos x2,x3,x4,x6 used ;

output activity 2 used : none
output activity 3 used : none
output activity 4 used 1*
output activity 6 used : none
INTER > actfos x2,x3,x4,x6 avail ;

output activity 2 available 3,2~,1
output activity 3 available : none
output activity 4 available 3~,2,1*
output activity 6 available : none

INTER > actfos x2,x3,x4,x6 exclude ;

output activity 2 excluded 3,2~
output activity 3 excluded : none
output activity 4 excluded 3~,2
output activity 6 excluded : none

INTER >
```

The above example shows the designer asking DPSIS to display the storage element allocation information for the output values of activities x_2, x_3, x_4 and x_6. The command should specify the outputs. Note that presently we are dealing with only one output from each activity, hence it is sufficient to simply specify the activity. In the third query, the designer asks DPSIS to display information about storage elements that are available to store the output value of the activities for the current partial design. Consider the output value of activity x_4. Currently, storage elements s_1, s_2 and s_3 are available, with s_1 selected for use. In continuations of the current partial implementation, s_1 will always be used.

For reasons of expediency, the syntax for the **actfos** command is not adequate to uniquely specify some values. In particular, external inputs cannot be properly specified. Information given in response to a query about activity x_0 is specific to input $i_{0,1}$. Inputs $i_{0,2}$, $i_{0,3}$, ... cannot be specified.

Example :

```
INTER> varr

Gr<1,7*0,1>=1*     Gr<1,7*2,1>=0.99B Gr<1,7*4,1>=0*
Gr<2,7*0,2>=0.03B Gr<2,7*2,1>=0.01B Gr<2,7*4,1>=1.00B
Gr<3,7*4,1>=0.00B

Nonbasic variables:

Gr<3,7*2,1>=0      Gr<1,7*0,1>=0      Gr<1,7*4,1>=0
INTER> stor s1,s2,s3 used ;

Storage 1 used in activities 0*
Storage 2 used in activities 4
Storage 3 used in activities nowhere
INTER> stor s1,s2,s3 avail ;

Storage 1 available in activities 2,0*
Storage 2 available in activities 4,2,0
Storage 3 available in activities 4,2~

INTER> stor s1,s2,s3 exclude ;

Storage 1 excluded in activities 4*
Storage 2 excluded in activities nowhere
Storage 3 excluded in activities 4,2~
```

The example above shows the designer asking DPSIS about the allocation of storage elements s_1, s_2 and s_3 in the current partial design. In the fourth query, the designer asks where these storage elements are not used. Consider the storage elements s_1 and s_3 of this query. Currently, they are both excluded from being used to store the output value of activity x_4.

Similarly to command **actfos**, the ***stor*** *command* syntax is not adequate to uniquely specify some values. In particular, external inputs cannot be properly specified. Information given in response to query about storage elements to be used to store external values is specific to input $i_{0,1}$. Inputs $i_{0,2}$, $i_{0,3}$, ... cannot be specified.

5.2.14. time

The **time** command allows the designer to obtain the current values of timing variables of all types. He/she can look at a list of individual timing variables, the timing variables of a list of activities, and specific types of timing variables for a list of activities.

From the mathematical point of view, if a timing variable is an element of BT_j then the value of the timing variable is the corresponding basic value. If a timing variable is an element of NB_j then the timing variable has a value of 0.

Example :

```
INTER> time Tor<7*4,1>,Tss<7*2,1>,Tor<7*7,1> ;

Tor<7*4,1>    = 134.224599B
Tss<7*2,1>    = 134.224599B
Tor<7*7,1>    = 304.018109B

INTER> time Txr,Txs x1,x3 ;

time information for activity : 1
Txs<7*1>      = 2.224599B
Txr<7*1>      = 134.224599B

time information for activity : 3
Txs<7*3>      = 2.224599B
Txr<7*3>      = 134.224599B
INTER> time x5,x6,x7 ;

time information for activity : 5
Tia<7*5>      = 174.224599B
Txs<7*5>      = 174.224599B
Txr<7*5>      = 306.224599B
Tor<7*5,1>    = 306.224599B

time information for activity : 6
Tia<7*6>      = 281.224599B
Tor<7*6,1>    = 306.224599B
Tss<7*6,1>    = 306.224599B
Tsr<7*6,1>    = 900.000000B

time information for activity : 7
Tia<7*7>      = 174.224599B
Txs<7*7>      = 174.224599B
Txr<7*7>      = 304.018109B
Tor<7*7,1>    = 304.018109B
```

The above example shows three different ways the **time** command could display timing information. In the first query, the designer requests timing information for the individual timing variables Tor<7*4,1>, Tss<7*2,1> and Tor<7*7,1>. In the second query, the designer requests the values of the T_{xs} and T_{xr} variables for activities x_1 and x_3. In the third query, the designer requests all timing information for activities x_5, x_6 and x_7.

Chapter 6
Evaluation of DPSIS

To investigate the effectiveness of DPSIS, we used it to solve a few synthesis problems.

6.1. Evaluation Criteria

DPSIS is an interactive system that lets the designer guide the exploration of the branch-and-bound tree maintained by the BandBX program. We are interested in whether this guiding results in the optimum solution earlier than if BandBX were allowed to run by itself. From the extensive set of statistics that BandBX compiles, the following three pieces of information were used to evaluate how much time the branch-and-bound process takes with and without DPSIS:

- The number of the subproblem at which the optimal solution was found.
- The total number of subproblems solved.
- The total number of simplex pivots performed to solve all linear relaxations.

Since DPSIS is a man-machine interface, the above three numbers will depend on the decisions made by the designer.

The next three sections will introduce three design problems: CrissX, Logic and Power. We walk through a DPSIS session for each of these problems, showing how DPSIS helps the designer in making design decisions.

6.2. CrissX Example

Figure 6-1 presents the data flow representation of the algorithm. Activities x_2, x_4, x_6 and x_8 are field extraction operations which produce as outputs a subfield of their input. These field extraction operations are needed because activities x_1, x_3, x_5 and x_7 have 16 bit inputs and will produce an output of 17 bits. No operators are required to implement these activities, since they are performed simply by connecting to the proper bits of the values $o_{5,1}$, $o_{7,1}$, $o_{1,1}$, and $o_{3,1}$.

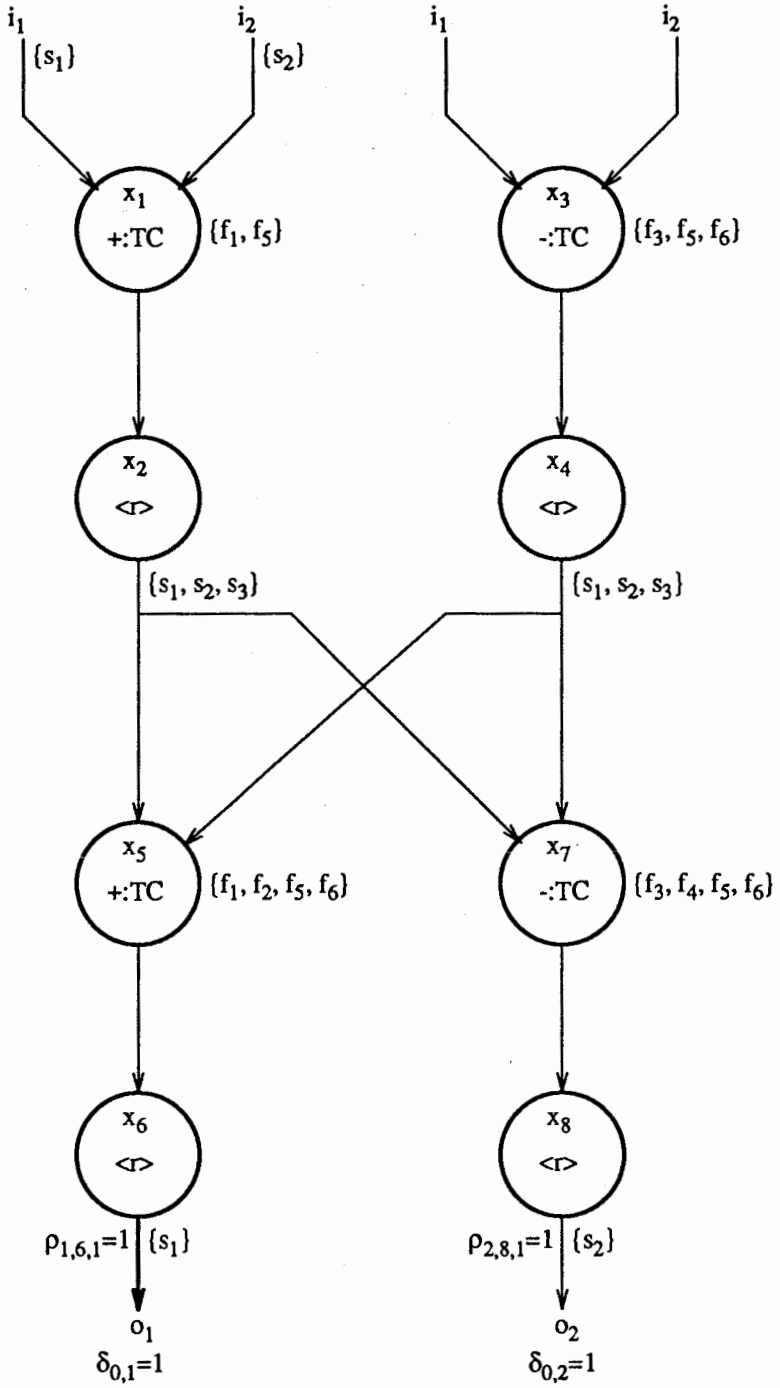

Figure 6-1: Data flow representation for the CrissX problem

The period during which the inputs I_0 are valid is restricted to the interval 0 to 100 ns., and we have assumed a single control signal to latch the inputs. As shown in Figure 6-2, we have required the output values O_0 to be accessed from register outputs, and forced the values $o_{6,1}$ and $o_{8,1}$ to be stored, by setting the variables $\delta_{0,1}$, $\delta_{0,2}$, $\rho_{1,6,1}$, and $\rho_{2,8,1}$ to 1. The outputs have to be valid at the time t=800 ns. and remain valid until the time t=900 ns. The objective of this design problem is to optimise the cost. The proper objective function is

$$19.8\beta_6 + 19.8\beta_5 + 23.7\sigma_{4,7} + 23.7\beta_3 + 14\sigma_{2,5} + 14\beta_1 + 8.83\chi_3$$

where β and χ are derived binary variables (described in section 2.2) which are 1 if the hardware element they represent is ever used in the implementation. The coefficient for each variable is the cost of the hardware element.

Translated into a MILP constraint set, this problem consists of 144 constraints involving 88 variables, of which 47 are decision (binary) variables. The details of the hardware set available for

$S_{0,1} = \{s_1\}$
$S_{0,2} = \{s_2\}$
$S_{1,1} = \{s_1, s_2, s_3\}$
$S_{2,1} = \{s_1, s_2, s_3\}$
$S_{3,1} = \{s_1\}$
$S_{4,1} = \{s_2\}$
$F_1 = \{f_1 f_5\}$
$F_3 = \{f_3, f_5, f_6\}$
$F_5 = \{f_1, f_2, f_5, f_6\}$
$F_7 = \{f_3, f_4, f_5, f_6\}$

$T_{OA}(i_{0,1}) = 0$
$T_{OA}(i_{0,2}) = 0$
$T_{OR}(i_{0,1}) \leq 100$ ns
$T_{OR}(i_{0,2}) \leq 100$ ns
$T_{SS}(i_{0,1}) = T_{SS}(i_{0,2})$
$\delta_{0,1} = 1$
$\delta_{0,2} = 1$
$\rho_{1,3,1} = 1$
$\rho_{2,4,1} = 1$

Figure 6-2: Restrictions on the implementation of CrissX

implementation are given 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 (IDDMA maintains this information, as described in section 4.3.1).

To ease the explanation a number is attached to each command. We begin when BandBX has just finished solving the first relaxation problem, when no decision variable has been fixed.

Initially, the designer has the image of the data flow representation of this problem, and the goal of minimising its cost. The designer will try to reuse storage elements and operators whenever

storage	bits	D_{SS}	D_{SH}	D_{SP}	cost
s_1	<16>	25 ns.	0 ns.	40 ns.	$8.38
s_2	<16>	25 ns.	0 ns.	40 ns.	$8.38
s_3	<16>	25 ns.	0 ns.	40 ns.	$8.38

operator	bits	function	D_{FP}	cost
f_1	<16>	+	70 ns.	$13.96
f_2	<16>	+	70 ns.	$13.96
f_3	<16>	-	85 ns.	$23.71
f_4	<16>	-	85 ns.	$23.71
f_5	<16>	ALU	107 ns.	$19.80
f_6	<16>	ALU	107 ns.	$19.80

Table 6-1: Hardware elements for CrissX implementation

he/she finds it possible. To begin with, the designer requests the current values of the operator mapping variables:

