74-14,648

BARBACCI, Mario Roberto, 1945-
AUTOMATED EXPLORATION OF THE DESIGN SPACE FOR REGISTER TRANSFER (RT) SYSTEMS.

Carnegie-Mellon University, Ph.D., 1973
Computer Science

University Microfilms, A XEROX Company, Ann Arbor, Michigan

THIS DISSERTATION HAS BEEN MICROFILMED EXACTLY AS RECEIVED.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Automated Exploration of the Design Space

For Register Transfer (RT) Systems

by

Mario R. Barbacci

Department of Computer Science
Carnegie-Mellon University
Pittsburgh, Pa. 15213
November 1973

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was supported by the National Science Foundation under Grant GJ 32758X.
This document has been approved for public release and sale; its distribution is unlimitted.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Carnegie-Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE Automated Exploration of the Design Space for Register Transfer Systems

PRESENTED BY Mario Roberto Barbacci

ACCEPTED BY THE DEPARTMENT OF Computer Science

C. Gordon Bell — MAJOR PROFESSOR — 9 Nov 1973 — DATE

J. F. Traub — DEPARTMENT HEAD — 11/28/74 — DATE

APPROVED BY THE COMMITTEE ON GRADUATE DEGREES

James S. Tanger — CHAIRMAN — 2/1/74 — DATE

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Abstract

Early computer aided design (CAD) systems have proved the feasibility and the advantages of the CAD approach for designing digital systems. They were limited, however, by practical considerations regarding the size of the systems that could be dealt with, the type of analysis that could be performed, the synthesis algorithms etc. These limitations existed mainly because of the low level (gates) of components used as building blocks in the designs being constructed. A feature lacking in existing CAD systems is the exploitation of alternative design implementations derived from an initial behavioral specification. The approach in this thesis is to start with a representation of the desired system in terms of behavioral specifications at the Register Transfer (RT) level, and produce alternative structural specifications at the RT level, in this case, in terms of modular components, the Register Transfer Modules (RTM) set (DEC PDP-16).

The behavioral description of the system to be designed is compiled into an internal representation (a graph model). A set of routines test and perform transformations on the graph, thus producing alternative control flow structures. These are parameterized according to cost and speed requirements and hence define a design space. Heuristic techniques are used to reduce the design space search process and the number of alternative solutions by implementing only worthwhile transformations, according to goals specified by the user as part of the input. Test algorithms, based on the analysis of control flow and data dependencies, detect

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

computations (represented by subgraphs) that can be performed in sequence, in parallel, or in some special combinations of both. Transformation algorithms reconnect the graph to reflect the new flow of computations. Since the conditions for the transformations depend upon the processor-memory structure of the building blocks, several PMS structures are examined and compared with regard to their effect on the basic set of conditions.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Acknowledgement

I wish to acknowledge the great help and encouragement provided by my advisor Professor Gordon Bell throughout my years at CMU. Dr. Daniel Siewiorek provided valuable advise during the development of the thesis. Drs. John Grason and Nico Habermann provided many comments and suggestions.

The environment in the Department of Computer Science at Carnegie-Mellon University had a great deal to do with the succesful completion of the thesis. I want to thank Drs. Peder Emstead, Anita Jones, and Michael Knudsen for the time they dedicated to reading the thesis and for their valuable comments. Ross Scroggs and Steve Goldman implemented the ISP compiler. Joe Newcomer was most helpful with his improvements of the XOFF text editor and the TIMER system. Roy Levin modified the SIX12 system to make it compatible with the timing routines.

Last but not the least, I want to thank my friends and colleagues Robert Chen, Wing-Hing Huen, Henry Goldberg, and Satish Rege for many stimulating discussions. The "Downstairs Lounge" crowd provided many hours of relaxation and partying, they were much needed.

Finally, not having a wife to blame (or thank) for the time it took to write the thesis, it remains as an open question whether or not she would have helped; this is a suggestion for future research.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Table of contents

		page
	Abstract	i
	Acknowledgements	iii
	Table of Contents	iv
I	Introduction	1
	I.1 Register Transfer Level	2
	I.1.1 RT and switching circuit levels	2
	I.1.2 RT and programming levels	3
	I.2 Register Transfer Design	6
	I.2.1 The Technology	7
	I.2.2 The Behavior	8
	I.3 Computer Aided Design	10
	I.4 The EXPL System – an introduction	12
II	The Graph Model	17
	II.1 The Elements	17
	II.1.1 Data Operations and Variable Sets	18
	II.1.2 Control Operations	20
	II.2 Flowcharts	22
	II.3 Dominance Relationships	24
	II.3.1 Non–concurrent Flowcharts	25
	II.3.2 Concurrent Flowcharts	27
	II.4 Variable Sets of a Subgraph	29
	II.5 Conditions for Parallel Processing	31
	II.6 PMS Configurations	34
III	RTM Design Trade–offs	47
	III.1 Register Transfer Modules (RTM)	47
	III.2 The Design Space	50
	III.3 The Transformations	52
	III.3.1 Rule SP	53
	III.3.2 Rule PS	58
	III.3.3 Rule SPS0	60
	III.3.3 Rule SPS1	61
	III.3.5 Rule SPS2	62
	III.3.6 Rule CK0	64
	III.3.7 Rule CK1	64
	iii.4 Completeness of the transformations	66
	III.5 RTM Design Evaluation	77
	III.5.1 Control Binding	78
	III.5.2 Bus Binding	79
	III.5.3 Data Binding	80

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

IV	The EXPL System	82
	IV.1 The ISP Compiler	82
	IV.2 Data Structures	86
	IV.3 System Organization	94
	IV.4 Heuristic Search	98
	IV.4.1 Design Space Exploration	98
	IV.4.2 Design Space Reduction	102
V	Experimental Results	104
	V.1 An RTM Multiplier	105
	V.1.1 Initial Solution	106
	V.1.2 Variations in the Basic Algorithm	110
	V.2 A Controller for a Conveyor-Bin System	116
	V.2.1 Initial Solution	117
	V.2.2 Alternative Implementations	119
	V.3 A Modified Conveyor-Bin System	121
	V.4 Timing Measures	126
VI	Conclusions	131
	VI.1 The Implementation	131
	VI.2 A Methodology for RT level Design	134
	VI.3 Extensions	137
	VI.4 Summary	139
References		141
Appendix 1	Register Transfer Modules	144
Appendix 2	Bus Identification - Example	154
Appendix 3	The Syntax of ISP	160

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

List of Figures

Figure	Title	Page
1.	Register Transfer Design System	12
2.	EXPL Block Diagram	13
3.	Classification of Variables	19
4.	Flowchart Elements	23
5.	Non-Concurrent Flowchart	26
6.	Concurrent Flowchart	28
7.	Subgraph	32
8.	Concurrent Computation - Scheme 1	33
9.	PMS Types	35
10.	Flow of Computations - Models A, B, and C	38
11.	Flow of Computations - Models B1 and C1	40
12.	Concurrent Computation - Scheme 2	42
13.	Concurrent Computation - Scheme 3	42
14.	Concurrent Computation - Scheme 4.	44
15.	Parallel Computation	45
16.	Time Estimation	51
17.	Rule SP - 1	53
18.	Rule SP - 2	55
19.	Rule SP - 3	56
20.	Rule PS	58
21.	Rule SPS0	60
22.	Rule SPS1	61
23.	Rule SPS2	63
24.	Rule CK0	65
25.	Rule CK1	65
26.	3-Block Computations - Shorthand	69
27.	3-Block Computations - Flowcharts	70
28.	3-Block Computations - Derivations	71
29.	3-Block Computations - Transition Matrix	72
30.	3-Block Computations - Constraints	73
31.	Constraint A => B	74
32.	Constraint A => B, A => C	75
33.	Constraint A///B	76
34.	Symbol Table	88
35.	Statement Table	88
36.	Block Table	90
37.	Connectivity Matrix	90
38.	Distance Matrix	90
39.	Interval Table	91
40.	Region Table	92
41.	Group Table	93

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

42.	Cases Table	94
43.	Design Space Reduction	101
44.	Multiplier Flowchart	107
45.	RTM Multiplier – Templates	108
46.	Multiplier Transformations – 1	111
47.	Multiplier Transformations – 2	112
48.	Multiplier Design Space	114
49.	Multiplier Design Space Exploration – Trace	115
50.	Conveyor–Bin Flowchart	118
51.	Conveyor–Bin Design Space	120
52.	Conveyor–Bin Design Space Exploration – Trace	122
53.	Modified Conveyor–Bin Design Space	124
54.	Modified Conveyor–Bin Design Space – Rule PS Only	127
55.	Groups vs. Blocks	129

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

INTRODUCTION

The research reported here is a case study in *Computer Aided Design.* Its purpose is to study a methodology for digital systems design at the *Register Transfer Level* that are considered optimal along a given dimension. The primary result is the implementation of a set of programs (the EXPL System), that translates an initial behavioral description of the system into alternative structural specifications from which the system can be built. For simplicity, the structural specifications are given in terms of a specific set of building blocks, the Register Transfer Modules (RTM) [DEC71,Bell72b]. EXPL is intended to be used as a tool for experimenting with various design strategies at the RT level.

The EXPL system in a nutshell: A behavioral description of a digital system, in ISP [Bell71a], is compiled into a graph representation. A set of routines test and perform transformations on the graph, from which alternative structural specifications of the digital system can be obtained. These alternatives are parameterized by their cost and time requirements. Heuristic routines are used to reduce the design space by exploring only those alternatives whose characteristics approach a set of goals specified by the user as part of the input.

Background: A computing system is composed of thousands of individually designed, built, and interconnected components. These components have gone through a series of technological improvements, from relays, to vacuum tubes, to solid state devices, to integrated circuits. A result of this development has been the increase in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

complexity and logical power of the elementary components, introducing a hierarchy of conceptual and physical building blocks out of which systems are built. Several levels of detail can be identified [Bell71a]:

PMS (System level): its elements are processors, memories, switches, peripheral units etc.

Programming level: the basic components are the interpretation cycle, the machine instructions, and the data operations (which are defined at the RT level). This level is associated only with a subset of the possible digital systems that can be built, namely, those that are capable of interpreting an instruction set. There are many applications of digital technology where such interpretation is not required, and for which there is no programming level.

Register Transfer level (Functional level): a combination of switching circuits is used to form registers, register transfers and other operations. This level has existed, at least conceptually, for quite some time and it owes its emergence as a full fledged level of design to the evolving technology, MSI and LSI.

Switching circuit level (Sequential and Combinational sublevels): the system structure is given by a collection of gates and flip-flops, and the behavior by a set of Boolean equations. This has been the traditional level of digital design.

Circuit level: gates are described as some interconnection of diodes, transistors, resistors, etc. according to electrical circuit laws.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This thesis is concerned mainly with design at the RT level. Particul mphasis is given to one set of components, the family of Register Transfer Modu s (RTM). Although the main objective is the study of general techniques, this specialization simplifies the problem and also provides a "real world" context for our work.

I.1 REGISTER TRANSFER LEVEL

As mentioned before, the RT primitives are made of switching circuits and we should point out the differences between these two levels that make RT a separate level of design. By the same token, since the primitives of the programming level are defined at the RT level, the analogies and differences between these two levels should be discussed.

I.1.1 RT and switching circuit levels.- Two things separate the RT level from the switching circuit level. The first is a functional difference. The systems at the switching circuit level are considered as networks of elements performing their activities continuously. At the RT level the elements are assumed to be going through periods of activity and rest. Clearly, the physical elements are always active. The abstraction merely consists of the specification, at different times, of the components were "interesting" activities are taking place, e.g.: storing new information in a register or changing the inputs to a functional unit. The second characteristic of the RT level is

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the appearance of "families" of building blocks out of which systems can be build, eliminating the need for a translation to the lower level. This is a structural difference.

1.1.2 RT and programming levels.- Although the programming level is not considered part of the realm of hardware design, there are strong analogies between the programming and RT levels.

Both the programming and RT levels are based on the view that the system under consideration is made of two distinct but interconnected entities: A ***data flow structure*** and a ***control flow structure.*** The data flow structure is made of data carriers and operators. The carriers are used to store and transmit the data. The data operators work on data types, composed of a value (meaning) and a representation (encoding). Data operators perform transformations on the data and produce new data. The control flow structure is made of those components that direct (evoke) the actions of the data flow structure, i.e. the data operators. Some control operations may use the results of previous data operations in the selection of alternative activities in the system. Other control operators are used to initiate concurrent activities and may be independent of the data flow structure. To provide means of synchronizing different activity paths, control operators are used to merge control flow paths.

The RT and the programming levels share another common characteristic, namely, the existence of an algorithm that describes the behavior of the system.