```
1 INTER> vars

  Gs<1,7*1>=0.00B Gs<3,7*7>=0.00B Gs<5,7*1>=1.00B
  Gs<5,7*3>=1.00B Gs<5,7*5>=0.98B Gs<5,7*7>=0.98B
  Gs<6,7*3>=0.00B Gs<6,7*5>=0.02B Gs<6,7*7>=0.02B

  Nonbasic variables:

  Gs<1,7*5>=0     Gs<2,7*5>=0     Gs<3,7*3>=0
  Gs<4,7*7>=0
```

The **vars** command shows the value of each operator mapping variable. Currently, activity x_3 is using operator f_5 (Gs<5,7*3> = 1). The designer knows that activity x_5 is data-dependent on activity x_3, and therefore x_5 can reuse operator f_5, thus decreasing the design cost. The designer instructs BandBX to fix this decision (by setting Gs<5,7*5> = 1) and go to the next partial design. The alternate partial design (in which Gs<5,7*5> = 0) is stored in case it's needed later.

```
2 INTER> fixvar one Gs<5,7*5> ;

3 INTER> cont 1 ;

  1 more partial implementation
```

After making the decision the designer requests the information regarding the value of the operator mapping variables again.

```
4 INTER> vars

  Gs<1,7*1>=0.02B Gs<3,7*3>=0.00B Gs<5,7*1>=0.98B
  Gs<5,7*3>=0.98B Gs<5,7*5>=1*     Gs<5,7*7>=1.00B
  Gs<6,7*3>=0.02B Gs<6,7*5>=0.00B Gs<6,7*7>=0.00B

  Nonbasic variables:

  Gs<1,7*5>=0      Gs<2,7*5>=0      Gs<3,7*7>=0
  Gs<4,7*7>=0      Gs<5,7*5>=0
```

The **vars** command shows that the designer's previous decision changed the value of variable Gs<5,7*3> from 1 to 0.98. Note that variable Gs<5,7*5> is fixed and in the nonbasic set. (The explanation is given in section 2.4.1.) The designer decides to make sure that the operator f_5 will be used to implement activity x_3 if there is a complete design for the current subproblem

```
5 INTER> fixvar one Gs<5,7*3> ;

6 INTER> cont 1 ;
```

The designer tries to find out more information regarding the distribution of operator f_5 and the values of the operator mapping variables.

```
7 INTER> oper f5 used ;

  Operator 5 used in activities 5*,3*,1

8 INTER> vars

  Gs<1,7*1>=0.00B Gs<3,7*3>=0.00B Gs<5,7*1>=1.00B
  Gs<5,7*3>=1*     Gs<5,7*5>=1*     Gs<5,7*7>=0.98B
  Gs<6,7*3>=0.00B Gs<6,7*5>=0.00B Gs<6,7*7>=0.02B

  Nonbasic variables:

  Gs<1,7*5>=0      Gs<2,7*5>=0      Gs<3,7*7>=0
  Gs<4,7*7>=0      Gs<5,7*3>=0      Gs<5,7*5>=0
```

From command 7, the designer knows that operator f_5 is currently shared among activities x_5, x_3 and x_1. The designer decides to have activity x_7 use it as well (Gs<5,7*1> = 1). Then, if a complete solution results, it will use only one operator, which might minimise cost.

```
9 INTER> fixvar one Gs<5,7*7> ;

10 INTER> cont 1 ;
```

The designer confirms that the current design uses operator f_5 in all the activities that need an operator. The designer then requests timing information (execution start and release time) for all the activities.

```
11 INTER> oper f5 used ;

   Operator 5 used in activities 7*,5*,3*,1
12 INTER> time Txs,Txr x1,x3,x5,x7 ;

   time information for activity : 1
   Txs<7*1>      = 2.224599B
   Txr<7*1>      = 134.224599B

   time information for activity : 3
   Txs<7*3>      = 2.224599B
   Txr<7*3>      = 134.224599B
   time information for activity : 5
   Txs<7*5>      = 134.224599B
   Txr<7*5>      = 266.224599B

   time information for activity : 7
   Txs<7*7>      = 134.224599B
   Txr<7*7>      = 518.839984B
```

Activity x_1 overlaps with x_3 and activity x_5 overlaps with x_7. The designer would like to obtain some information regarding the storage elements.

```
13 INTER> stor s1,s2,s3 used ;

   Storage 1 used in activities nowhere
   Storage 2 used in activities nowhere
   Storage 3 used in activities nowhere

14 INTER> vard

   Gd<7*7,2>=0.38B Gd<7*7,1>=0.38B Gd<7*5,2>=0.13B
   Gd<7*5,1>=0.13B Gd<7*3,2>=0.03B Gd<7*3,1>=0.03B
   Gd<7*1,2>=0.03B Gd<7*1,1>=0.03B

   Nonbasic variables:
```

The **stor** tells the designer that none of the storage elements is assigned to an activity output. The

vard command gives information about the inputs of all activities - whether an input comes from a storage element ($\delta_{a,c}$ = 1) or directly from an operator ($\delta_{a,c}$ = 0). This, together with the information provided by commands 11 and 12, indicates that storage elements have to be assigned to the output values of activities x_3 and x_1 in order to allow reuse of operator f_5 by activities x_5 and x_7. The designer decides to have the second input to activity x_5 (*i.e.*, $i_{5,2}$) come from a storage element (Gd<7*5,2> = 1). Note that all the Gd<v*a,c> variables are basic. Therefore, there are no variables displayed after the "Nonbasic variables:" message.

```
15 INTER> fixvar one Gd<7*5,2> ;

16 INTER> cont 1 ;
```

At this time, the designer is still trying to obtain information regarding where the inputs of each of the activities comes from.

```
17 INTER> vard

   Gd<7*7,2>=0.38B Gd<7*7,1>=0.38B Gd<7*5,2>=1*
   Gd<7*5,1>=0.17B Gd<7*3,2>=0.03B Gd<7*3,1>=0.03B
   Gd<7*1,2>=0.03B Gd<7*1,1>=0.03B

   Nonbasic variables:

   Gd<7*5,2>=0
```

From the new **vard** information, the designer knows that the second input to activity x_7 (*i.e.*, $i_{7,2}$) has to come from a storage element (Gd<7*7,2> = 1), if operator f_5 is used in both activities x_5 and x_7.

```
18 INTER> fixvar one Gd<7*7,2> ;

19 INTER> cont 1 ;
```

It is now time to look at the ordering information between activities.

```
20 INTER> vara

   Ga<7*3,7*1>=0.13B Ga<7*7,7*5>=0.13B

   Nonbasic variables:

   Ga<7*2,1,7*0,1>=0 Ga<7*2,1,7*0,2>=0 Ga<7*4,1,7*0,1>=0
   Ga<7*4,1,7*0,2>=0 Ga<7*4,1,7*2,1>=0 Ga<7*6,1,7*0,1>=0
   Ga<7*6,1,7*2,1>=0 Ga<7*6,1,7*4,1>=0 Ga<7*8,1,7*0,2>=0
   Ga<7*8,1,7*2,1>=0 Ga<7*8,1,7*4,1>=0
```

The **vara** command shows that the ordering of activities x_1 and x_3, and of x_5 and x_7 has not been decided yet. The designer must decide the order of activities x_1 and x_3 since they both use the operator f_5. First, the designer recalls what design decisions have been made, to ensure that his/her present analysis is on the right track.

```
21 INTER> flist

   Gs<5,7*7>=1 Gs<5,7*5>=1 Gs<5,7*3>=1
   Gd<7*5,2>=1 Gd<7*7,2>=1
```

Then he/she checks the input available time, execution start time and execution release time for activities x_1 and x_3.

```
22 INTER> time Tia,Txs,Txr x1,x3 ;

   time information for activity : 1
   Tia<7*1>      = 2.224599B
   Txs<7*1>      = 2.224599B
   Txr<7*1>      = 134.224599B

   time information for activity : 3
   Tia<7*3>      = 2.224599B
   Txs<7*3>      = 2.224599B
   Txr<7*3>      = 134.224599B
```

The data-flow diagram (Figure 6-1) shows that inputs to the activities are available at the same time. It will make no difference which activity starts first. The designer chooses to have x_3 start before x_1 (Ga<7*3,7*1> = 1).

```
23 INTER> fixvar one Ga<7*3,7*1> ;

24 INTER> cont 1 ;
```

The designer confirms the ordering between activities x_1 and x_3, and at the same time requests the

ordering information for activities x_5 and x_7. Then, the designer requests the current values of the operator mapping variables and the timing information for activities x_1, x_3, x_5 and x_7.

```
25 INTER> vara

   Ga<7*3,7*1>=1* Ga<7*7,7*5>=0.13B

   Nonbasic variables:

   Ga<7*2,1,7*0,1>=0 Ga<7*2,1,7*0,2>=0 Ga<7*4,1,7*0,1>=0
   Ga<7*4,1,7*0,2>=0 Ga<7*4,1,7*2,1>=0 Ga<7*6,1,7*0,1>=0
   Ga<7*6,1,7*2,1>=0 Ga<7*6,1,7*4,1>=0 Ga<7*8,1,7*0,2>=0
   Ga<7*8,1,7*2,1>=0 Ga<7*8,1,7*4,1>=0 Ga<7*3,7*1>=0
26 INTER> vars

   Gs<1,7*1>=0.07B Gs<3,7*3>=0.00B Gs<5,7*1>=0.93B
   Gs<5,7*3>=1*    Gs<5,7*5>=1*    Gs<5,7*7>=1*
   Gs<6,7*3>=0.00B Gs<6,7*5>=0.00B Gs<6,7*7>=0.00B

   Nonbasic variables:

   Gs<1,7*5>=0     Gs<2,7*5>=0     Gs<3,7*7>=0
   Gs<4,7*7>=0     Gs<5,7*3>=0     Gs<5,7*5>=0
   Gs<5,7*7>=0
27 INTER> time Tia,Txs,Txr x1,x3,x5,x7 ;

   time information for activity : 1
   Tia<7*1>     = 65.327607B
   Txs<7*1>     = 65.327607B
   Txr<7*1>     = 243.121591B

   time information for activity : 3
   Tia<7*3>     = 2.224599B
   Txs<7*3>     = 2.224599B
   Txr<7*3>     = 134.224599B

   time information for activity : 5
   Tia<7*5>     = 174.224599B
   Txs<7*5>     = 174.224599B
   Txr<7*5>     = 306.224599B

   time information for activity : 7
   Tia<7*7>     = 174.224599B
   Txs<7*7>     = 174.224599B
   Txr<7*7>     = 306.224599B
```

The **vara** and **time** commands confirm that activity x_3 starts before activity x_1. The **vars** command

indicates that operator f_5 is not fully assigned to activity x_1 (Gs<5,7*1> = 0.93), which was assumed earlier. Knowing this information, the designer is interested in where the inputs of these activities come from.

```
28 INTER> vard

   Gd<7*7,2>=1*     Gd<7*7,1>=0.06B Gd<7*5,2>=1*
   Gd<7*5,1>=0.06B  Gd<7*3,2>=0.03B Gd<7*3,1>=0.03B
   Gd<7*1,2>=0.90B  Gd<7*1,1>=0.14B

   Nonbasic variables:

   Gd<7*7,2>=0      Gd<7*5,2>=0
```

The **vard** command tells the designer that the inputs to activity x_1 have not been fixed to come from a storage element or directly from the outside world. Since the propagation delay time for operator f_5 is 107 ns., and the output release times of inputs $i_{0,1}$ and $i_{0,2}$ are less than or equal to 100ns., the inputs to activity x_1 and x_3 have to be stored in order to allow their sharing of operator f_5. The designer gathers the information from the last four commands and makes three decisions. He/she assigns operator f_5 to activity x_1 (Gs<5,7*1> = 1) which was assumed earlier. He/she chooses to have activity x_5 start before activity x_7 (Ga<7*7,7*5> = 0). (As with x_1 and x_3, it doesn't matter which goes first.) And he/she decides to have the second input to activity x_1 come from a storage element (Gd<7*1,2> = 1).

```
29 INTER> fixvar one Gs<5,7*1>,Gd<7*1,2> ;

30 INTER> fixvar zero Ga<7*7,7*5> ;

31 INTER> cont 1 ;

   3 more partial implementation
   2 more partial implementation
   1 more partial implementation
```

In command 31, the designer asks DPSIS to continue to the next subproblem, but DPSIS displayed the information that three subproblems are solved. The reason is, the designer has made three decisions and BandBX has to fix the decision variables one at a time. After making these decisions, the designer would like to know the information about the operator mapping variables and the timing variables.

```
32 INTER> oper f5 used ;

   Operator 5 used in activities 1*,3*,5*,7*

33 INTER> vars

   Gs<1,7*1>=0.00B Gs<1,7*5>=0.00B Gs<3,7*3>=0.00B
   Gs<5,7*1>=1*    Gs<5,7*3>=1*    Gs<5,7*5>=1*
   Gs<5,7*7>=1*    Gs<6,7*3>=0.00B Gs<6,7*7>=0.00B

   Nonbasic variables:

   Gs<2,7*5>=0     Gs<3,7*7>=0 Gs<4,7*7>=0
   Gs<5,7*1>=0     Gs<5,7*3>=0 Gs<5,7*5>=0
   Gs<5,7*7>=0     Gs<6,7*5>=0
34 INTER> time Txs,Txr x1,x3,x5,x7 ;

   time information for activity : 1
   Txs<7*1>     = 134.224599B
   Txr<7*1>     = 306.224599B

   time information for activity : 3
   Txs<7*3>     = 2.224599B
   Txr<7*3>     = 134.224599B

   time information for activity : 5
   Txs<7*5>     = 306.224599B
   Txr<7*5>     = 438.224599B

   time information for activity : 7
   Txs<7*7>     = 438.224599B
   Txr<7*7>     = 570.224599B
```

The **oper** command shows that if there is a complete design for the current partial design then operator f_5 will be used to implement all the activities that need an operator. The **time** command indicates that the activities no longer overlap. The designer now must decide where to allocate storage elements in the current design.

```
35 INTER> vard

   Gd<7*7,2>=1* Gd<7*7,1>=0.30B Gd<7*5,2>=1*
   Gd<7*5,1>=1* Gd<7*3,2>=0.03B Gd<7*3,1>=0.03B
   Gd<7*1,2>=1* Gd<7*1,1>=0.21B

   Nonbasic variables:

   Gd<7*7,2>=0  Gd<7*5,2>=0     Gd<7*5,1>=0

36 INTER> stor s1,s2,s3 used ;

   Storage 1 used in activities 2
   Storage 2 used in activities 0
   Storage 3 used in activities nowhere
```

Due to the current implementation, the information for storage element s_2 is incomplete. A proper response should specify the identity of the external input (the output of activity x_0) that s_2 is used for (see Section 5.3.3).

```
37 INTER> varr

   Gr<1,7*0,1>=0.21B Gr<1,7*2,1>=1.00B Gr<1,7*4,1>=0.70B
   Gr<2,7*0,2>=1.00B Gr<2,7*4,1>=0.30B Gr<3,7*4,1>=0.00B

   Nonbasic variables:

   Gr<2,7*2,1>=0     Gr<3,7*2,1>=0
```

The **vard** command shows the designer that his/her last 3 decisions have forced DPSIS to make a decision, that is, the first input value to activity x_5 must come from a storage element (Gd<7*5,1> = 1* is a monotonic variable). And the **stor** and **varr** commands show that no storage element is permanently assigned to any activity's output value. Since the first inputs of activities x_5 and x_7 are from the same storage element, the designer makes the obvious decision (Gd<7*7,1> = 1).

```
38 INTER> fixvar one Gd<7*7,1> ;

39 INTER> cont 1 ;
```

At this point, the designer continues investigating the distribution of storage elements in the current partial design and then their timing information.

```
40 INTER> stor s1,s2,s3 exclude ;

   Storage 1 excluded in activities nowhere
   Storage 2 excluded in activities nowhere
   Storage 3 excluded in activities 4~,2
41 INTER> vard

   Gd<7*7,2>=1* Gd<7*7,1>=1*     Gd<7*5,2>=1*
   Gd<7*5,1>=1* Gd<7*3,2>=0.03B Gd<7*3,1>=0.03B
   Gd<7*1,2>=1* Gd<7*1,1>=0.21B

   Nonbasic variables:

   Gd<7*7,2>=0  Gd<7*7,1>=0      Gd<7*5,2>=0
   Gd<7*5,1>=0  Gd<7*1,2>=0
42 INTER> varr

   Gr<1,7*0,1>=0.21B Gr<1,7*2,1>=0.87B Gr<1,7*4,1>=0.70B
   Gr<2,7*0,2>=1.00B Gr<2,7*2,1>=0.13B Gr<2,7*4,1>=0.30B
   Gr<3,7*2,1>=0.00B

   Nonbasic variables:

   Gr<3,7*4,1>=0
```

```
43 INTER> time x2,x4,x6,x8 ;

   time information for activity : 2
   Tia<7*2>      = 241.224599B
   Tor<7*2,1>    = 266.224599B
   Tss<7*2,1>    = 266.224599B
   Tsr<7*2,1>    = 570.224599B

   time information for activity : 4
   Tia<7*4>      = 109.224599B
   Tor<7*4,1>    = 134.224599B
   Tss<7*4,1>    = 134.224599B
   Tsr<7*4,1>    = 570.224599B

   time information for activity : 6
   Tia<7*6>      = 413.224599B
   Tor<7*6,1>    = 438.224599B
   Tss<7*6,1>    = 438.224599B
   Tsr<7*6,1>    = 900.000000B

   time information for activity : 8
   Tia<7*8>      = 545.224599B
   Tor<7*8,1>    = 570.224599B
   Tss<7*8,1>    = 570.224599B
   Tsr<7*8,1>    = 900.000000B
```

From the **vard** command, the designer knows that both inputs to activities x_7 and x_5 are from storage elements. The **time** command shows that storage start time (T_{ss}) and storage release time (T_{sr}) of activities x_2 and x_4 are overlapped. At this point, the designer knows that the two inputs for activity x_7 (and similarly, those for activity x_5) have to be stored in different storage elements, since the activity needs both inputs to start. Therefore, the designer decides to assign the storage elements s_2 and s_1 to the output values of activities x_2 (Gr<2,7*2,1> = 1) and x_4 (Gr<1,7*4,1> = 1) respectively. After making the decision, the designer asks DPSIS to run 200 subproblems before resuming interaction.