The algorithm, being a conceptual tool for describing a pattern of behavior must

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be translated into a structural representation that describes the interconnection of the data and control elements. This synthesis problem deals with the translation from a (source program) representation of the system into another (object program) representation, with the property that the two representations define the same functional operation, the original algorithm. The synthesis of programming languages is carried out by a compilation process into an object program in the so called machine language of the computer. The register transfer synthesis defines a network consisting of standard physical building blocks, a register transfer set. During the compilation process of a program the machine language instructions are produced according to a set of rules, namely, the semantics of the language. The equivalent of language semantics at the RT level is given by a set of interconnection rules. These rules are a fundamental property of what constitutes a register transfer set, where by an RT set we mean something beyond a catalog of integrated circuit chips.

A characteristic of this synthesis process is the existence of alternative object representations. These representations often confront the designer (programmer) with conflicting trade-offs among a set of required resources. The trade-offs are predicated on the existence of certain critical resources. The level of importance varies with the resource, and in practical applications there is no infinite supply of any of them. Machine language programs present a trade-off between the storage required to store the instructions and the data, and the time required to execute the instructions, including the time needed to fetch operands and store results. Register transfer systems present a similar set of trade-offs among resources: sharing data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

operators, merging control paths, computing in parallel and pipelines, etc. These different trade-offs can be classified according to objective functions: cost, speed, reliability. The main difference between the trade-offs at the programming and RT levels is the latter's ability to restructure the physical system. Programmers are usually faced with a fixed architecture.

The next section describes the different approaches that have been used to design and implement RT systems.

I.2 REGISTER TRANSFER DESIGN

From the time of the first stored program computer, a certain part of the digital system design has taken place at the RT level. This level of design has not, however, been fully exploited for several reasons:

- The technology did not exist:

  1) The principal method, microprogramming, could not be used until recently because we did not have the necessary memory technology.

  2) There were no register transfer primitives in the same sense that there are primitives for doing combinational and sequential circuit design.

- Behavioral specifications were not easy to provide

  1) Although RT languages have existed for some time their use has met with only minor success. Besides, logic designers have not been comfortable with programming.

  2) Formal models have been used mainly in theoretical studies, they are difficult to use as design documents. Formal designers' objectives are sometimes orthogonal to those of practical designers (components used, level of detail etc).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

– The level of design was not taught to any degree as evidenced by each of the books concerned with digital design. This is probably a consequence, not an antecedent, of the previous reasons.

I.2.1 The technology.– Three main approaches to this problem can be observed:

Conventional RT systems.– The designer translates into the RT structure the various data and control operations for carrying out the particular task. The main emphasis is to minimize the amount of components (cost) and to increase the rate of register transfers (speed). This is a hybrid technique, the specifications of the behavior and the structure being at different levels, the former at the RT level, the latter at the switching circuit level. Design in this fashion has some problems: a design is fundamentally a large network of conceptually small components (gates and flip–flops); the design task is tedious, time consuming and difficult to modify if new specifications are added; errors are easily introduced and difficult to detect and correct. Mechanical procedures to handle this process have been successful in relieving the designer of some of these repetitive tasks.

Microprogramming.– The objective here is to have a regular, simple, and programmable (modifiable) control structure to increase reliability, to defer design decisions until as late as possible, and to allow relatively complex instruction sets to be implemented easily. Although microprogramming has indeed proved successful in this task, it does not concern itself at all with the problem of the implementation of the data part. A microprogrammed machine is a relatively fixed RT structure and although it

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

presents solutions to some of the problems it only hides the design of the microprogrammed machine – a problem similar to the design of the original, stored program computer. Microprogramming, for all its value, is a difficult task. Few design techniques exist and most of its problems (and solutions) are reformulations of the state of the art of programming in the 50's, before the generalization of the use of high level languages [Rosi69].

Modular design.– A third method of constructing digital systems is with a fixed set of RT components from which RT systems are constructed using some interconnection schema. It has the advantage (over microprogramming) of providing flexibility both in the data and control parts. The disadvantage is the sacrifice of some efficiency in order to have flexibility and ease of interconnection. State of the art RT sets are cost effective only for small or special purpose systems [Bell72b,Clar67,Full73]. The main advantage of modular systems is the short design and debugging time. Modular design tends to be a one pass design process, where the translation of an algorithm from paper design to hardware, disregarding wiring errors, always produces a system which operates as specified.

I.2.2 The behavior.– Register Transfer languages have been used for some time. They tend to be used in informal descriptions since no general conventions for RT notations exist, and different languages address themselves to different technologies, timing methods, levels of detail, etc. RT languages are similar to most programming languages

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

since both carry register assignments. In fact, most RT languages are simple extensions of programming languages, mainly Algol 60. This simplistic approach has been useful in the past. With the current emphasis in the development and formalization of the RT level the requirements imposed on a useful RT language are more stringent. A useful language must be able to describe in a precise and concise manner any register transfer system, independently of the technology used, and at different levels of detail. Since such a notation must be used as a description tool, other considerations like readability, familiarity, and simplicity must be taken into account [Barb73].

Another approach to the description of RT systems is based on formal models, usually represented by directed graphs. Several such graph models have been proposed. Huen [Huen73] has proposed a taxonomy of such models. The main classification separates them into data flow models and control flow models.

Data flow models associate the processing of information with the nodes of the directed graph. The edges represent the flow of data and control information. The operations are initiated by the presence of such information on the edges leading to the node (processing element). In data flow models no specific storage is assigned to the data items. Storage capabilities, instead, are associated with the edges, these being considered as queues.

Control flow models associate the control operations with the nodes of the graph and the flow of control with the edges. The nodes generate signals to the (external)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

data operators, the nature of which is unspecified to the system. Control flows by means of signaling mechanisms, varying from model to model, but they are all based on the existence of some sort of "token" used to request or acknowledge the activation of the nodes.

I.3 COMPUTER AIDED DESIGN

Early attempts at design automation (DA) were directed towards a reduction in the cost and time of the design process itself [Breu72]. These objectives were accomplished by relieving engineers of repetitive time consuming tasks such as:

1) The generation of detailed design information.

2) The control of changes in the design documents.

3) The checking of systems for electrical, logical, and physical compatibility.

4) The generation of detailed manufacturing information.

This view of design automation limits itself to filling the gap between the low level design specifications and the manufacturing data. Behavioral specifications are in the form of Boolean equations and the systems translate them into an equivalent gate level specification. Most of the synthesis algorithms at this level deal with the problem of reduction or simplification of the Boolean equations. In other words, computer aided design has been so far limited to the gate level.

Recent efforts at design automation have been directed towards a system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

capable of accepting a high level behavioral description and translating it into an equivalent gate level structure. APDL [Darr69] and ALERT [Frie69] are two such systems, aimed towards the automation of conventional RT level design. A behavioral description in a high level programming language (Algol 60 and APL respectively) is translated into a detailed specification at the gate level. The system expands the behavioral specification by providing the details needed at the lower level. It is interesting to note that both systems carry their activities only up to certain point. The detailed physical implementation is left in both cases to a subsequent program (unspecified in APDL, the IBM automation system in ALERT). DCDS [Pota69] takes a different approach, closer to the microprogramming style of design. Behavioral specifications of the data components are used only for simulation purposes while the behavior of the control part is translated into the specifications of a microprogramming control unit.

A more ambitious goal in design automation would be to allow the specification not only of the behavior, as in current systems, but also of the building blocks. This problem is similar to the compilation of programs for different object machines. Such a system would take as inputs a specification of the behavior in some high-level RT language, together with the specifications of the RT level primitives, and produce as output the wiring lists needed to implement the system. The design system would generate, as by-products, test information and various measurements on the design. Simulation is sometimes a useful operation and the system should provide this capability. Figure 1, reproduced from [Bell71b] shows the organization of a system that would attack this problem.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

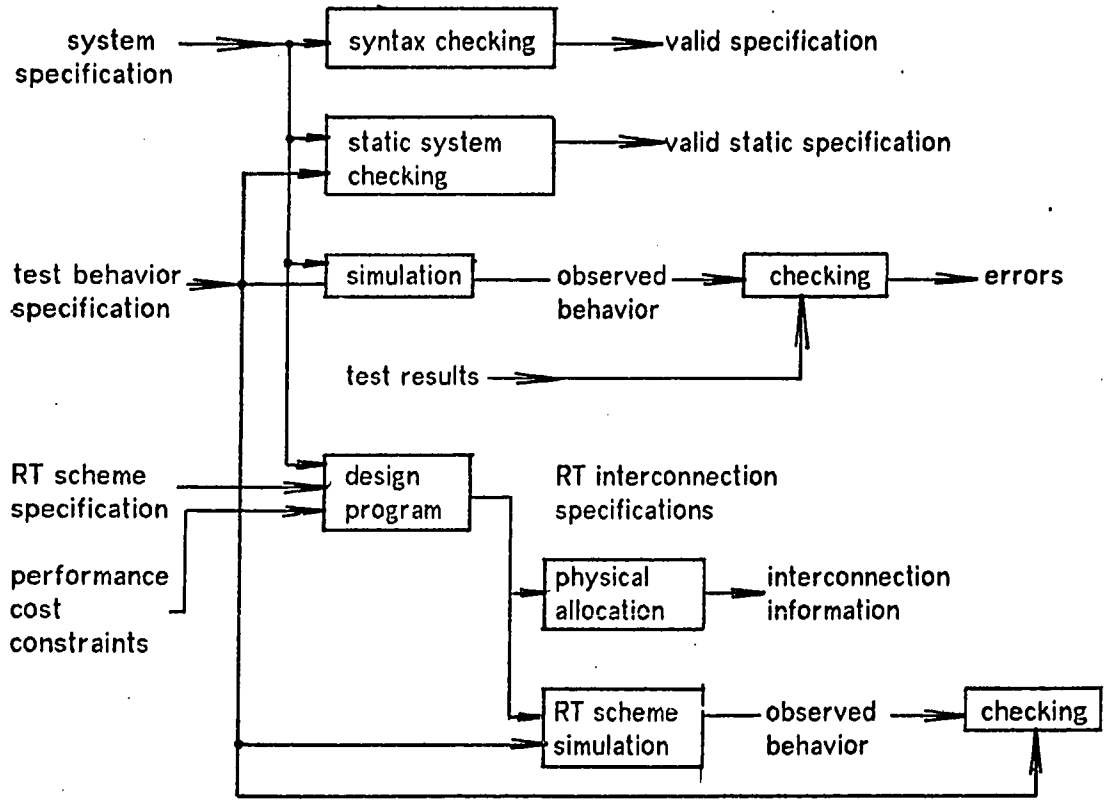


Figure 1. Register Transfer Design System

I.4 THE EXPL SYSTEM – AN INTRODUCTION

A feature lacking in existing DA systems is the exploitation of alternative implementations derived from the initial behavioral specifications. The emphasis of our approach is to start with a representation of the desired object system in terms of behavioral specifications at the RT level, and produce alternative specifications of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

control structure in terms of modular components also at the RT level, in our case, the RTM set. This synthesis is independent of the actual data on which the object system will perform. Thus, the transformations applied to obtain alternative structures are limited to those that are data independent.

Figure 2 shows a block diagram of the RT automatic design system, EXPL, that was implemented by the author. It could be used as the skeleton of the more ambitious system sketched before.

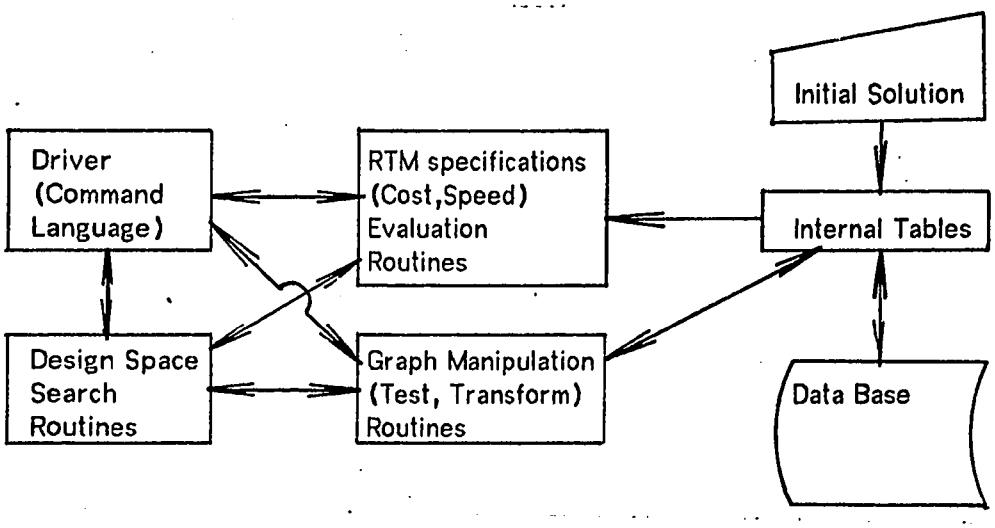


Figure 2. EXPL Block Diagram

The system produces alternative implementations, represented by graphs, of a given behavioral description. Different structures (implementations) are parameterized

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

according to cost and speed and these are the only two factors used to evaluate alternative structures.