```
44 INTER> fixvar one Gr<2,7*2,1>,Gr<1,7*4,1> ;

45 INTER> cont 200 ;

   1 more partial implementation
   2 more partial implementation

   •

   •

   last partial implementation report

   no active subproblem, therefore DPSIS stop.

   statistical report

   % (back to unix)
```

Before 200 subproblems have been examined, the system stops because there are no alternative partial designs to explore. DPSIS displays a report which gives the statistical information for the design process. Some of this information will appear in Chapter 7 for comparison purposes. The sample session above shows how the information displayed by DPSIS's commands could help a designer to make design decisions for the CrissX problem.

6.3. Logic Example

Figure 6-3 presents the data flow representation of the algorithm.

The restrictions placed on the implementation are shown in Figure 6-4 and the details of the hardware set are shown in Table 6-2. To ensure satisfactory performance, we require that the period during which the external output $o_{0,1}$ is valid is not less that 100 ns. The performance requirements are that the outputs have to be valid at time t=500 ns. and remain valid until time t=600 ns. The objective of this design problem is to optimise the cost. The proper objective function

$$33.5\chi_3 + 8.38\chi_2 + 8.38\chi_1 + 19.8\beta_7 + 19.8\beta_6 + 12.8\sigma_{5,7} + 12.8\sigma_{4,7} + 12.8\beta_3 + 12.8\sigma_{2,5} + 12.8\beta_1$$

Translated into a MILP constraint system, the problem contains 180 constraints involving 111 variables of which 66 are decision (binary) variables.

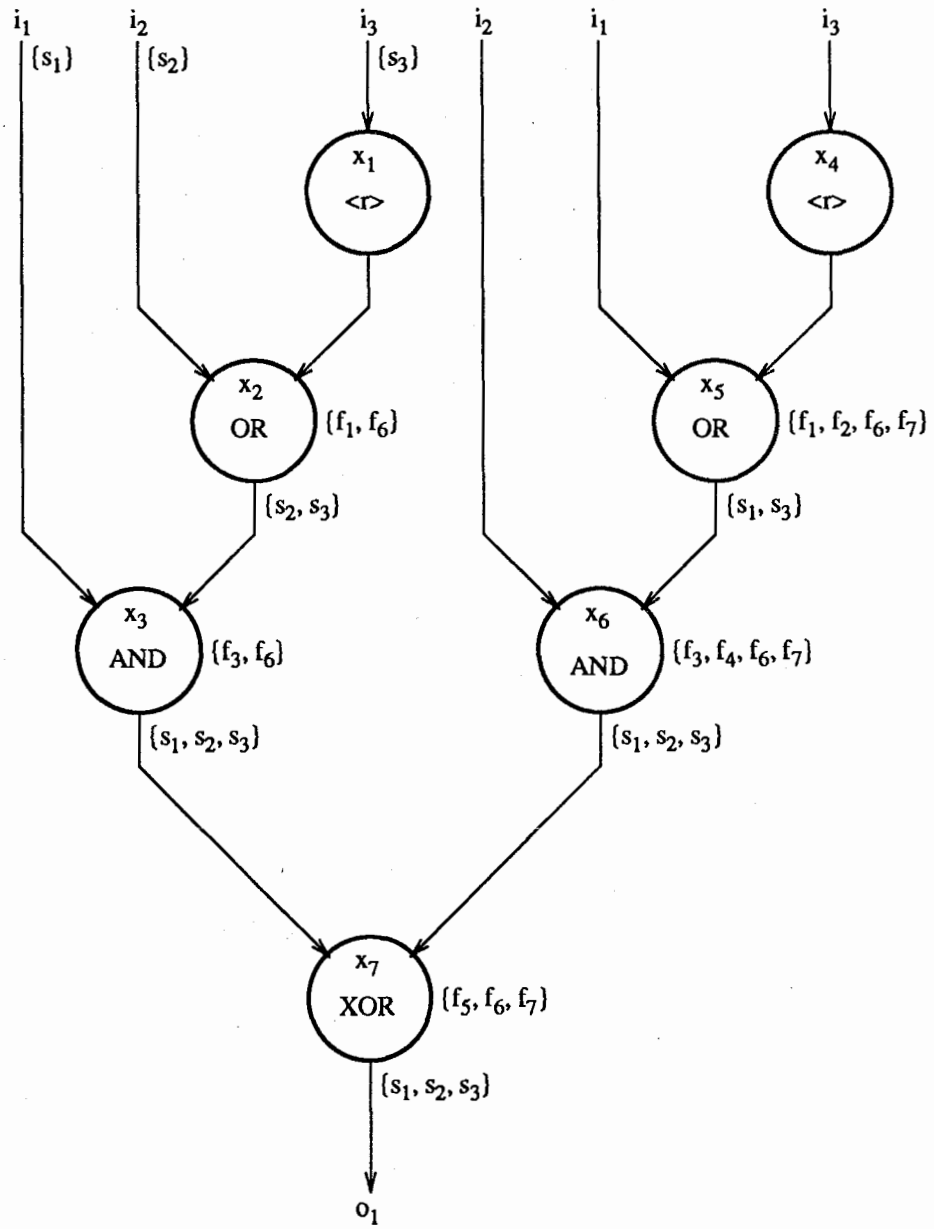

Figure 6-3: Data flow representation for the Logic problem

$S_{0,1} = \{s_1\}$
$S_{0,2} = \{s_2\}$
$S_{0,3} = \{s_3\}$
$S_{2,1} = \{s_2, s_3\}$
$S_{3,1} = \{s_1, s_2, s_3\}$
$S_{5,1} = \{s_1, s_3\}$
$S_{6,1} = \{s_1, s_2, s_3\}$
$S_{7,1} = \{s_1, s_2, s_3\}$
$T_{SS}(i_{0,1}) = T_{SS}(i_{0,2}) = T_{SS}(i_{0,3})$

$T_{OA}(i_{0,1}) = 0$
$T_{OA}(i_{0,2}) = 0$
$T_{OA}(i_{0,3}) = 0$
$F_2 = \{f_1, f_6\}$
$F_3 = \{f_3, f_6\}$
$F_5 = \{f_1, f_2, f_6, f_7\}$
$F_6 = \{f_3, f_4, f_6, f_7\}$
$F_7 = \{f_5, f_6, f_7\}$

Figure 6-4: Restrictions on the implementation of Logic

storage	bits	D_{SS}	D_{SH}	D_{SP}	cost
s_1	<16>	25 ns.	0 ns.	25 ns.	$8.38
s_2	<16>	25 ns.	0 ns.	25 ns.	$8.38
s_3	<64>	25 ns.	0 ns.	25 ns.	$33.52

operator	bits	function	D_{FP}	cost
f_1	<16>	OR	22 ns.	$12.76
f_2	<16>	OR	22 ns.	$12.76
f_3	<16>	AND	20 ns.	$12.84
f_4	<16>	AND	20 ns.	$12.84
f_5	<16>	ALU	48 ns.	$19.00
f_6	<16>	ALU	48 ns.	$19.00

Table 6-2: Hardware elements for Logic implementation

We now discuss a sample session for the Logic problem. This sample shows the usefulness of DPSIS in a different aspect than the CrissX sample. This time, the designer participates in selecting an alternative design when the current design is fathomed due to bounds, integrality or infeasibility.

We begin when BandBX has just finished solving the first relaxation problem, when no decision variable has been fixed. To begin with, the designer requests the current values of the information about operator mapping variables:

```
1 INTER> vars

  Gs<3,7*3>=0.03B Gs<3,7*6>=0.03B Gs<6,7*2>=1.00B
  Gs<6,7*3>=0.97B Gs<6,7*6>=0.97B Gs<7,7*5>=0.00B
  Gs<7,7*6>=0.00B Gs<7,7*7>=0.00B

  Nonbasic variables:

  Gs<1,7*2>=0     Gs<1,7*5>=0     Gs<2,7*5>=0
  Gs<4,7*6>=0     Gs<5,7*7>=0     Gs<6,7*5>=0
  Gs<6,7*7>=0
```

The **vars** command tells the designer that the current partial design uses f_6 to implement activity x_2 (Gs<6,7*2> = 1), but that none of the other activities have been assigned an operator. Since activity x_3 is data-dependent on activity x_2, the designer knows that operator f_6 can be reused in activity x_3 (Gs<6,7*3> = 1), which will lead to a cheaper design.

```
2 INTER> fixvar one Gs<6,7*3> ;

3 INTER> cont 1 ;

  1 more partial implementation
```

The designer requests the current values of the operator mapping variables and where operator f_6 is available.

```
4 INTER> oper f6 avail ;

  Operator 6 available in activities 7,6,5~,3*,2

5 INTER> vars

  Gs<1,7*2>=0.02B Gs<1,7*5>=0.00B Gs<3,7*3>=0.00B
  Gs<3,7*6>=0.02B Gs<6,7*2>=0.98B Gs<6,7*3>=1*
  Gs<6,7*6>=0.98B Gs<6,7*7>=0.97B Gs<7,7*7>=0.03B

  Nonbasic variables:

  Gs<2,7*5>=0     Gs<4,7*6>=0     Gs<5,7*7>=0
  Gs<6,7*3>=0     Gs<6,7*5>=0     Gs<7,7*5>=0
  Gs<7,7*6>=0
```

The designer knows that activity x_3 is data-dependent on activity x_2 and activity x_7 is data-dependent on activity x_3. Since operator f_6 is sure to be used to implement activity x_3, the designer decides to assign operator f_6 to activities x_2 (Gs<6,7*2> = 1) and x_7 (Gs<6,7*7> = 1) as well.

```
6 INTER> fixvar one Gs<6,7*2>,Gs<6,7*7> ;

7 INTER> cont 1 ;

  2 more partial implementation
  1 more partial implementation
```

Next, the designer has in mind to decide the ordering between activities x_3 and x_5.

```
8 INTER> vara

  Ga<7*5,7*2>=0.10B Ga<7*5,7*3>=0.27B Ga<7*6,7*3>=0.04B

  Nonbasic variables:

  Ga<7*0,1,7*3,1>=0 Ga<7*0,1,7*5,1>=0 Ga<7*0,1,7*6,1>=0
  Ga<7*0,2,7*2,1>=0 Ga<7*0,2,7*3,1>=0 Ga<7*0,2,7*6,1>=0
  Ga<7*0,3,7*2,1>=0 Ga<7*0,3,7*3,1>=0 Ga<7*0,3,7*5,1>=0
  Ga<7*0,3,7*6,1>=0 Ga<7*5,1,7*2,1>=0 Ga<7*5,1,7*3,1>=0
  Ga<7*6,1,7*2,1>=0 Ga<7*6,7*2>=0
9 INTER> vars

  Gs<1,7*2>=0.00B Gs<1,7*5>=0.00B Gs<3,7*3>=0.00B
  Gs<3,7*6>=0.03B Gs<6,7*2>=1*    Gs<6,7*3>=1*
  Gs<6,7*6>=0.97B Gs<6,7*7>=1*    Gs<7,7*7>=0.00B

  Nonbasic variables:

  Gs<2,7*5>=0     Gs<4,7*6>=0     Gs<5,7*7>=0
  Gs<6,7*2>=0     Gs<6,7*3>=0     Gs<6,7*5>=0
  Gs<6,7*7>=0     Gs<7,7*5>=0     Gs<7,7*6>=0
```

The **vara** command tells the designer that the ordering of x_3 and x_5 is not decided yet (Ga<7*5,7*3> = 0.27). The **vars** command confirms that operator f_6 will be used in activities x_2, x_3, and x_7 if a complete design exists from the current design. If activities x_2 and x_5 use different operators, then activity x_5 most probably will start before activity x_3. If activities x_2 and x_5 share operator f_6 then either activity can start first. So the designer decides to have activity x_5 start before activity x_3 (Ga<7*5,7*3> = 1).

```
10 INTER> fixvar one Ga<7*5,7*3> ;

   There are forced decision (monotonic variables)
```

This message means that there are decisions which have to be made by DPSIS, so the designer lets DPSIS proceed.

```
11 INTER> cont 1 ;

   1 more partial implementation
```

The designer is interested in knowing what decision the system has just made, and also whether the ordering between activities x_3 and x_5 has changed.

```
12 INTER> flist

   Gs<6,7*7>=1 Gs<6,7*3>=1 Gs<6,7*2>=1
   Gd<7*3,2>=1
13 INTER> vara

   Ga<7*5,7*2>=0.10B Ga<7*5,7*3>=0.27B Ga<7*6,7*3>=0.04B

   Nonbasic variables:

   Ga<7*0,1,7*3,1>=0 Ga<7*0,1,7*5,1>=0 Ga<7*0,1,7*6,1>=0
   Ga<7*0,2,7*2,1>=0 Ga<7*0,2,7*3,1>=0 Ga<7*0,2,7*6,1>=0
   Ga<7*0,3,7*2,1>=0 Ga<7*0,3,7*3,1>=0 Ga<7*0,3,7*5,1>=0
   Ga<7*0,3,7*6,1>=0 Ga<7*5,1,7*2,1>=0 Ga<7*5,1,7*3,1>=0
   Ga<7*6,1,7*2,1>=0 Ga<7*6,7*2>=0
```

The **flist** command shows that the forced decision was to have the second input to activity x_3 come from a storage element (Gd<7*3,2> = 1). The **vara** command shows that the ordering information has not changed, so the designer goes ahead with his/her previous decision to have activity x_5 start before activity x_3 (Ga<7*5,7*3> = 1).

```
14 INTER> fixvar one Ga<7*5,7*3> ;

15 INTER> cont 1 ;

   1 more partial implementation
```

The designer looks at the ordering information and the distribution of operators f_1, f_2, f_6 and f_7 in the design.

```
16 INTER> vara

   Ga<7*5,7*2>=0.10B Ga<7*5,7*3>=1* Ga<7*6,7*3>=0.10B

   Nonbasic variables:

   Ga<7*0,1,7*3,1>=0 Ga<7*0,1,7*5,1>=0 Ga<7*0,1,7*6,1>=0
   Ga<7*0,2,7*2,1>=0 Ga<7*0,2,7*3,1>=0 Ga<7*0,2,7*6,1>=0
   Ga<7*0,3,7*2,1>=0 Ga<7*0,3,7*3,1>=0 Ga<7*0,3,7*5,1>=0
   Ga<7*0,3,7*6,1>=0 Ga<7*5,1,7*2,1>=0 Ga<7*5,1,7*3,1>=0
   Ga<7*5,7*3>=0     Ga<7*6,1,7*2,1>=0 Ga<7*6,7*2>=0
17 INTER> oper f1,f2,f6,f7 used ;

   Operator 1 used in activities nowhere
   Operator 2 used in activities nowhere
   Operator 6 used in activities 7*,6,3*,2*
   Operator 7 used in activities nowhere
18 INTER> vars

   Gs<1,7*2>=0.00B Gs<1,7*5>=0.00B Gs<3,7*3>=0.00B
   Gs<3,7*6>=0.00B Gs<6,7*2>=1*    Gs<6,7*3>=1*
   Gs<6,7*6>=1.00B Gs<6,7*7>=1*    Gs<7,7*7>=0.00B

   Nonbasic variables:

   Gs<2,7*5>=0     Gs<4,7*6>=0     Gs<5,7*7>=0
   Gs<6,7*2>=0     Gs<6,7*3>=0     Gs<6,7*5>=0
   Gs<6,7*7>=0     Gs<7,7*5>=0     Gs<7,7*6>=0
```

The designer decides to have activity x_5 start before activity x_2 (Ga<7*5,7*2> = 1), because he/she wants to know what the effect will be on the operator mapping variables. Then he/she checks the timing information for these activities.

```
19 INTER> fixvar one Ga<7*5,7*2> ;

20 INTER> cont 1 ;

   1 more partial implementation
```

```
21 INTER> vars

   Gs<1,7*2>=0.00B Gs<1,7*5>=0.09B Gs<3,7*3>=0.00B
   Gs<6,7*2>=1*    Gs<6,7*3>=1*    Gs<6,7*5>=0.91B
   Gs<6,7*6>=1.00B Gs<6,7*7>=1*    Gs<7,7*7>=0.00B

   Nonbasic variables:

   Gs<2,7*5>=0     Gs<3,7*6>=0     Gs<4,7*6>=0
   Gs<5,7*7>=0     Gs<6,7*2>=0     Gs<6,7*3>=0
   Gs<6,7*7>=0     Gs<7,7*5>=0     Gs<7,7*6>=0
22 INTER> time Txs,Txr x2,x5 ;

   time information for activity : 2
   Txs<7*2>      = 0.000000B
   Txr<7*2>      = 73.000000B

   time information for activity : 5
   Txs<7*5>      = 0.000000B
   Txr<7*5>      = 65.564738B
```

The **vars** command shows that activity x_5 has not been assigned an operator yet (Gs<6,7*5>=0.91 and Gs<1,7*5>=0.09). The **time** command shows that the execution interval of x_2 and x_5 still overlap. At this point, the designer wants to know what the effect will be if instead, activity x_2 starts before activity x_5. So he/she abandons the current partial design temporarily and selects the partial design that has activity x_2 start before activity x_5 (Ga<7*5,7*2> = 0).

```
23 INTER> subpro range 4,5 ;

   Number = 4 Current Bound = -26.452857
   Node # = 5 Variable Ga<7*5,7*3> to be fixed = 0
   Variables fixed information in order :
   Gd<7*3,2> = 1c Gs<6,7*2> = 1r Gs<6,7*7> = 1r
   Gs<6,7*3> = 1r
   Number = 5 Current Bound = -26.452857
   Node # = 6 Variable Ga<7*5,7*2> to be fixed = 0
   Variables fixed information in order:
   Ga<7*5,7*3> = 1r Gd<7*3,2> = 1c  Gs<6,7*2> = 1r
   Gs<6,7*7> = 1r   Gs<6,7*3> = 1r
```

From the **subpro** command the designer knows that he/she wants to retrieve partial design number 5 to continue.

```
24 INTER> select 5 ;

   1 more partial implementation

25 INTER> flist

   Gs<6,7*7>=1 Gs<6,7*3>=1   Gs<6,7*2>=1
   Gd<7*3,2>=1 Ga<7*5,7*3>=1 Ga<7*5,7*2>=0
```

The **flist** command confirms that the designer chose the right partial design (Ga<7*5,7*2>=0). At this stage he/she wants to look at the distribution of the operators and the operator mapping variables again.

```
26 INTER> oper f1,f2,f6,f7 avail ;

   Operator 1 available in activities 5,2
   Operator 2 available in activities 5~
   Operator 6 available in activities 7*,6,5,3*,2*
   Operator 7 available in activities 7,6~,5~

27 INTER> vars

   Gs<1,7*2>=0.00B Gs<1,7*5>=0.02B Gs<3,7*3>=0.00B
   Gs<3,7*6>=0.00B Gs<6,7*2>=1*    Gs<6,7*3>=1*
   Gs<6,7*5>=0.98B Gs<6,7*6>=1.00B Gs<6,7*7>=1*
   Gs<7,7*7>=0.00B

   Nonbasic variables:

   Gs<2,7*5>=0     Gs<4,7*6>=0     Gs<5,7*7>=0
   Gs<6,7*2>=0     Gs<6,7*3>=0     Gs<6,7*7>=0
   Gs<7,7*5>=0     Gs<7,7*6>=0
```

From commands 18, 21 and 27, the designer sees that the value of Gs<1,7*5> changed from 0 to 0.09 and 0.02, when he/she tried to order activities x_2 and x_5. There is still no operator assigned to activity x_5, and the designer notices that he/she could share operator f_6 with x_5 (Gs<6,7*5> = 1), which decision might minimise cost.

```
28 INTER> fixvar one Gs<6,7*5> ;

29 INTER> cont 1 ;

   1 more partial implementation
```

At this point, the designer hopes that the previous decisions will be a good start for a solution, so

he/she decides to have DPSIS continue solving the design and only participate when a design is fathomed.

```
30 INTER> setfat on ;
```

The **setfat** command allows the designer to interact again when the current design s fathomed.

```
31 INTER> cont 50 ;

   50 more partial implementation
   49 more partial implementation

   SUBPROBLEM FATHOMED DUE TO INFEASIBILITY

   ** Enter Interaction System Due to Fathom **
```

At this stage the designer wants to select another subproblem to continue.

```
32 INTER> subpro random 6,8 ;

   Number = 6      Current Bound = -26.452857
   Node # = 8      Variable Gs<6,7*5> to be fixed = 0
   Variables fixed information in order :
   Ga<7*5,7*2> = 0l Ga<7*5,7*3> = 1r Gd<7*3,2> = 1c
   Gs<6,7*2> = 1r   Gs<6,7*7> = 1r   Gs<6,7*3> = 1r
   Number = 8      Current Bound = -27.200861
   Node # = 10     Variable Gd<7*-1,1> to be fixed = 1
   Variables fixed information in order :
   Gr<1,7*5,1> = 0r Gs<6,7*5> = 1r Ga<7*5,7*2> = 0l
   Ga<7*5,7*3> = 1r Gd<7*3,2> = 1c Gs<6,7*2> = 1r
   Gs<6,7*7> = 1r   Gs<6,7*3> = 1r
```

The designer decides not to choose subproblem number 6, because the next decision to be made is not assigning operator f_6 to activity x_5 (Gs<6,7*5> = 1). The designer wants f_6 to be used in activities x_5 and x_2. As for subproblem number 8, he/she does not like the idea that the output value of the problem comes from a storage element. The designer goes on looking for a subproblem to continue with.

```
33 INTER> subpro range 4,5 ;

   Number = 4     Current Bound = -26.452857
   Node # = 5     Variable Ga<7*5,7*3> to be fixed = 0
   Variables fixed information in order :
   Gd<7*3,2> = 1c Gs<6,7*2> = 1r Gs<6,7*7> = 1r
   Gs<6,7*3> = 1r

   Number = 5      Current Bound = -26.732685
   Node # = 7      Variable to be fixed = none
   Variables fixed information in order :
   Ga<7*5,7*2> = 1r Ga<7*5,7*3> = 1r Gd<7*3,2> = 1c
   Gs<6,7*2> = 1r   Gs<6,7*7> = 1r   Gs<6,7*3> = 1r
```

The designer decides not to choose subproblem number 4, because the next decision to be made is to have activity x_3 start before activity x_5. Finally, the designer decides on subproblem number 5 since the previous design decisions (variables fixed) show that it is a feasible choice.

```
34 INTER> select 5 ;

35 INTER> cont 50 ;

   50 more partial implementation
   49 more partial implementation

   •

   •

   30 more partial implementation

   SUBPROBLEM FATHOMED DUE TO INFEASIBILITY

   ** Enter Interaction System Due to Fathom **
```

At this stage, the designer looks at the active subproblems again to select another subproblem to continue with.