The system is organized as a set of modules that perform specialized functions. EXPL in its present implementation is used as an interactive program interpreting commands typed by the user on a time-sharing terminal. The commands perform the following types of functions:

- The user can provide the specifications of the behavior of the digital system being designed. These specifications are in the form of a flowchart (obtained by compilation of an ISP program) providing an initial solution from which alternative flowcharts can be obtained.
- The user can interrogate the system about possible modifications in the control flow of the flowchart and the trade-offs in cost and speed that these modifications would bring about.
- The user can command the system to perform a transformation, thus obtaining an alternative flowchart, usually with different values in cost and speed from the previous flowchart.
- The user can store and retrieve flowcharts from a data base, thus allowing him to explore the effect of sequences of transformations and to back up to an earlier design (flowchart) if a given sequence of transformations does not yield an acceptable design.
- The user can provide goals and trade-offs between the cost and time requirements of a design and then command the system to perform an automated exploration of the design space obtained by applications of the transformations defined in the system.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The following paragraphs describe the components shown in Fig. 2. The arrows in the figure indicate interactions between components.

Initial solution: the initial solution is obtained by compiling the ISP description into the internal representation. This process, similar to the more conventional computer aided design approach, provides us with a "seed" structure from which the alternative structures are generated.

Internal tables: these are a set of internal data structures used to represent the behavioral description and the alternative structures. Only one set of these is kept in core at a given time.

Data base: graphs, representing alternative structures are kept in disk files. Fetch and store routines are provided in the driver module.

Structure manipulation: a set of routines that test for and perform transformations in the internal tables, thus obtaining alternative structures.

RTM specifications: a set of routines that measure the cost and time characteristics of an alternative structure. Although RTM independence was not a major requirement, this objective is still reasonably achievable. All the RTM specific information is contained here, and in principle it is easy to obtain module independence by just replacing this set of routines (minor modification would have to be made on the definition of the data structures to make them module independent).

Heuristic search: a set of routines that exercise the components described above

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in order to explore the design space (limited in our case to two dimensions: cost and speed). These routines make use of some simple heuristic techniques to reduce the search process by implementing only worthwhile alternatives.

Driver: a set of routines, driven by a command language, that allow the user to exercise, from a terminal, all the facilities of the system.

A more detailed description of the system is postponed until chapter 4. The present introductory remarks about EXPL are given so that the reader can have a mental picture of the system during the intervening chapters.

Chapter 2 deals with the graph model used to represent the digital systems, the transformations that can be applied to the graph, and the conditions under which a given transformation is feasible or desirable. Chapter 3 describes the family of modules used by the EXPL system. This description is necessarily brief since the modules have been reported elsewhere [DEC71,Bell72b]. This chapter also presents the design trade-offs between cost and speed of RTM systems and the graph transformations applicable to RTM flowcharts. Chapter 4 describes EXPL and the heuristic techniques used in the exploration of the design space. Experimental results are described in chapter 5. Finally, some conclusions are drawn from the design approach based on structural reconfigurations and its implications in hardware design.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

II. THE GRAPH MODEL

This chapter introduces the graph model used to represent the behavior of a digital system. Sections II.1 through II.4 present the basic elements of the model, the data and control operations, the variable sets used to characterize the blocks and subgraphs, and the dominance relationships between elements of a flowchart. The second half of the chapter, sections II.5 and II.6, makes use of these elements to describe several schemas and conditions under which the control flow of a program can be altered to make more efficient use of multiple processing units. The basic conditions for the alteration of the control flow depend upon the processor–memory structure assumed for the processing units. Section II.6 presents several of these PMS configurations and their effect on the conditions.

II.1 – THE ELEMENTS

For our purposes we use a model that combines characteristics of both control and data flow models. This model is a directed graph, similar to a program flowchart, where the nodes represent the basic blocks of the program and the edges represent the flow of control. Thus, the nodes contain both the data and control operations.

A ***basic block*** is a linear sequence of operations having one ***entry point*** or ***head*** (the first operation in the sequence), and one ***exit point*** or ***tail*** (the last operation executed). It may have many predecessors and many successors, and may even be its own successor. An ***entry block*** is a block with no predecessors in the graph. An ***exit block*** is a block with no successors in the graph.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

II.1.1 Data operations and variable sets.- Memory locations (variables) can be used in two ways by a data operation. A variable can be fetched and used as a source operand; the variable is not modified by this type of usage. On the other hand, a variable can be modified as a result of the operation; the original value of the variable is destroyed. Since we are dealing with arbitrary programs we will impose the restriction that no information about what the program actually does is required. With this restriction, the nature of the data operations can be ignored. We are interested only in the sets of variables involved and in whether a variable is used and/or modified by the data operations.

Destination set (D): is the set of variables modified or stored into during a computation, regardless of whether the information being stored is identical or not to the previous value of the variable.

Source set (S): is the set of variables used (read from) during a computation and may or may not be modified later.

Required set (R): is the subset of *source* variables whose first use is as a source variable and may or may not be modified at a later time.

The variable sets defined above, for the basic blocks, will be generalized in section II.4 to subgraphs and used later, in section II.5, to define the conditions under which a program can be executed more efficiently by using multiple processing units.

Several relationships between the R,S, and D sets can be observed from the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

definitions. A variable may belong to more than one of these sets; therefore the sets are not necessarily disjoint or nested. These relationships can be visualized in Fig. 3. The figure is organized as a truth table, where each row represent one of the possible cases by which a variable, used in a block, can be characterized.

The presence (absence) of a variable in one of the sets is represented by a 1 (0) under the corresponding set name. Variables can be classified according to the sets to which they belong. There are 8 such possible classes or derived sets, although some of them are impossible according to our definition.

R	S	D	
0	0	0	impossible, a variable must belong to a set
0	0	1	variables used only as destinations
0	1	0	impossible, a source variable that is not computed must be required
0	1	1	variables first modified and later used
1	0	0	impossible, the required set is a subset of the source set
1	0	1	impossible, the required set is a subset of the source set
1	1	0	variables used only as sources
1	1	1	variables first used as source and later modified

Figure 3 – Classification of variables

The following example will clarify these ideas. Assume the following sequence of operations:

1) X←Z
2) Z←X+Y
3) W←X−Y

From the definitions we can readily find the *source* and *destination* sets. These

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

are the variables used in the right and left hand sides of the assignments, respectively. Thus, the *source* and *destination* sets are:

D:= {X,Z,W}, S:= {Z,X,Y}

The *required* set is obtained by analysis of the sequence of data operations. Since the *source* variables of the first statement are not previously computed in the sequence, they belong in the *required* set, in this example the variable Z. In the second assignment, X is a *source* variable but since it was computed in the previous line, it is not in the *required* set. Y is *required* since its first use (line 2) is as a *source*. This analysis is carried out for each successive statement in the sequence, until all variables have been accounted for:

R:= {Z,Y}

II.1.2 Control operations.- Four primitive control operations are used in the graph model:

- *Branch.*- This is a generalized decision operation. It activates 1 out of n possible control paths depending on a condition. In the simplest case, the decision is based on a test of a Boolean variable. The more general case involves a decoding network that activates 1 out of n paths depending on a value stored in a register.

- *Diverge.*- This control operation activates a number of concurrent control paths. It does not depend on any decision mechanism.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

– ***Serial Merge.***– This control operation merges several mutually exclusive control paths into one. It activates the succeeding control path when it receives an activation signal from one of the incoming control paths.

– ***Parallel Merge.***– This control operation merges several concurrent control paths. It activates the succeeding control path when it receives an activation signal from all the incoming control paths.

The generality of these four control operations is born out by the many applications in which they appear. The *branch* appears as the If–Then–Else operation in programming languages, the Kbranch module in RTMs, the DECISION module in Macromodules [Clar67]. The *diverge* and *parallel merge* were first proposed as the FORK/JOINT operators by Conway [Conw63], they are implemented as the Kdiverge/Kparallel.merge in RTM, the CONTROL.BRANCH/ RENDEZVOUS in Macromodules, the DO–TOGETHER/HOLD defined by Opler [Ople65]. The *serial merge* operator is usually implemented as a label in programming languages, as the Kserial.merge in RTM, and as the MERGE module in Macromodules.

These control operators are sufficient to describe sequential and concurrent operations. No provision is made for the representation of control operations involving monitoring and synchronization functions [Fish70]. The reason for this omission is, as will be made apparent in later chapters, that we are interested in a particular set of graph transformations that depend only on the four chosen control operators.

Other control operations, of a simpler nature, are used in real RT sets to direct

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the sequencing of the data operators (e.g. SEQUENCE in [Denn70], Kevoke in RTM). These operators are implied only, since their use would impose a dependency on a particular set of building blocks.

II.2 – FLOWCHARTS

A *Program Control Flow Chart* (Flowchart) is a directed graph that contains a unique entry node, a unique exit node, and in which the nodes represent ***basic blocks*** and the edges represent control flow paths. Since control flow is affected only by the control operations, these can occur only as ***head*** or ***tail*** operations. This feature allows the analysis of control flow by considering only the ***head*** and ***tail*** operations of the ***blocks***, ignoring the data operations.

Two special control operators are used here to provide the initialization and termination of all activities in the system. Thus we have a ***start*** operation that has no preceeding actions in the system and constitutes a unique entry point to the graph. A ***stop*** operation, with no succeeding actions in the system, constitutes a unique exit point. Flowcharts will be represented by circles (***blocks***) and directed lines (edges) connecting the ***blocks***.

Figure 4 shows the basic elements of a flowchart. Control operations are made explicit, when necessary, by drawing them outside their ***blocks***, indicated by broken lines. Associated with each ***block*** we have three (possibly empty) variable sets, the ***destination***, ***source***, and ***required*** sets described before.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

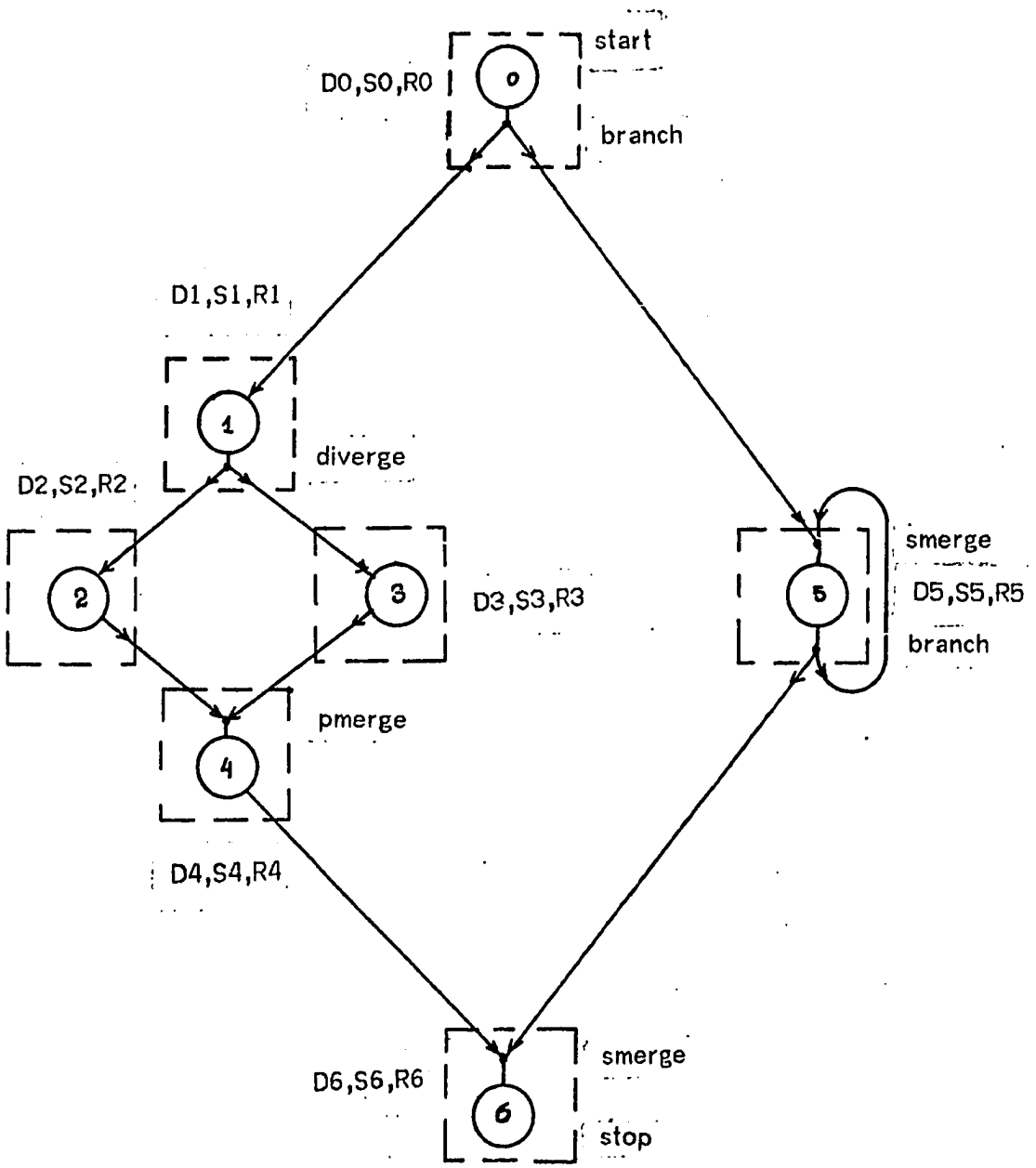


Figure 4. Flowchart Elements

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

A *path* in a directed graph is a directed subgraph represented by an ordered set (list) of *blocks* B1,...,Bn where Bi+1 is an immediate successor of *block* Bi.

A *block* Bj is a *successor* of *block* Bi if there exists a path Bi,...,Bj. Bi is said to be a *predecessor* of Bj.

The *length* of a path is the number of edges in the sequence. The shortest distance between two *blocks* Bp and Bq is the length of the shortest path Bp,...,Bq.

II.3 – DOMINANCE RELATIONSHIPS

This section defines the *dominance* relationships among blocks. These relationships will be used in section II.4 to generalize the variable sets (*destination*, *source*, and *required*) that were previously defined for the blocks, to subgraphs of arbitrary size.

Define the block containing the start operation as the start or entry block Be. A block, Bi, is said to *predominate* a block, Bk, if Bi is on every path from Be to Bk. The predominator set of block Bi is defined as:

$$BDi = \{Bj \mid Bj \text{ predominates } Bi\}$$

The immediate predominator of block Bk is the predominator which is "closest" (i.e. with the shortest distance) to Bk.

A block, Bi, is said to *postdominate* a block, Bk, if Bi is on every path from Bk to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the exit block, Bx. Similarly, we can define the postdominator set and the immediate postdominator for each block Bk ≠ Bx.

An *articulation* block in a directed graph is a block which lies on every Be,....,Bx path. The postdominators of Be, together with Be constitute the articulation blocks of the graph.

The definition of block dominance presented above is based on the connectivity of the flowchart. It will now be refined to distinguish between dominance in non-concurrent flowcharts (flowcharts were no concurrent operations can take place) and concurrent flowcharts (were concurrent activities may take place).

II.3.1 Computational dominance in non-concurrent flowcharts.- The concept of dominance based on connectivity considerations can be applied to the computations themselves to define preceding and succeeding operations. A computation (data or control operation) in Block Bi is said to predominate a computation in block Bj if the computations in block Bi must be executed in order to arrive to (and execute) the computations in block Bj. Similarly, a computation in block Bk postdominates a computation in Block Bi if the computations in block Bk are executed whenever the computations in block Bi are executed. It can be shown that every block Bk ≠ Be has at least one predominator and exactly one immediate predominator, and that the set of predominators is strictly ordered by their distances from Be [Alle70]. Figure 5 shows an example of a non-concurrent flowchart. The pre(post)dominator sets for each

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

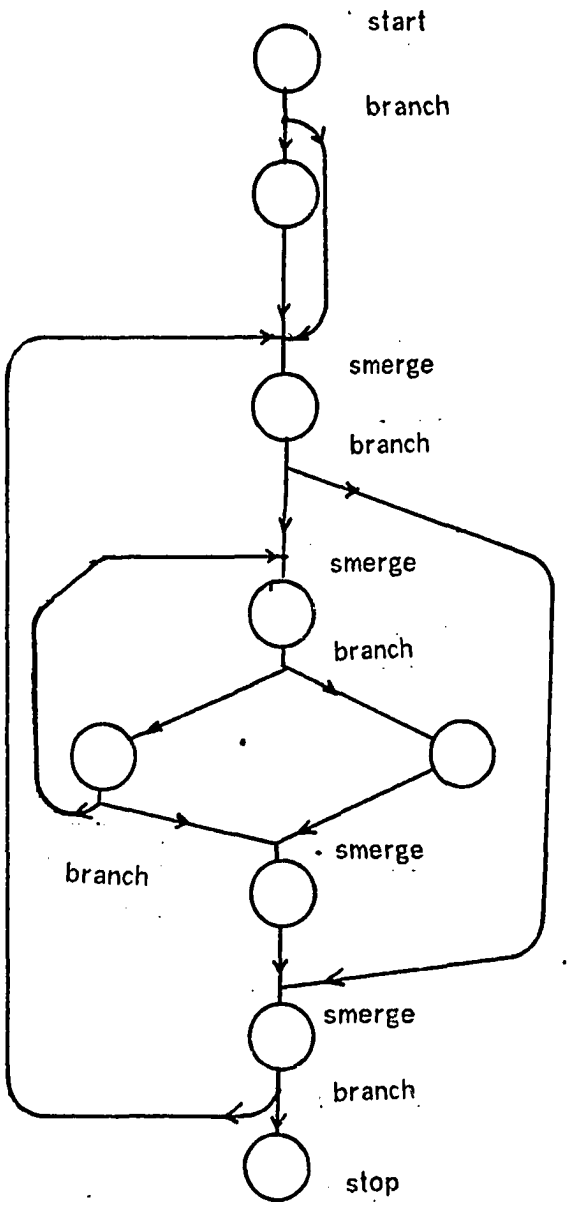


Figure 5. Non-Concurrent Flowchart

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

block are:

BLOCK	PREDOMINATOR SET	POSTDOMINATOR SET
0	---	{B2,B7,B8}
1	{B0}	{B2,B7,B8}
2	{B0}	{B7,B8}
3	{B0,B2}	{B6,B7,B8}
4	{B0,B2,B3}	{B6,B7,B8}
5	{B0,B2,B3}	{B6,B7,B8}
6	{B0,B2,B3}	{B7,B8}
7	{B0,B2}	{B8}
8	{B0,B2,B7}	---

The articulation blocks are {B0,B2,B7,B8}.

II.3.2 Computational dominance in concurrent flowcharts.- The concept of immediate dominance changes slightly. the computations in a block headed by a parallel merge operation are immediately predominated by all the computations in the immediate predecessor blocks. The computations in blocks with a diverge operation as tail are immediately postdominated by all the computations in the immediate successor blocks. As a consequence, we do not have necessarily unique immediate predominator and postdominator computations. It also follows that the computations in block Bi are (pre)postdominated by the computations in the (pre)postdominator blocks, that the dominator sets in the connectivity sense are a subset of the dominators in the computation sense, and that the (post)predominators are partially ordered by their distance to (Bx)Be. Figure 6 shows an example of a concurrent flowchart. The pre(post)dominator sets for each block are:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

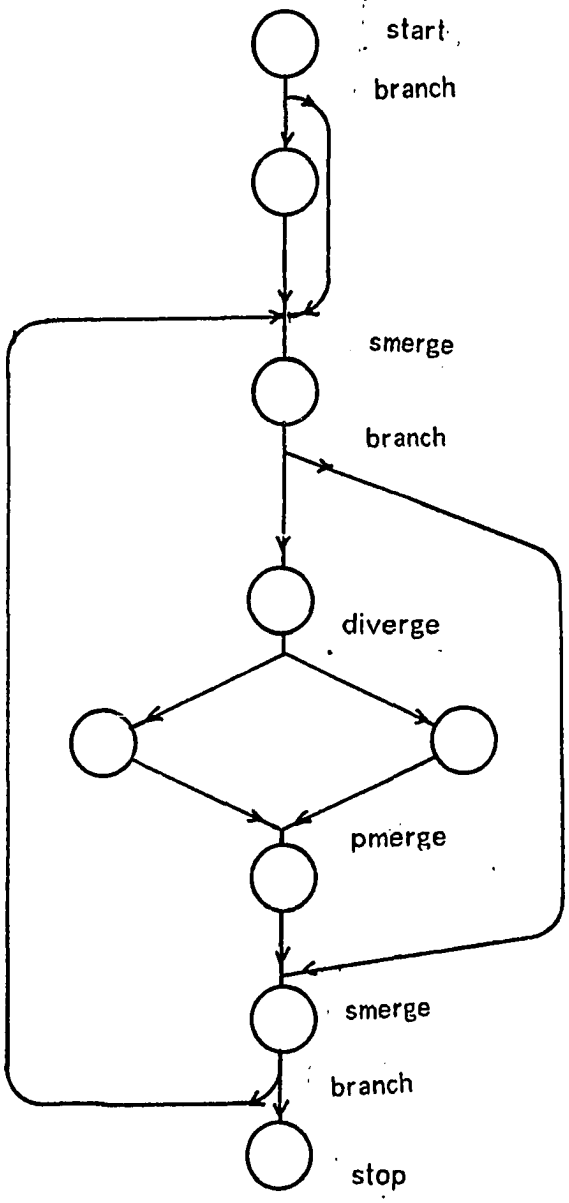


Figure 6. Concurrent Flowchart

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BLOCK	PREDOMINATOR SET	POSTDOMINATOR SET
0	---	{B2,B7,B8}
1	{B0}	{B2,B7,B8}
2	{B0}	{B7,B8}
3	{B0,B2}	{B4,B5,B6,B7,B8}
4	{B0,B2,B3}	{B6,B7,B8}
5	{B0,B2,B3}	{B6,B7,B8}
6	{B0,B2,B3,B4,B5}	{B7,B8}
7	{B0,B2}	{B8}
8	{B0,B2,B7}	---

The articulation nodes are {B0,B2,B7,B8}.

From now on, unless stated otherwise, the concept of dominance used will be that of "computational dominance".

II.4 – VARIABLE SETS OF A SUBGRAPH

The only subgraphs we are interested in are those with a single entry block, and the definitions used in this and future sections are predicated on such a unique entry block.

Given the source, Si, destination, Di and required, Ri sets corresponding to the blocks of a subgraph, we can define the source and destination sets for the subgraph as:

$S := U_i$ Si and $D := U_i$ Di, for all blocks Bi in the subgraph.

The definition of the required set for the subgraph is more complicated. A

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

variable that is in the required set of a block Bi may or may not be computed in another block Bj of the subgraph. Since no assumption can be made about the actual (exact) sequence of computations (the path followed to reach Bi), our definition will be based on a worst case situation.

A *local predominator set* for a block Bi in the subgraph is the subset of predominator blocks that belong to the subgraph. Since the subgraph has a unique entry block, Bh, it must be on every path from Be to Bi, therefore the local predominators are the predominators that lie between Bh and Bi, including Bh for Bi≠Bh. Bh has no local predominators.

The *reduced required set* R'i of block Bi is defined as:

R'i := Ri∩ ¬(U_K Dk), ∀k, where {Bk} are the local predominators of Bi. The reduced set R'i merely states that some variables required by Bi may be computed by some predominator block and we do not include them in the reduced set. Notice that the variables required by Bi (Ri) may be computed along many paths but that the only computations we can be sure of are those in the local predominator set.

The *subgraph required* set is defined as:

R := U_i R'i for all blocks Bi in the subgraph.

The required set of the subgraph represents the set of variables that are (in the worst case) used as sources before they are modified or defined. It represents the maximal amount of external information that the subgraph requires. R varies in size

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

from Rh (the required set of the entry block) to U_i Ri. According to the previous definitions, the subgraph enclosed by the dotted lines in Fig. 7 has the following variable sets:

S:=S2 U S3 U S4 U S5 U S6

D:=D2 U D3 U D4 U D5 U D6

R:=R'2 U R'3 U R'4 U R'5 U R'6

where the *reduced required* sets R'i are:

BLOCK	LOCAL PREDOMINATORS	REDUCED REQURED SET (R'i)
2	- - -	R2
3	{B2}	R3∩¬D2
4	{B2,B3}	R4∩¬(D2UD3)
5	{B2,B3}	R5∩¬(D2UD3)
6	{B2,B3}	R6∩¬(D2UD3)

The use of these variable sets will become evident in the following sections, where they are used to express the conditions for topological reorganization of a flowchart.

II.5 - CONDITIONS FOR PARALLEL PROCESSING

In this section we will consider the conditions under which a program, represented by its flowchart, can be executed more efficiently by detecting portions of the flowchart that can be executed concurrently by multiple processing units.