```
36 INTER> subpro range 5,6 ;

   Number = 5 Current Bound = -26.452857
   Node # = 8 Variable Gs<6,7*5> to be fixed = 0
   Variables fixed information in order :
   Ga<7*5,7*2> = 01 Ga<7*5,7*3> = 1r Gd<7*3,2> = 1c
   Gs<6,7*2> = 1r   Gs<6,7*7> = 1r   Gs<6,7*3> = 1r
```

```
Number = 6 Current Bound = -27.12497
Node # = 9 Variable Gr<1,7*5,1> to be fixed = 1
Variables fixed information in order :
Gs<6,7*5> = 1r Ga<7*5,7*2> = 01 Ga<7*5,7*3> = 1r
Gd<7*3,2> = 1c Gs<6,7*2> = 1r   Gs<6,7*7> = 1r
Gs<6,7*3> = 1r
```

The designer does not choose subproblem number 5 because in it, activity x_5 excludes operator f_6, while he/she wants x_5 to share f_6 with other activities. As for subproblem number 6, the operator f_6 is shared among activities x_2, x_3, x_5 and x_7, and the next decision will be to assign storage element s_1 to the output value of activity x_5. This seems reasonable since the output value of activity x_5 must have a storage element in order to share operator f_6, so the designer selects subproblem number 6.

```
37 INTER> select 6 ;

   1 more partial implementation
```

After this point, the designer decides not to participate in selecting a subproblem, if a subproblem is fathomed.

```
38 INTER> setfat off ;

39 INTER> cont 150 ;

   150 more partial implementation
   149 more partial implementation

   •

   •

   141 more partial implementation

   SUBPROBLEM FATHOMED DUE TO INFEASIBILITY

   140 more partial implementation

   •

   1 more partial implementation
```

After solving 9 subproblems the selected subproblem was fathomed, but by turning off the interaction switch in command 38, the designer let DPSIS to do the rest of the design, rather than resuming interaction.

```
40 INTER > cont 300 ;
    •
    •
    SUBPROBLEM FATHOMED DUE TO BOUNDS,
    TEST=-40.48092  BEST=-40.36000
    1 more partial implementation
    last partial implementation report
    no active subproblem, therefore DPSIS stop
    statistical report
    % (back to unix)
```

The sample session above shows how the information displayed by DPSIS could help designer to participate in selecting an alternative design. Parts of the statistical report will be used in Section 7.2.

6.4. Power Example

Figure 6-3 presents the data flow representation of the algorithm.

The restrictions placed on the implementation are show in Figure 6-6 and the details of the hardware set are shown in Table 6-3. The performance requirements are that the outputs have to be valid at time t=500 ns. and remain valid until time t=600 ns. The objective of this design problem is to optimize the cost.

$$40\sigma_{9,8} + 20\sigma_{7,8} + 20\beta_6 + 19.2\beta_5 + 19.2\beta_4 + 9.16\sigma_{3,5} + 9.16\beta_2 + 9.16\beta_1 + 2.79\chi_9 + 2.89\chi_8 + 2.79\chi_7 + 2.79\chi_6 + 1.79\chi_5 + 1.79\chi_4 + 1.89\chi_3 + 1.79\chi_2 + 1.79\chi_1$$

Translated into a MILP constraint system, the problem contains 201 constraints involving 138 variables, of which 77 are decision (binary) variables.

We now discuss a sample session for the Power problem. This sample shows how DPSIS can guide the designer to the first integer solution. To begin with, the designer requests the current values of the operator mapping variables:

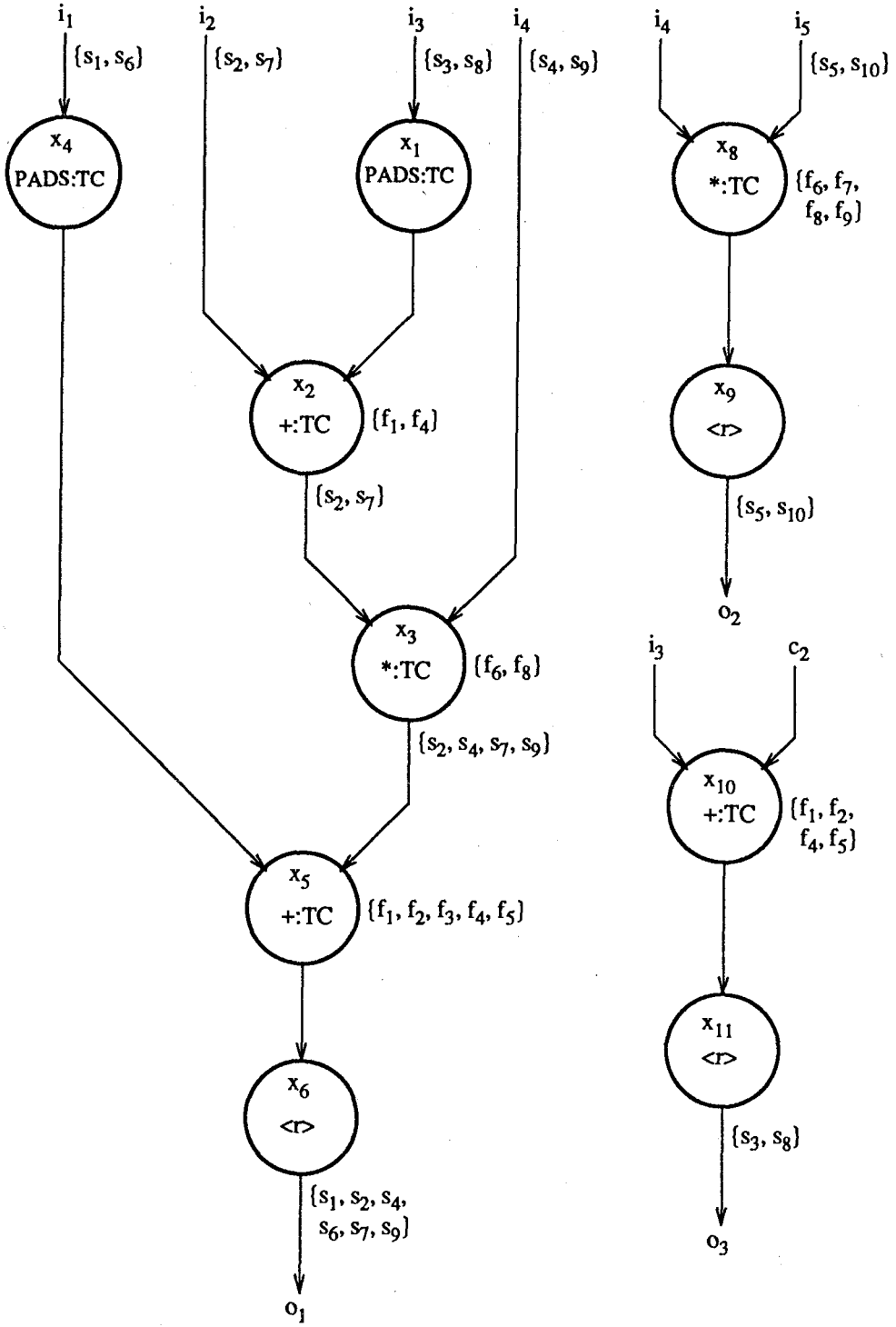

Figure 6-5: Data flow representation for the Power problem

$S_{0,1} = \{s_1, s_6\}$
$S_{0,2} = \{s_2, s_7\}$
$S_{0,3} = \{s_3, s_8\}$
$S_{0,4} = \{s_4, s_9\}$
$S_{0,5} = \{s_5, s_{10}\}$
$S_{2,1} = \{s_2, s_7\}$
$S_{3,1} = \{s_2, s_4, s_7, s_9\}$
$S_{6,1} = \{s_1, s_2, s_4, s_6, s_7, s_9\}$
$S_{9,1} = \{s_5, s_{10}\}$
$S_{11,1} = \{s_3, s_8\}$

$T_{OA}(i_{0,1}) = 0$
$T_{OA}(i_{0,2}) = 0$
$T_{OA}(i_{0,3}) = 0$
$T_{OA}(i_{0,4}) = 0$
$T_{OA}(i_{0,5}) = 0$
$T_{OR}(i_{0,1}) \leq 100$ ns
$F_2 = \{f_1, f_4\}$
$F_3 = \{f_6, f_8\}$
$F_5 = \{f_1, f_2, f_3, f_4, f_5\}$
$F_8 = \{f_6, f_7, f_8, f_9\}$
$F_{10} = \{f_1, f_2, f_4, f_5\}$

Figure 6-6: Restrictions on the implementation of Power

storage	bits	D_{SS}	D_{SH}	D_{SP}	cost
s_1	<16>	20 ns.	5 ns.	27 ns.	$1.79
s_2	<16>	20 ns.	5 ns.	27 ns.	$1.79
s_3	<4>	20 ns.	5 ns.	27 ns.	$1.89
s_4	<16>	20 ns.	5 ns.	27 ns.	$1.79
s_5	<16>	20 ns.	5 ns.	27 ns.	$1.79
s_6	<16>	10 ns.	5 ns.	17 ns.	$2.79
s_7	<16>	10 ns.	5 ns.	17 ns.	$2.79
s_8	<4>	10 ns.	5 ns.	17 ns.	$2.89
s_9	<16>	10 ns.	5 ns.	17 ns.	$2.79
s_{10}	<16>	10 ns.	5 ns.	17 ns.	$2.79

operator	bits	function	D_{FP}	cost
f_1	<32>	+	173 ns.	$9.16
f_2	<32>	+	173 ns.	$9.16
f_3	<32>	+	173 ns.	$9.16
f_4	<32>	+	73 ns.	$19.16
f_5	<32>	+	73 ns.	$19.16
f_6	<32>	*	180 ns.	$20.00
f_7	<32>	*	180 ns.	$20.00
f_8	<32>	*	90 ns.	$40.00
f_9	<32>	*	90 ns.	$40.00

Table 6-3: Hardware elements for Power implementation

```
1 INTER> vars

  Gs<6,9*3>=1.00B  Gs<6,9*8>=1.00B Gs<8,9*3>=0.00B
  Gs<1,9*2>=1.00B  Gs<1,9*5>=1.00B Gs<1,9*10>=0.87B
  Gs<2,9*10>=0.13B Gs<4,9*2>=0.00B

  Nonbasic variables:

  Gs<3,9*5>=0      Gs<7,9*8>=0      Gs<8,9*8>=0
  Gs<9,9*8>=0      Gs<2,9*5>=0      Gs<4,9*5>=0
  Gs<4,9*10>=0     Gs<5,9*5>=0      Gs<5,9*10>=0
```

Currently, activities x_2 and x_5 are using operator f_1 (Gs<1,9*2> = Gs<1,9*5> = 1). The designer decides to have them share operator f_1 with activity x_{10}.

```
2 INTER> fixvar one Gs<1,9*10> ;

3 INTER> cont 1 ;
```

The designer requests the values of the operator mapping variables again.

```
4 INTER> vars

  Gs<6,9*3>=1.00B  Gs<6,9*8>=1.00B Gs<8,9*3>=0.00B
  Gs<1,9*2>=0.99B  Gs<1,9*5>=1.00B Gs<1,9*10>=1*
  Gs<2,9*10>=0.00B Gs<4,9*2>=0.01B

  Nonbasic variables:

  Gs<3,9*5>=0      Gs<7,9*8>=0      Gs<8,9*8>=0
  Gs<9,9*8>=0      Gs<1,9*10>=0     Gs<2,9*5>=0
  Gs<4,9*5>=0      Gs<4,9*10>=0     Gs<5,9*5>=0
  Gs<5,9*10>=0
```

The **vars** command shows that the designer's previous decision changed the assignment of operator f_1 to activity x_2 (*i.e.*, the value of Gs<1,9*2> changed from 1 to 0.99). The designer decides to make sure that the operator f_1 will be used to implement activity x_2 if there is a complete design for the current subproblem.

```
5 INTER> fixvar one Gs<1,9*2> ;

6 INTER> cont 1 ;
```

The designer tries to find out more information regarding the distribution of operators and the values of the operator mapping variables.

```
7 INTER> oper f1,f6,f8,f7,f9 used ;

  Operator 1 used in activities 10*,5,2*
  Operator 6 used in activities 8,3
  Operator 8 used in activities nowhere
  Operator 7 used in activities nowhere
  Operator 9 used in activities nowhere

8 INTER> vars

  Gs<6,9*3>=1.00B  Gs<6,9*8>=1.00B Gs<8,9*3>=0.00B
  Gs<1,9*2>=1*     Gs<1,9*5>=1.00B Gs<1,9*10>=1*
  Gs<2,9*10>=0.00B Gs<4,9*2>=0.00B

  Nonbasic variables:

  Gs<3,9*5>=0      Gs<7,9*8>=0     Gs<8,9*8>=0
  Gs<9,9*8>=0      Gs<1,9*2>=0     Gs<1,9*10>=0
  Gs<2,9*5>=0      Gs<4,9*5>=0     Gs<4,9*10>=0
  Gs<5,9*5>=0      Gs<5,9*10>=0
```

The designer learns that operator f_1 is used in activities x_{10}, x_5 and x_2. Operator f_6 is used in activities x_8 and x_3. At this point the designer wants to know information about the inputs to each activity.

```
9 INTER> vard

  Gd<9*-1,3>=0.27B Gd<9*-1,2>=0.28B Gd<9*-1,1>=0.23B
  Gd<9*5,2>=0.23B  Gd<9*3,1>=0.27B

  Nonbasic variables:

  Gd<9*8,2>=0      Gd<9*8,1>=0      Gd<9*3,2>=0
  Gd<9*10,1>=0     Gd<9*1,1>=0      Gd<9*2,1>=0
  Gd<9*4,1>=0
```

The **vard** command shows that the first input to activity x_3 (Gd<9*3,1> = 0.27) and the second output value of the problem (Gd<9*-1,2> = 0.28) have not been fixed to come from a storage element or an operator. From the information given by command 7, the designer knows that these two values have to come from storage elements (Gd<9*3,1> = Gd<9*-1,2> = 1) in order to use operators f_1 and f_6 respectively.

```
10 INTER> fixvar one Gd<9*-1,2>,Gd<9*3,1> ;

11 INTER> cont 1 ;

   2 more partial implementation
   1 more partial implementation
```

The designer wants to look at the input information for the activities again and the ordering information between activities.

```
12 INTER> vard

   Gd<9*-1,3>=0.28B Gd<9*-1,2>=1* Gd<9*-1,1>=0.15B
   Gd<9*3,1>=1*

   Nonbasic variables:

   Gd<9*-1,2>=0      Gd<9*5,2>=0    Gd<9*8,2>=0
   Gd<9*8,1>=0       Gd<9*3,1>=0    Gd<9*3,2>=0
   Gd<9*10,1>=0      Gd<9*1,1>=0    Gd<9*2,1>=0
   Gd<9*4,1>=0
13 INTER> vara

   Ga<9*10,9*2>=0.28B Ga<9*10,9*5>=0.85B

   Nonbasic variables:

   Ga<9*3,1,9*0,4>=0 Ga<9*6,1,9*0,4>=0 Ga<9*11,1,9*0,3>=0
   Ga<9*8,9*3>=0
```

The **vard** command shows that the first (Gd<9*-1,1> = 0.15) and third (Gd<9*-1,3> = 0.28) output values are not accessed from a storage element. The **vara** command shows that the order of activity x_{10} with with respect to activities x_2 and x_5 is not decided yet. Before making the next decision, the designer needs to recall the decisions that have been made and the information regarding the execution start time (T_{xs}) and execution release time (T_{xr}) of activities x_2, x_5 and x_{10}.

```
14 INTER> flist

   Gs<1,9*10>=1 Gs<1,9*2>=1 Gd<9*3,1>=1
   Gd<9*-1,2>=1

15 INTER> time Txs,Txr x2,x5,x10 ;

   time information for activity : 2
   Txs<9*2>      = 0.000000B
   Txr<9*2>      = 198.000000B

   time information for activity : 5
   Txs<9*5>      = 400.000000B
   Txr<9*5>      = 598.000000B

   time information for activity : 10
   Txs<9*10>     = 0.000000B
   Txr<9*10>     = 502.000000B
16 INTER> vars

   Gs<6,9*3>=1.00B  Gs<6,9*8>=1.00B Gs<8,9*8>=0.00B
   Gs<1,9*2>=1*     Gs<1,9*5>=1.00B Gs<1,9*10>=1*
   Gs<2,9*10>=0.00B Gs<4,9*2>=0.00B

   Nonbasic variables:

   Gs<3,9*5>=0      Gs<7,9*8>=0     Gs<8,9*3>=0
   Gs<9,9*8>=0      Gs<1,9*2>=0     Gs<1,9*10>=0
   Gs<2,9*5>=0      Gs<4,9*5>=0     Gs<4,9*10>=0
   Gs<5,9*5>=0      Gs<5,9*10>=0
```

The vars command shows that currently operator f_1 is shared among activities x_2, x_5 and x_{10}. From command 15, the designer decides that activity x_2 will start before activity x_{10} (Ga<9*10,9*2> = 0), and x_{10} will start before x_5 (Ga<9*10,9*5> = 1). Because of this ordering assumption, the third output value might need a storage element to store it (Gd<9*-1,3> = 1).

```
17 INTER> fixvar one Ga<9*10,9*5>,Gd<9*-1,3> ;

18 INTER> fixvar zero Ga<9*10,9*2> ;

19 INTER> cont 1 ;

   3 more partial implementation
   2 more partial implementation
   1 more partial implementation
```

The designer is interested in looking at the operator mapping information and ordering information.

```
20 INTER> vars

   Gs<6,9*3>=1.00B Gs<6,9*8>=1.00B  Gs<8,9*8>=0.00B
   Gs<1,9*2>=1*    Gs<1,9*5>=0.99B  Gs<1,9*10>=1*
   Gs<2,9*5>=0.01B Gs<2,9*10>=0.00B Gs<4,9*2>=0.00B

   Nonbasic variables:

   Gs<3,9*5>=0     Gs<7,9*8>=0      Gs<8,9*3>=0
   Gs<9,9*8>=0     Gs<1,9*2>=0      Gs<1,9*10>=0
   Gs<4,9*5>=0     Gs<4,9*10>=0     Gs<5,9*5>=0
   Gs<5,9*10>=0
21 INTER> vara

   Ga<9*8,9*3>=0.99B Ga<9*10,9*2>=0* Ga<9*10,9*5>=1*

   Nonbasic variables:

   Ga<9*3,1,9*0,4>=0 Ga<9*6,1,9*0,4>=0 Ga<9*11,1,9*0,3>=0
   Ga<9*10,9*2>=0    Ga<9*10,9*5>=0
```

Before making the next decision, the designer wants to find out timing information for some of the activities.

```
22 INTER> time Txs,Txr x2,x3,,x5,x8,x10 ;

   time information for activity : 2
   Txs<9*2>      = 0.000000B
   Txr<9*2>      = 188.000000B

   time information for activity : 3
   Txs<9*3>      = 200.000000B
   Txr<9*3>      = 695.000000B

   time information for activity : 5
   Txs<9*5>      = 380.192857B
   Txr<9*5>      = 700.000000B

   time information for activity : 8
   Txs<9*8>      = 0.000000B
   Txr<9*8>      = 205.000000B

   time information for activity : 10
   Txs<9*10>     = 188.000000B
   Txr<9*10>     = 386.000000B
```

The last three decisions have caused a change in the assignment of operator f_1 to activity x_5 (from

1.00 to 0.99). The designer decides to make sure that it is used there (Gs<1,9*5> = 1). From the timing information the designer decides to have activity x_8 start before activity x_3 (Ga<9*8,9*3> = 1).

```
23 INTER> fixvar one Gs<1,9*5>,Ga<9*8,9*3> ;

24 INTER> cont 1 ;

   2 more partial implementation
   1 more partial implementation
```

At this point the designer wants to look at the component mapping information.

```
25 INTER> vars

   Gs<6,9*3>=1.00B Gs<6,9*8>=1.00B  Gs<8,9*8>=0.00B
   Gs<1,9*2>=1*    Gs<1,9*5>=1*     Gs<1,9*10>=1*
   Gs<2,9*5>=0.00B Gs<2,9*10>=0.00B Gs<4,9*2>=0.00B

   Nonbasic variables:

   Gs<3,9*5>=0     Gs<7,9*8>=0      Gs<8,9*3>=0
   Gs<9,9*8>=0     Gs<1,9*2>=0      Gs<1,9*5>=0
   Gs<1,9*10>=0    Gs<4,9*5>=0      Gs<4,9*10>=0
   Gs<5,9*5>=0     Gs<5,9*10>=0
26 INTER> varr

   Gr<1,9*0,1>=0.00B  Gr<2,9*0,2>=0.00B  Gr<2,9*2,1>=0.60B
   Gr<2,9*3,1>=0.00B  Gr<2,9*6,1>=0.00B  Gr<4,9*0,4>=0.00B
   Gr<6,9*0,1>=0.00B  Gr<7,9*2,1>=0.40B  Gr<5,9*0,5>=0.00B
   Gr<5,9*9,1>=1.00B  Gr<10,9*9,1>=0.00B Gr<3,9*0,3>=0.00B
   Gr<3,9*11,1>=1.00B Gr<8,9*11,1>=0.00B

   Nonbasic variables:

   Gr<1,9*6,1>=0      Gr<4,9*3,1>=0      Gr<4,9*6,1>=0
   Gr<6,9*6,1>=0      Gr<7,9*0,2>=0      Gr<7,9*3,1>=0
   Gr<7,9*6,1>=0      Gr<9,9*0,4>=0      Gr<9,9*3,1>=0
   Gr<9,9*6,1>=0      Gr<10,9*0,5>=0     Gr<8,9*0,3>=0
```

The designer has in mind to assign a storage element to the output value of activity x_2. The **varr** command shows that the output value of activity x_2 can be stored in storage elements s_2 and s_7. Since s_2 is cheaper than s_7, the designer decides to use s_2 to store the value.