The basic conditions sufficient for independence were first formalized by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

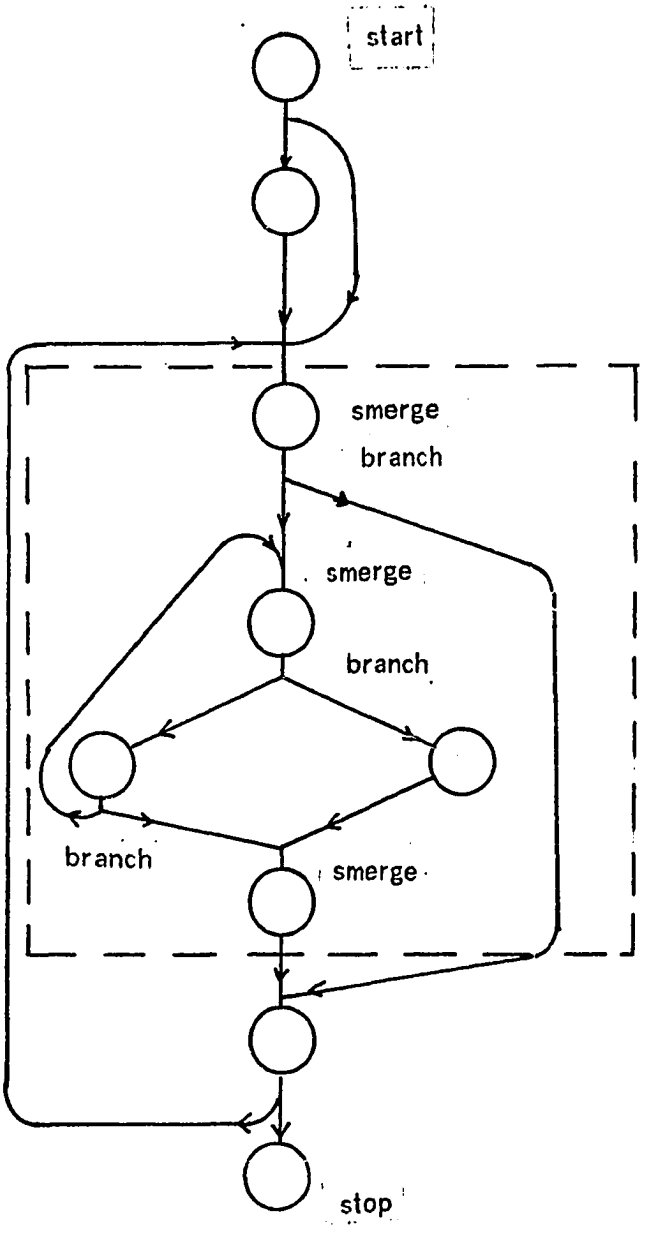


Figure 7. Subgraph

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bernstein [Bern66]. The main characteristic of these conditions is the use of data dependencies for the detection of parallelism. The conditions are expressed in terms of the variable sets (D,S, and R) presented earlier in the chapter.

Given a sequence of computations, P1, P2, P3 , where P3 represents the total portion of the flowchart following P2, the conditions for computing P1 and P2 concurrently (Fig. 8) are the following:

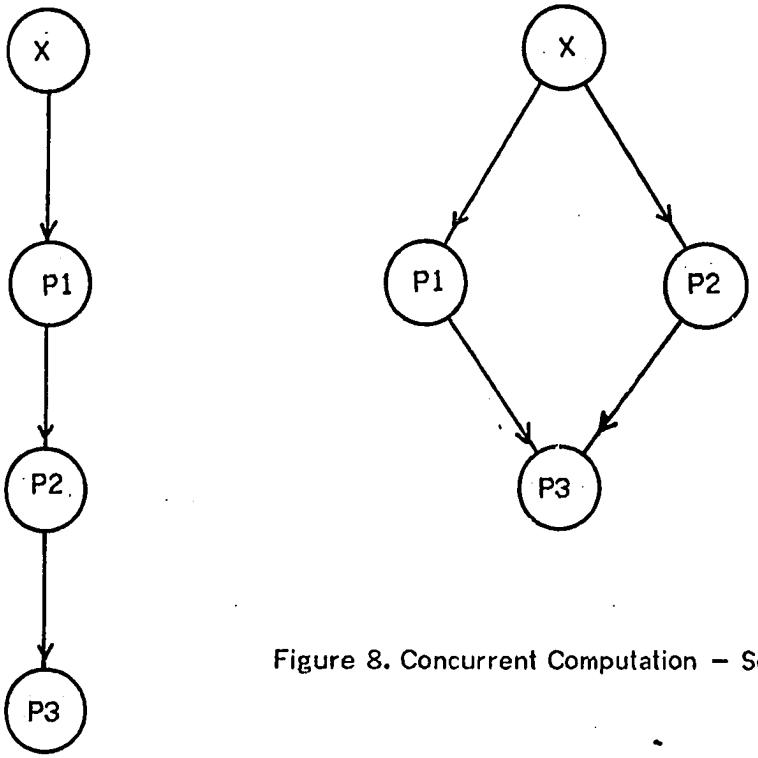


Figure 8. Concurrent Computation – Scheme 1

CONDITION	EXPRESSION
Ia	$R1 \cap D2 = 0$
Ib	$(S1 \cap \neg R1) \cap D2 = 0$
IIa	$R2 \cap D1 = 0$
IIb	$(S2 \cap \neg R2) \cap D1 = 0$
III	$D1 \cap D2 \cap R3 = 0$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The meaning of the conditions is the following:

Ia is required since the execution of P2 must not affect the initial values of the inputs for P1.

Ib is required since P2 must not modify partial results of P1.

IIa is required in order to execute P2 before P1 has been completed (i.e. concurrently with P1).

IIb is required to prevent P1 from modifying partial results computed by P2.

III is required to present (to P3) the same state regardless of the order in which the results where computed.

These conditions constitute a strong set of requirements. They can, under certain circumstances, be weakened by the use of appropriate processing units as will be shown next.

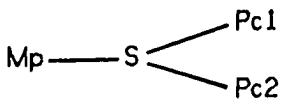
II.6 – PMS CONFIGURATIONS

As a consequence of the relationship between the conditions for concurrent execution and the variables used by the program, the conditions themselves depend on the Processor–Memory structure of the machine that executes the program. The current section examines several of these Processor–Memory–Switches (PMS) configurations and their effect on the conditions for concurrent computation. The first two models were examined by Bernstain [Bern66]. The last three models were developed by the author; they indicate how a given set of conditions can be satisfied

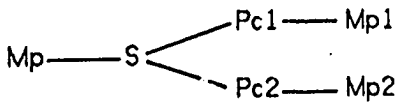
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

by less expensive PMS structures.

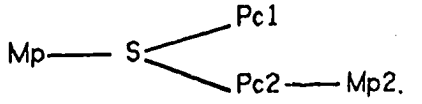
In model A (Fig 9.a) [Bern66], each processor communicates directly with a single large main memory. The important feature is that each processor can modify



(a) Model A



(b) Model B



(c) Model C

Figure 9. PMS Types

information which is to be used by the others.

In model B (Fig. 9.b) [Bern66], slave memories (buffers) have been added to the system. A processor can fetch information from main memory, but any information to be stored is put in its buffer. The buffer acts as intermediate storage between the processor and the main memory. When a processor needs some information it looks first in the associated buffer to see if the information has been stored there as a result

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

of a previous computation. If not, the data is obtained directly from main memory. When both processors have completed their tasks, the information in the slave memories is transfered to the appropriate locations in main memory in the proper order, namely, first Mp1 and then Mp2. Slave memories in this context are not to be construed as "cache" memories. Information is allocated in the slave memory only when it is created or modified by its associated processor. If the same variable is accessed more than once, it always comes from Mp and it is not copied into the slave memory. Furthermore, no consideration is given to relative memory speeds, locality of reference, accessing techniques, and other features of cache based memory systems.

Model C (Fig 9.c) differs from model B in that only one slave memory is used. Pc1 can fetch and modify variables directly in the main memory. Pc2 can only fetch variables from main memory and uses Mp2 as a buffer for computations (P2's ***destination*** set).

The effect of these models on the original set of requirements is the following:

Condition	Model A	Model B	Model C
Ia ($R1 \cap D2 = 0$)	required	not required	not required
Ib ($S1 \cap \neg R1 \cap D2 = 0$)	required	not required	not required
IIa ($D1 \cap R2 = 0$)	required	required	required
IIb ($D1 \cap S2 \cap \neg R2 = 0$)	required	not required	not required
III ($D1 \cap D2 \cap R3 = 0$)	required	not required	not required

Conditions Ia and Ib are not required in models B and C because the results of P2 (D2) are stored in the slave memory and can not, therefore, affect the variables used by P1 (S1).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Condition IIa is still required to preserve the correct behavior of P2.

Condition IIb is not required since the partial computations of P2 ($S2 \cap \neg R2$) are stored in the slave memory and can not be altered by P1.

Condition III is not required since now the destination sets are isolated and they are transferred back to Mp in the proper order: D1,D2 in model B, D2 in model C.

Several things can be observed from the table:

1) Models B and C present less strict requirements, thus allowing more flexibility.

2) Models B and C pay a price for their generality, both in components (the slave memories) and in time since the results must be stored in the main memory after both P1 and P2 are finished but before P3 can start.

Figure 10 shows the sequence of operations in the different models. P13 and P23 are the transfer of results from the slave memories Mp1 and Mp2, respectively, to the main memory (where they can be used by P3).

3) Models B and C present the same (minimal) set of constraints. Thus, if two computations P1 and P2 can be performed in parallel in a system with configuration B, they can likewise be performed in parallel in a system with configuration C, with less overhead.

The presence of alternative computations (due to conditional branches) introduces some difficulty since it is not known a priori (by static analysis of the flowchart) whether some computations that create or modify data will be performed or

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

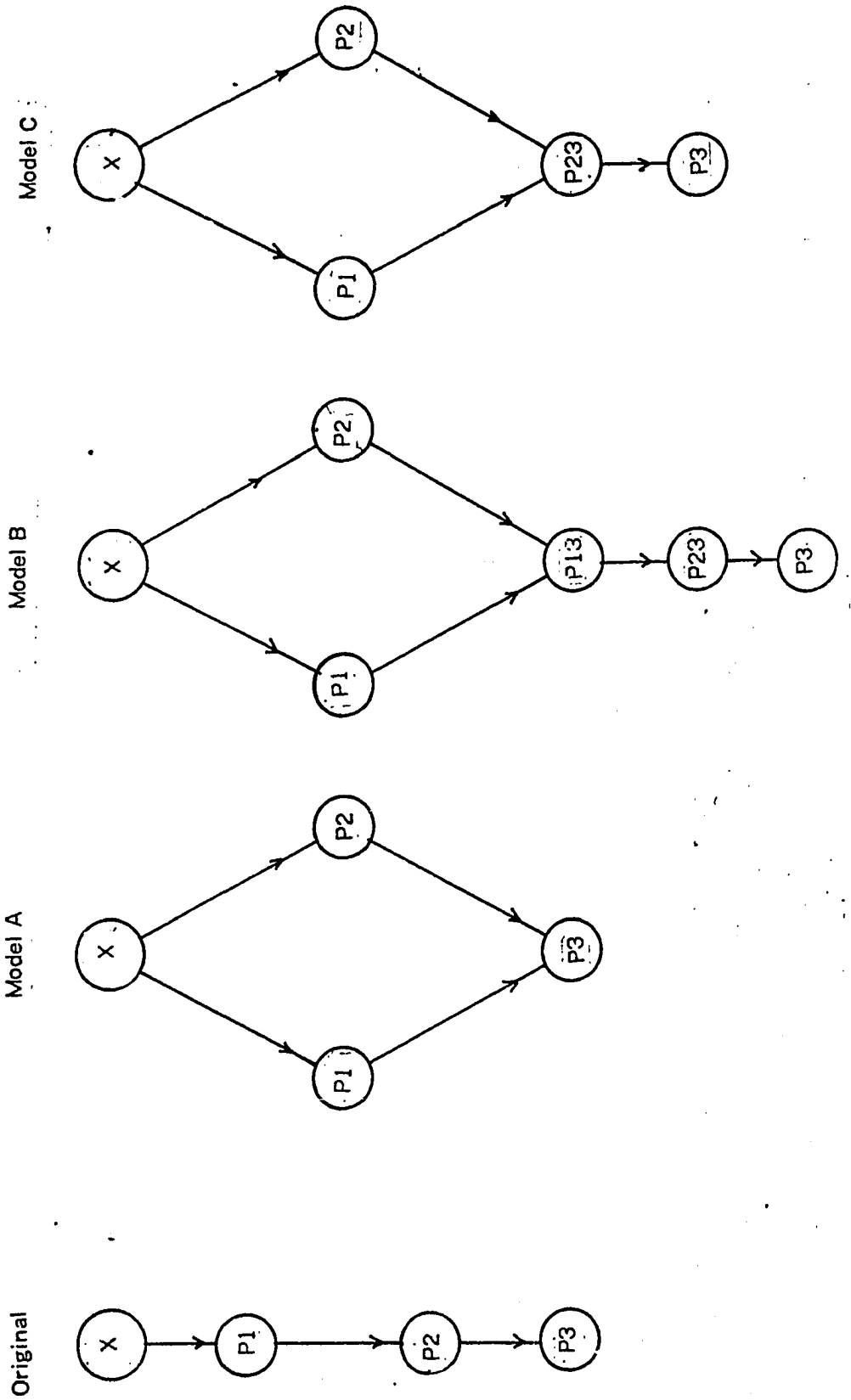


Figure 10. Flow of Computations -- Models A, B, and C

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

not. This is reflected in the underlying assumption that a slave memory does remember what was computed by its associated processor, namely, through the mechanism that decides whether something should be fetched from Mp or the slave memories. This "see through" mechanism essentially implies a flag for each variable in the destination set of the processor. This may prove too expensive if we are processing large sets of variables, for instance, arrays (a single flag for an entire array may produce undesirable side effects when the array is transferred back from the slave memory to Mp). In the absence of secondary memories with the "see through" property i.e. when a processor is forced to use its buffer as the only source of information, then an auxiliary computation (similar to the computations required to store the contents of the slave memories in the main memory, P13 and P23, in Fig. 10) is required to prestore the initial contents of the buffer memories. Models B and C become B1 and C1, and the new sequence of computations is shown in Fig. 11. Px1 and Px2 are the "computations" required to preload the slave memories.

The new models have one more condition to satisfy than their counterparts. The condition is that since we do not have a way to decide whether a variable has been modified or not (short of keeping old values around for comparison with the results of the computations) and since in the presence of branches this set of modified variables changes at every instance of the computations, P1 and P2 must not compute common results. Thus condition III is required.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

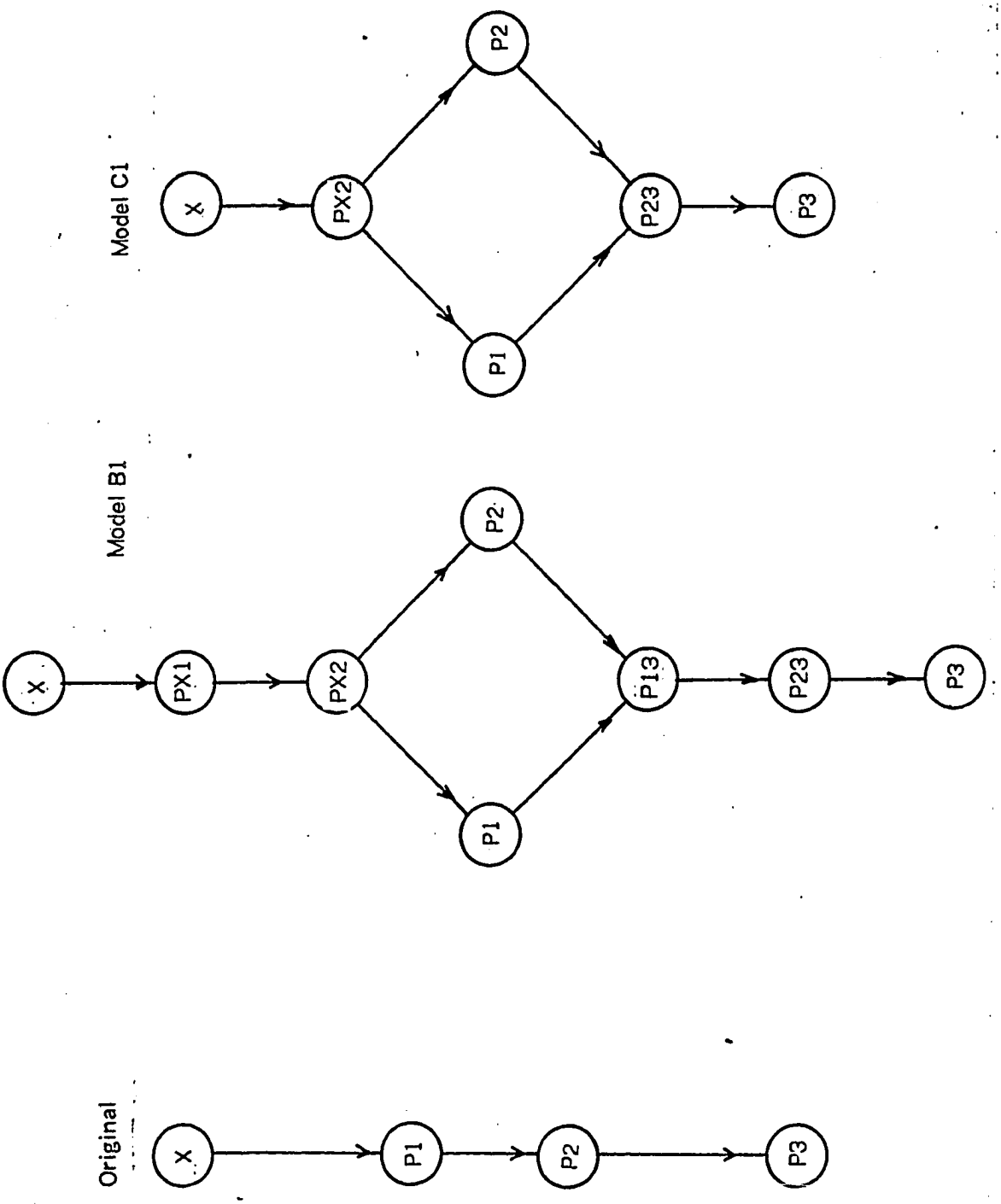


Figure 11. Flow of Computations – Models B1 and C1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Condition	Model A	Model B1	Model C1
Ia (R1∩D2=0)	required	not required	not required
Ib (S1∩¬R1∩D2=0)	required	not required	not required
IIa (D1∩R2=0)	required	required	required
IIb (D1∩S2∩¬R2=0)	required	not required	not required
III (D1∩D2∩R3=0)	required	required	required

In systems based upon models B1 and C1, where the buffer memories are preloaded, any variable that is *optionally* modified (or computed) must also be preloaded in the buffer*. Thus at the end of the computations, all variables that are in the destination set of the computation (whether they are modified or not by a particular instance of of the computation) can be transferred back to Mp without danger. As with their counterparts, Model C1 provides the same capabilities as Model B1 at a reduced cost.

The conditions for parallel computation can be determined for other schemas. They, together with the basic serial to parallel schema presented above, constitute the basis for the graph transformations that are discussed in chapter III.

The first two schemas, shown in Figs. 12 and 13, represent flowcharts in which two computations P1 and P2 are executed concurrently, followed by a third computation P3. The schemas and the conditions presented below analyze the possibility of initiating P3 before P1 (Fig.12) or P1 and P2 (Fig. 13) are completed. As in the

---

* That is, we keep the original value in case the variable is not recomputed in a given instance of the computation. These variables are those in the destination set of a subgraph that are not computed in the articulation blocks of the subgraph.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

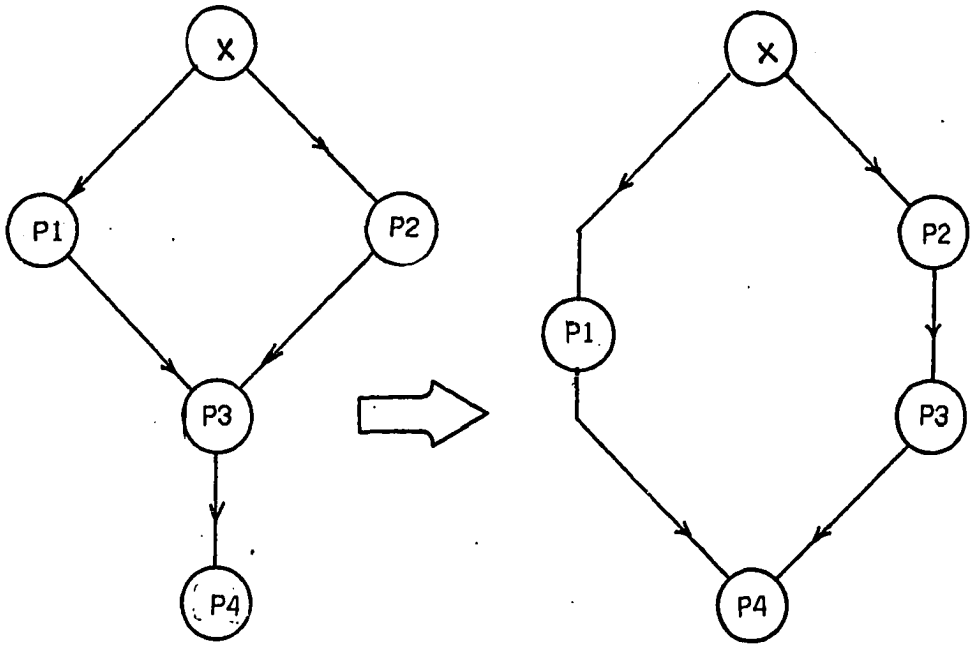


Figure 12. Concurrent Computation – Scheme 2

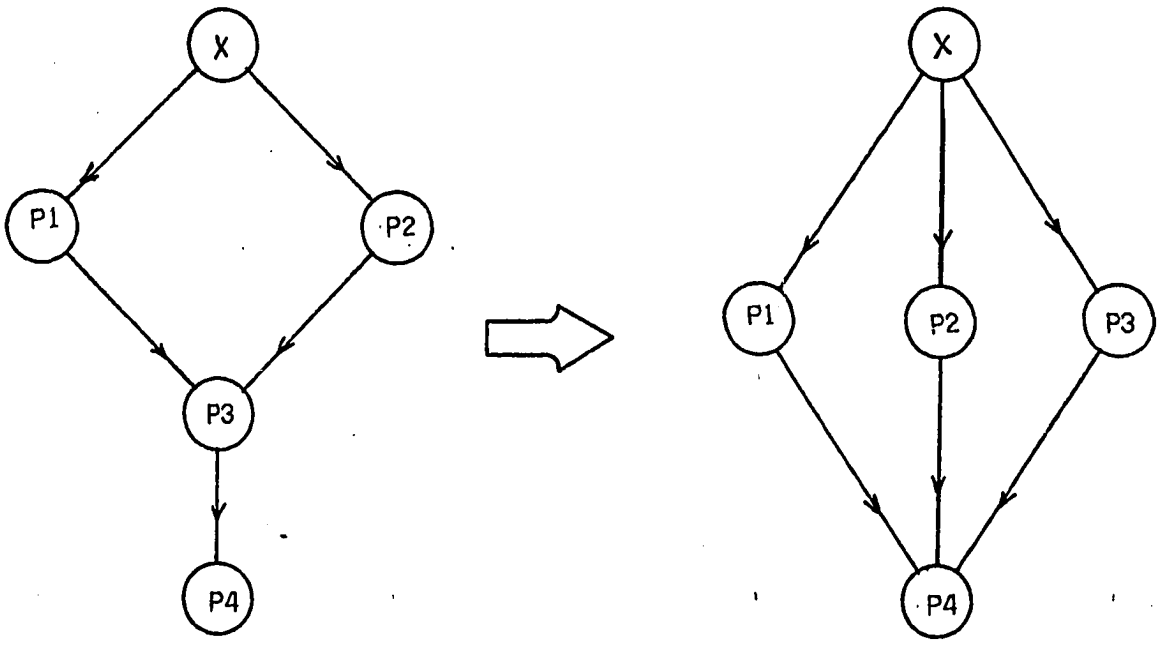


Figure 13. Concurrent Computation – Scheme 3

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

previous case we represent the "rest" of the program following P3 as a single computation, in this case, P4.

The conditions for the introduction of the extra concurrency in Fig. 12 can be stated as:

a) $R1 \cap D3 = 0$
b) $S1 \cap \neg R1 \cap D3 = 0$
c) $R3 \cap D1 = 0$
d) $S3 \cap \neg R3 \cap D1 = 0$
e) $D1 \cap D3 \cap R4 = 0$
f) $D2 \cap R3 \neq 0$

The first 5 conditions (a–e) are as we may expect, the conditions for concurrent execution of tasks P1 and P3. The sixth condition (f) expresses the dependence of P3 on results computed by P2.

The conditions for the extra concurrency shown in Fig. 13 can be stated as:

a) $R1 \cap D3 = 0$
b) $S1 \cap \neg R1 \cap D3 = 0$
c) $R3 \cap D1 = 0$
d) $S3 \cap \neg R3 \cap D1 = 0$
e) $D1 \cap D3 \cap R4 = 0$
f) $R2 \cap D3 = 0$
g) $S2 \cap \neg R2 \cap D3 = 0$
h) $R3 \cap D2 = 0$
i) $S3 \cap \neg R3 \cap D2 = 0$
j) $D2 \cap D3 \cap R4 = 0$

The conditions are those of simultaneous concurrency between (P1,P3), and (P2,P3).

It is obvious that the schema presented in Fig. 12 can be applied also to the case where P2 and P3 are independent. This possibility can be used to, in a sense,

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

degrade the extra concurrency that could be achieved from the schema of Fig. 13. This fact is used in chapter III by taking into account the cost/speed trade-offs that would result from applying the transformations derived from these schemas.