```
27 INTER> fixvar one Gr<2,9*2,1> ;

28 INTER> cont 1 ;
```

At this point, the designer wants to examine the values of all decision variables.

```
29 INTER> vars

   Gs<6,9*3>=1.00B Gs<6,9*8>=1.00B  Gs<8,9*8>=0.00B
   Gs<1,9*2>=1*    Gs<1,9*5>=1*     Gs<1,9*10>=1*
   Gs<2,9*5>=0.00B Gs<2,9*10>=0.00B Gs<4,9*2>=0.00B

   Nonbasic variables:

   Gs<3,9*5>=0     Gs<6,9*8>=0      Gs<7,9*8>=0
   Gs<9,9*8>=0     Gs<1,9*2>=0      Gs<1,9*5>=0
   Gs<1,9*10>=0    Gs<4,9*5>=0      Gs<4,9*10>=0
   Gs<5,9*5>=0     Gs<5,9*10>=0
30 INTER> varr

   Gr<1,9*0,1>=0.00B  Gr<2,9*0,2>=0.00B  Gr<2,9*2,1>=1*
   Gr<2,9*3,1>=0.00B  Gr<2,9*6,1>=0.00B  Gr<4,9*0,4>=0.00B
   Gr<6,9*0,1>=0.00B  Gr<7,9*2,1>=0.00B  Gr<5,9*0,5>=0.00B
   Gr<5,9*9,1>=1.00B  Gr<10,9*9,1>=0.00B Gr<3,9*0,3>=0.00B
   Gr<3,9*11,1>=1.00B Gr<8,9*11,1>=0.00B

   Nonbasic variables:

   Gr<1,9*6,1>=0      Gr<2,9*2,1>=0      Gr<4,9*3,1>=0
   Gr<4,9*6,1>=0      Gr<6,9*6,1>=0      Gr<7,9*0,2>=0
   Gr<7,9*3,1>=0      Gr<7,9*6,1>=0      Gr<9,9*0,4>=0
   Gr<9,9*3,1>=0      Gr<9,9*6,1>=0      Gr<10,9*0,5>=0
   Gr<8,9*0,3>=0
31 INTER> vara

   Ga<9*8,9*3>=1* Ga<9*10,9*2>=0* Ga<9*10,9*5>=1*

   Nonbasic variables:

   Ga<9*3,1,9*0,4>=0 Ga<9*6,1,9*0,4>=0 Ga<9*11,1,9*0,3>=0
   Ga<9*8,9*3>=0     Ga<9*10,9*2>=0    Ga<9*10,9*5>=0
```

```
32 INTER> vard

   Gd<9*-1,3>=1* Gd<9*-1,2>=1* Gd<9*3,1>=1*

   Nonbasic variables:

   Gd<9*-1,3>=0  Gd<9*-1,2>=0  Gd<9*-1,1>=0
   Gd<9*5,2>=0   Gd<9*8,2>=0   Gd<9*8,1>=0
   Gd<9*3,1>=0   Gd<9*3,2>=0   Gd<9*10,1>=0
   Gd<9*1,1>=0   Gd<9*2,1>=0   Gd<9*4,1>=0
33 INTER> varb

   Gb<8>=0.00B Gb<5>=0.00B Gb<4>=0.00B
   Gb<2>=0.00B Gb<1>=1.00B

   Nonbasic variables:

   Gb<6>=0

34 INTER> varc

   Gc<9>=0.00  Gc<7>=0.00 Gc<5>=0.50B
   Gc<4>=0.00  Gc<3>=0.50 Gc<2>=0.25B
   Gc<1>=0.00B

   Nonbasic variables:

   Gc<10>=0    Gc<8>=0    Gc<6>=0
```

From the **varr** command the designer knows that storage elements s_1, s_3, and s_5 are used in the current design. The designer then decides to have decision variables Gc<3>, Gc<2> and Gc<5> fixed to one.

```
36 INTER> fixvar one Gc<2>,Gc<3>,Gc<5> ;

37 INTER> cont 1 ;

   1 more partial implementation

   SUBPROBLEM FATHOMED IN INTGTR,
   CURBND = -47.03417 BEST = -47.03417
```

After obtaining the first complete design the designer lets DPSIS do the rest of the design.

```
36 INTER> cont 200 ;
   200 more partial implementation
   •
   •
   138 more partial implementation
   SUBPROBLEM FATHOMED DUE TO INFEASIBILITY
   last partial implementation report
   no active subproblem, therefore DPSIS stop.
   statistical reports
   % (back to unix)
```

This sample session shows that the designer can use the information provided by DPSIS to guide BandBX to the first integer solution. Some of the information in the statistical report will be used for discussion in section 7.2.

Chapter 7
Observations and Conclusion

This chapter characterises DPSIS as described in the previous chapters and summarises the work done in this research.

7.1. Observations

In the previous chapter, we evaluated DPSIS using three examples. The CrissX example shows how DPSIS can help the designer make an initial set of trivial decisions. With this start, the designer hopes that BandBX will find a good integer solution earlier and start fathoming subproblems earlier. The Logic example shows how DPSIS allows the designer to select a subproblem from among the active subproblems. With designer involvement, some unprofitable subproblems may be identified earlier. These subproblems can be abandoned either temporarily or permanently. The Power example shows how the information provided by DPSIS's commands can help the designer to obtain an initial good complete design.

The example sessions reveal that DPSIS can translate information in a partial solution tableau (produced while solving the MILP problem) from its original mathematical formulation into a form which can be understood by a digital hardware designer, who can then use it to make design decisions.

7.2. Comparisons

As mentioned in the last chapter, we are going to use three pieces of information from the statistics report generated by BandBX.

Table 7-1 summarises the statistics for the three problems when using DPSIS during the design process. Table 7-2 shows the same set of statistics when DPSIS was not used. The first column of

the tables is the number of subproblems needed to explore all the alternative designs. The second column is the number of simplex pivots needed to explore all the alternative subproblems. The third column is the number of the first subproblem that gives the optimal solution.

	Number of Subproblems	Number of Pivots	Optimal Value at Subproblem
CrissX	156	3673	67
Logic	507	15934	220
Pow	122	5308	12

Table 7-1: Design with DPSIS guide

	Number of Subproblems	Number of Pivots	Optimal Value at Subproblem
CrissX	274	6165	222
Logic	no solution	no solution	no solution
Pow	472	17507	397

Table 7-2: Design with no guide

The Logic example in Table 7-2 does not have any statistical report. The reason is that, when BandBX ran this problem, it stopped at one of the subproblems due to round-off error. This indicates that a set of decision variables fixed in a particular order may lead a problem into round-off error. When the problem ran using DPSIS, the designer fixed certain decision variables. This resulted in an initial set of decisions that did not lead the problem into round-off error.

With these three examples, the number of subproblems needed to explore all the alternative subproblems using the branch-and-bound technique is less using DPSIS than not using it.

The third column of the two tables shows that the initial set of decisions made by the designer leads to an earlier optimal solution. With an earlier optimal solution, the designer can hope that more problems will be fathomed due to bounds.

Note that the results are not always this way. Since DPSIS is a man-machine interface and humans can make poor decisions, the problems will sometimes run worse when using DPSIS.

7.3. Further Research

We have implemented a list of basic commands that allow users to interact with the design process. There are many other commands that can be added. In the following, we list a few useful commands that can be implemented in the future to provide more facilities for the users.

- A command to list out all the decision variables that are not fixed (basic and non-basic). This will give the designer a complete picture of what decisions he still has to make.

- A command that obtains the greatest or smallest bound change between two subproblems and examines the decision made. This will help the designer to identify the significance of the decision.

- A command that looks at all the active subproblems which have a certain decision pattern along the path v_j to v_0. This will help the designer group subproblems that he/she feels would produce a good design or bad design.

- A command for removing active subproblems from the branch-and-bound tree so that they will not be searched in the future. For example, if the designer is content with the objective range, then he/she may use the command to delete some subproblems.

- A command that undoes the previous command. This gives the designer the option of testing variables. Also, the designer might make a wrong decision that causes the infeasibility of a subproblem, and the undo command will help him to remove the decision and redecide.

- A command to fix variables that are trivial to the designer. At a certain stage of the design, the designer might wish to fix an obvious value without having BandBX create another branch.

The last three commands can be combined in a more general form. At any stage of the design process the designer should be able to disable or enable the exploration of certain portions of the branch-and-bound tree. With this feature the designer would be able to abandon a subtree temporarily or permanently if he/she feels the previous decisions might lead to a poor design.

References

1. Barbacci, M., Barnes, G., Cattell, R., Siewiorek, D. The Symbolic Manipulation of Computer Descriptions ; The ISPS Computer Description Language. Dept. of Computer Science, Carnegie-Mellon University, Pittsburgh, Pa.,March, 1978.

2. Satoshi Goto. Automatic Data Path Synthesis. In *Design Methodologies*, North-Holland, New York, N.Y., 1985, Chap. 13, pp. 401-439.

3. Hafer, L. *Automated Data-Memory Synthesis : A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic*. Ph.D. Th., 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.

4. Hafer, L., Parker, A. "A Formal Method for the Specification, Analysis, and Design of Register-Transfer Level Digital Logic". *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-2*, 1 (January 1983), 4-18.

5. Hafer, L. *IDDMA User's Manual.* Simon Fraser University, Burnaby, B.C., 1987.

6. Hafer, L. Logic Synthesis by Mixed-Integer Linear Programming. Dept. of Computer Science, Simon Fraser University, Burnaby, B.C.,May, 1988.

7. Martin, C. *LINPRO: A Linear Programming Code for Experiments in Mathematical Programming.* Industrial and Systems Engineering Dept., Ohio State University, 1977.

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

9. McFarland, M. The Value Trace: A Data Base for Automated Digital Design. Master Th., Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa.,December 1978.

10. Shiv Prakash. Guiding Design Decisions in RT-level Logic Synthesis. Master Th., School of Computing Science, Simon Fraser University, Burnaby, BC.,March 1987.

11. Snow, E. *Automation of Module Set Independent Register Transfer Level Design.* Ph.D. Th., Dept. of Electrical Engineering, Carnegie-Mellon University, Pittsburgh, Pa., April 1978.

12. Snow, E., Siewiorek, D., Thomas, D. A Technology-Relative Computer Aided Design System: Abstract Representations, Transformations, and Design Tradeoffs. Design Automation Conference Proceedings no. 15, ACM SIGDA, IEEE Computer Society-DATC, June, 1978, pp. 220-226.

13. Wuyi Wu. Heuristic Bounds for Automated Logic Synthesis. Master Th., School of Computing Science, Simon Fraser University, Burnaby, BC.,January 1987.