Another possible schema is shown in Fig. 14. Here one computation, P1, is

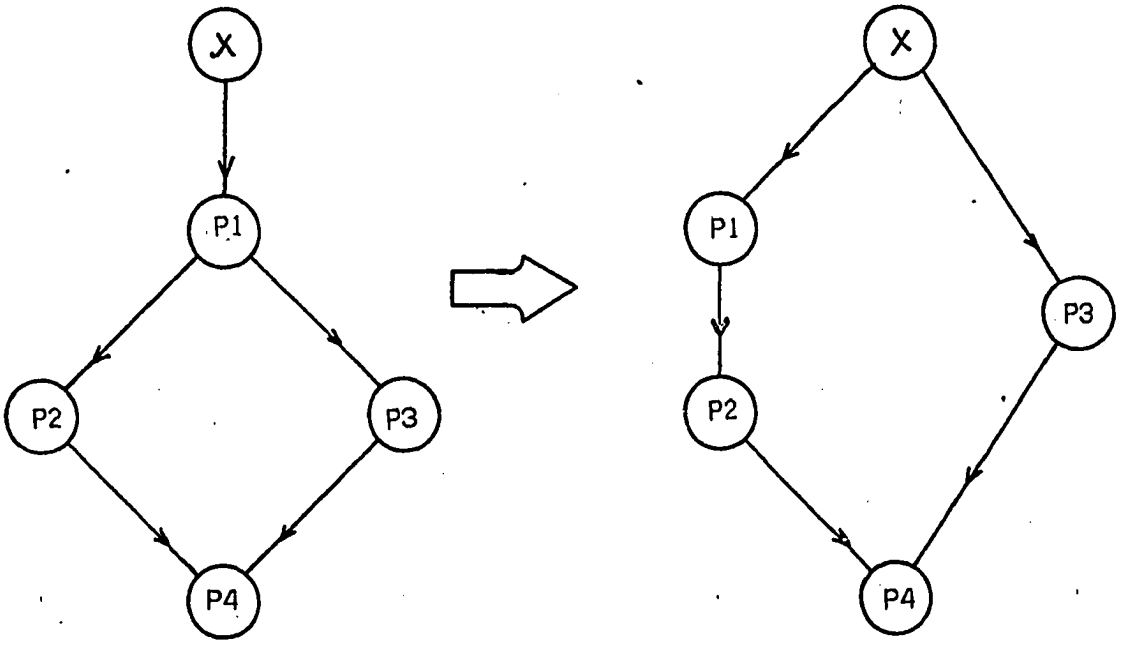


Figure 14. Concurrent Computation - Scheme 4.

followed by the concurrent execution of two other computations, P2 and P3. The schema and its conditions investigate the situation under which P3 could be initiated before P1 is completed. Although the level of concurrency remains the same, it is possible to obtain a gain in speed by computing P3 concurrently with the sequence P1,P2. The conditions are:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

a) $R1 \cap D3 = 0$
b) $S1 \cap \neg R1 \cap D3 = 0$
c) $R3 \cap D1 = 0$
d) $S3 \cap \neg R3 \cap D1 = 0$
e) $D1 \cap D3 \cap R4 = 0$

Given a pair of concurrent computations P1 and P2, Fig. 15, the conditions for

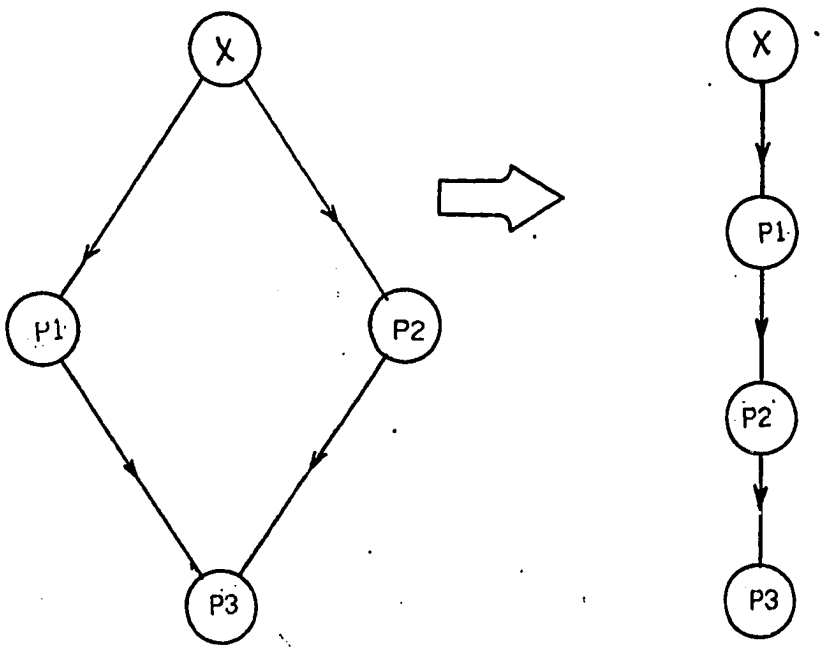


Figure 15. Parallel Computation

sequential processing can be stated as follows:

v) $D1 \cap R2 = 0$
vi) $D1 \cap D2 \cap R3 = 0$

The first condition is required so that the results of the process selected as the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

first in the sequence do not affect the second process (their parallel computation implied that neither required any results of the other). The second condition is required so that results of the first process that are modified by the second process are not needed by the rest of the program, P3.

Several facts about the schemas and their conditions can be observed. First of all, it is clear that the modifications in the order of execution of the computations preserve the behavior of the program. The conditions prevent the tasks from altering results computed by the other tasks, and since we are specifying an ordering in the transfer of results back to the main memory, the rest of the program is unaware that the execution sequence has been altered. A second fact is that the conditions for deciding the parallelism of two portions of a program are sufficient but not necessary. Bernstein shows that the parallelism of two program blocks (tasks) is undecidable i.e. there are no algorithms for deciding the parallelism or commutativity of arbitrary program blocks. Nevertheless, the set of sufficient conditions examined here, which represent the worst case analysis of the computations, can still be of value in practical applications. A third fact that must be pointed out is that a transformation schema that introduces extra concurrency in a program does not necessarily bring about a reduction in execution time. Similarly, a reduction in concurrency not always results in an increase in execution time. Chapter III will present the graph transformations derived from the schemas presented in this chapter and will also present the trade-offs under which a transformation achieves the desired goal (time or cost reduction).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

III. RTM DESIGN TRADE-OFFS

This chapter describes a set of transformations derived from the schemas for sequential/concurrent computations defined in the previous chapter. The first two sections, III.1 and III.2, present the characteristics of the RTM set and the design space parameters and their evaluation. Using a particular set of modules allows us to have a concrete basis for the study of the transformations and their effect on the cost/time requirements of a program in section III.3. The transformations are described not only in terms of the conditions that must be met for their application but also in terms of the net gain in cost or time they produce. It will be shown that the application of a transformation that adds concurrency to a set of computations not always results in a gain in speed (in fact, under certain conditions it may result in a loss). By the same token, a transformation that reduces the level of concurrency not always result in a loss of speed.

The last section describes the steps required to map the flowchart into an RTM system and how that system is evaluated in terms of cost and time.

III.1 – REGISTER TRANSFER MODULES

This section presents briefly the set of Register Transfer Modules (RTM). They will be used as the basic building blocks out of which digital systems can be designed and built. This section describes those RTM components and their interconnection rules that are of interest for our purposes. It is not intended to be a tutorial on RTM systems and more specific information can be found elsewhere[DEC71,Bell72b].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Control\K modules.- K modules are used to carry out the steps of the computation by activating the appropriate components in the system. Control modules can be classified in two groups, those that interact with the data structure of the RTM system and those that perform a control function proper. In the first group we have the Kevoke and Kbranch modules. In the second group we have, among others, the Kdiverge, the Kserial.merge, and the Kparallel.merge. The relation between the K modules and the control primitives used in the graph is obvious. Control flow enters a K module (of any type) via *activate* signals that enable the module. The Kevoke module generates an *evoke* signal requesting a specific action to be performed in the data part. When the operation is completed the Kevoke module is informed of this via a *done* signal, after which an *activate next* signal is generated so that the next step in the computation can be initiated. Kbranch, Kserial.merge, Kparallel.merge, and Kdiverge on the other hand, do not request actions from the data part and they simply produce the appropriate *activate next* signal(s).

Memory\M modules.- M modules provide a storing capability to the system. Memory modules perform data fetch and store operation under request of a Kevoke module.

Data operation\D modules.- D modules execute data operations and transformations under request of a Kevoke module.

Transducer\T modules.- T modules provide ways of communicating with the external world by transferring data to and from RTM systems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Bus.- The RTM bus is not a module in a normal sense, but a set of wires to which the RTM modules are connected. The bus provides a communication link between RTM modules. A special control module (Bus sense) is associated with a bus to provide several functions, among them the generation of timing signals (*done*, *data ready*, and *data accepted*).

The above is a general classification of the RTM modules. Each class provides different types of modules with different capabilities from which the user chooses the one best suited for his needs. Some modules are built combining functions in more than one class, for instance, there are modules capables of both data operation and storage functions; these are called the Data-Memory\DM modules. Several modules are described in greater detail in Appendix 1.

Several characteristics of RTM components can be observed:

1) The modules are specialized to carry out a selected and small set of functions. Thus it is not a difficult task to select the specific components to implement a system, albeit perhaps not the best possible.

2) The control and data parts are clearly separated, the latter being centered around the bus. Almost all data operations and transfers are implemented with the explicit use of the bus. Moreover, D, M, and DM modules are attached to a single bus, precluding the use of common storage between concurrent computations. RTM systems belong, therefore, to PMS configurations of type B1 or C1 (Chapter II).

3) RTM systems are capable of performing concurrent activities. Each bus in

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the system operates independently and asynchronously from the others. Since there are neither structural nor functional restrictions in the amount of parallelism it is possible to "improve" RTM systems by reducing the time required for a computation (increasing the concurrency) or by reducing the cost (reducing the overhead not only of the buses themselves, but all the non shareable data modules attached to the buses).

III.2 THE DESIGN SPACE

After the brief introduction to RTM components in the previous sections we can now introduce the design space dimensions, the transformation rules implemented in EXPL, and the trade-offs implied by the transformations. The delay in the presentation of these topics until this point is not because of any dependency on characteristics of the RTM set, but because having a concrete set of building blocks will facilitate the explanation.

Two parameters will be used to describe the design space: The cost of the hardware involved and the execution time of the program. The former is obtained by adding the costs of the components used in both the data and control structures. The latter is obtained from the average speed of the operations involved.

For a straight sequence of operations the time required is the sum of the individual times, Fig. 16.a. In the presence of concurrent activities, the operation time is that of the longest (timewise) sequence, Fig. 16.b. When alternative sequences are

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

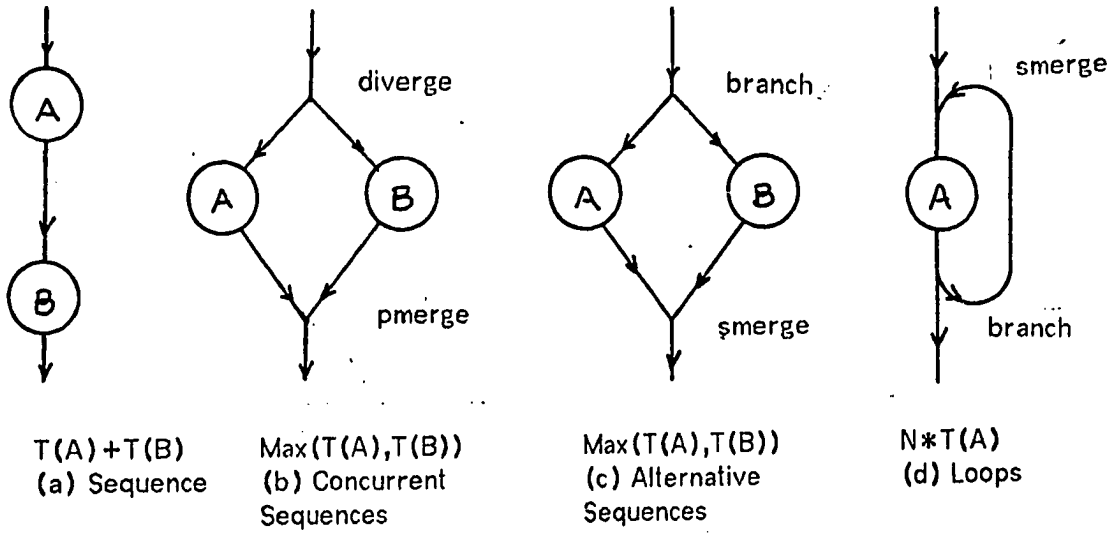


Figure 16. Time Estimation

initiated as a result of a data dependent decision, the time required for the execution is not known a priori. In this instance a worst case situation will be assumed, namely, that the longest path is the one selected, Fig. 16.c. - A more refined estimate would make use of branching probabilities assigned to each of the possible paths.

The presence of loops adds some complexity to the estimation of the operation time. In this case the level of nesting is assumed to be proportional to the frequency of execution of the operations, as shown in Fig. 16.d. Conceptually this is equivalent to replacing the cycle by a sequence of multiple copies of the individual operations. Since the number of times a loop is executed (i.e. the number of copies) is usually unknown, a default (2) is assumed. This default may be overruled by the designer by

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

specifying an estimated loop count.

Having defined the parameters of the design space we can now describe the transformations and their trade-offs.

III.3 THE TRANSFORMATIONS

Seven graph tranformation rules are implemented in EXPL. The rules themselves depend only on the graph model used and not on any characteristic of RTM's. The effect of the rules, however, as measured in terms of cost and time is obviously technology dependent and this should be kept in mind for the following discussion.

The abstract form of some of the rules has already been presented together with the conditions for serial and parallel computation. The concrete form of the rules is now presented, with their cost/time trade-offs in terms of RTMs. The data operations are left unspecified and the control operations, when relevant, are shown outside their blocks.

The transformations are of a general nature. They apply not only to individual blocks but to subgraphs of arbitrary complexity (provided that there is a unique entry block). Each subgraph (or block) is characterized by the variables used and/or modified by its computations. For clarity, the set of conditions required by systems type C1 are repeated with the transformations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

III.3.1 Rule SP (Serial to Parallel).- This rule, Fig. 17, implements the simplest case of adding concurrency to a computation, hopefully reducing the time requirements.

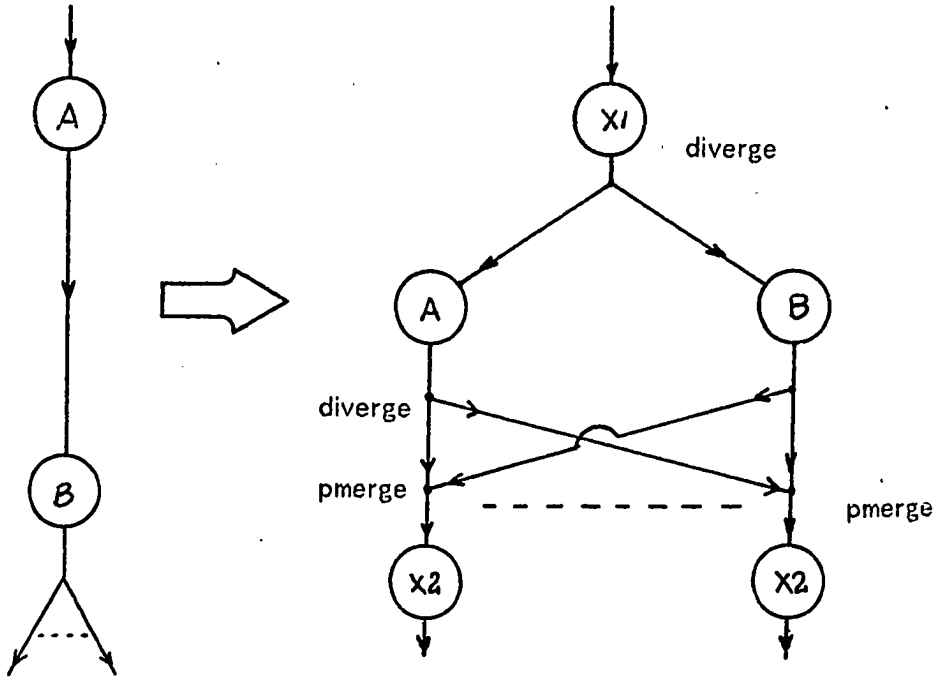


Figure 17. Rule SP - 1

The conditions for applicability of the transformation, in terms of the D, S, and R sets defined in chapter II, are the following:

$D(A) \cap R(B) = 0$;

$D(A) \cap D(B) \cap R(rest) = 0$

(R(rest) stands for the required set of the rest of the program, after B)

When these conditions are satisfied, the computations could be performed concurrently, as indicated in the transformed graph. Several data and control

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

operations appear in the resulting graph that were not present in the original graph. Block X1 is required to copy into local memory those variables used by computation B. Likewise, blocks X2 are required to transfer to the memory on the main bus those variables computed by B. Both, the X1 and X2 blocks are simple sequences of interbus data transfer operations (of whatever type is used in a particular module set).

A diverge operator is used as the tail of block X1 to initiate the concurrent computations of A and B. Another diverge operator has been added to A, to "broadcast" the termination of computation A. This is a general form of the rule and takes care of the case when more than one control path emanates from B (n>1). The parallel merges in nodes X2 are required to prevent the initiation of any of the computations following B (in the original graph) before both A and B are completed.

Several other control operators may be implied, depending on specific characteristics of the computations A and B. For instance, block A may have more than one incoming edge (i.e. the head operator of block A is a merge operator). In this case, the head of block X1 must have a control operator of the same type to preserve the behavior of the system, as shown in Fig. 18.

When the transformation is applied to more complicated computations, i.e. when A or B, or both are subgraphs of arbitrary complexity, the same general form of the rule applies. The only variant is the case when A is linked to B by more than one edge (the head operation of the entry block of subgraph B is a merge operation). In this case the rule takes the form shown in Fig. 19., where the difference is the use of an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

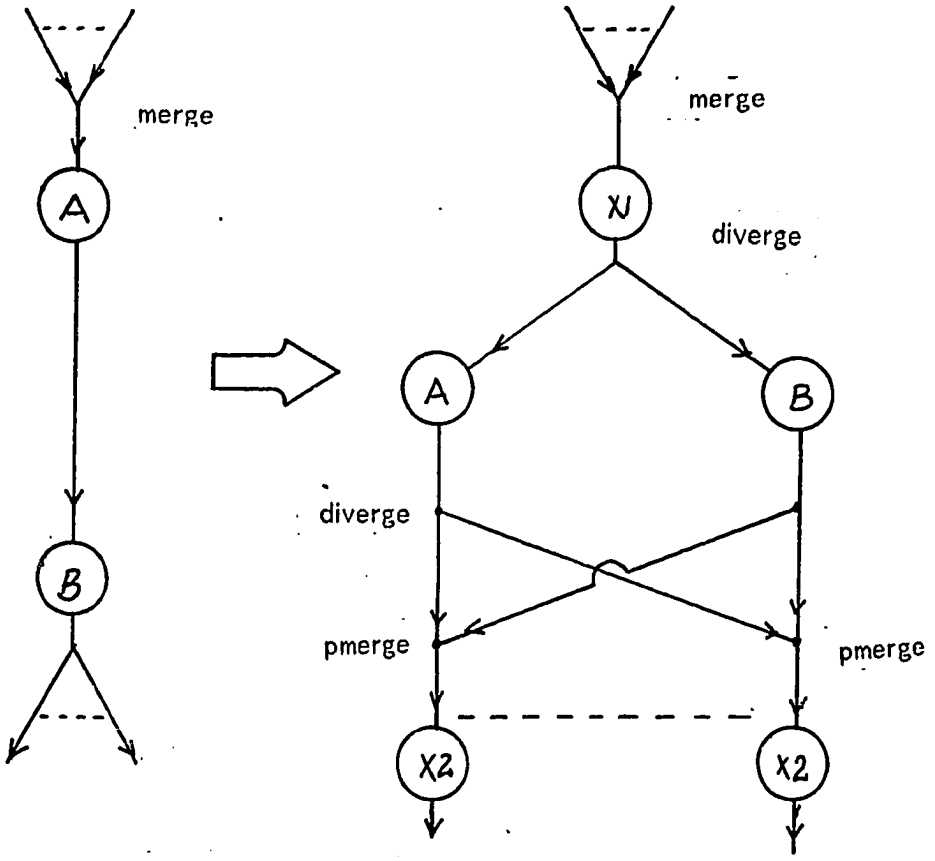


Figure 18. Rule SP – 2

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

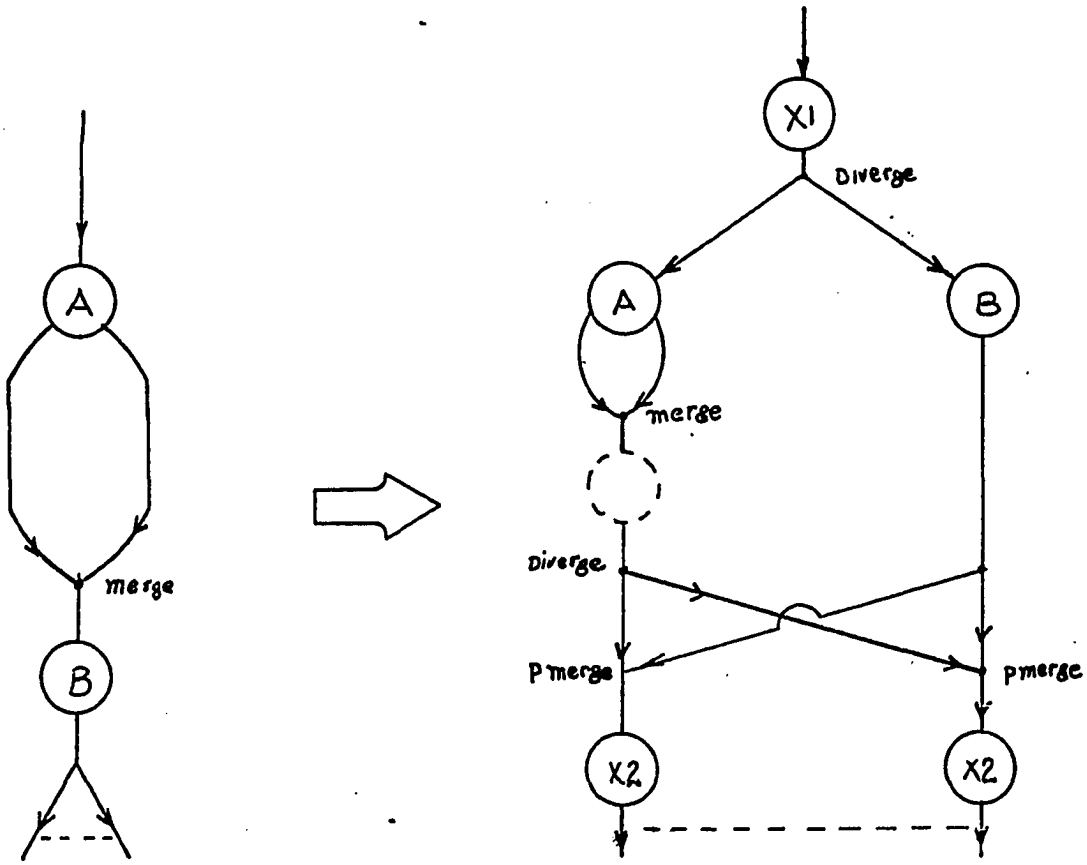


Figure 19. Rule SP – 3

extra control operation, a merge operator of the same type as the head of B. This operator together with the diverge forms a block without data operations, shown as a dotted circle.

The original sequence of computations can be characterized by the following cost and time values:

time = t(A) + t(B);

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$$cost = c(A) + c(B)$$

When the transformation is applied, the resulting graph can be characterized by:

$$time = t(X1) + \max(t(A), t(B)) + t(X2);$$

$$cost = c(X1) + c(A) + c(B) + n.c(X2) + \alpha.c(Bus)$$

where $\alpha = 0,1$ depending on whether the bus required by B is present and available (idle) in the system or not. In other words, if the extra bus can be shared then no extra cost is required on this account. The transfer blocks, however, as in the case of the time values may represent a non-trivial element in the expression.

The time savings are:

$$dt = t(A) + t(B) - t(X1) - t(X2) - \max(t(A), t(B))$$

It is possible then, according to this expression, that although the computations A and B are performed in parallel, no net gain in speed is achieved. And in some cases a loss in speed may occur, depending on the time required by the transfer operations.

The extra cost can be expressed as:

$$dc = c(X1) + n.c(X2) + \alpha.c(Bus)$$

These trade-offs point out the need for global analysis of the effect of a transformation. Thus, we can talk about *feasibility* conditions for a transformation, based on the graph connectivity and the data dependencies; and *desirability* conditions, based upon the effect of the transformation on the cost and time requirements. The former are based on local properties - the subgraphs representing the computations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

involved. The latter is based on global properties - the cost and time of the hardware required to implement the algorithm.

The other transformation rules are now described. These descriptions will be more compact than the previous one since most of the trade-offs considerations have been presented in rule SP.

III.3.2 Rule PS. (Parallel to Serial).- This rule transforms two concurrent computations into their sequential equivalent, Fig. 20.

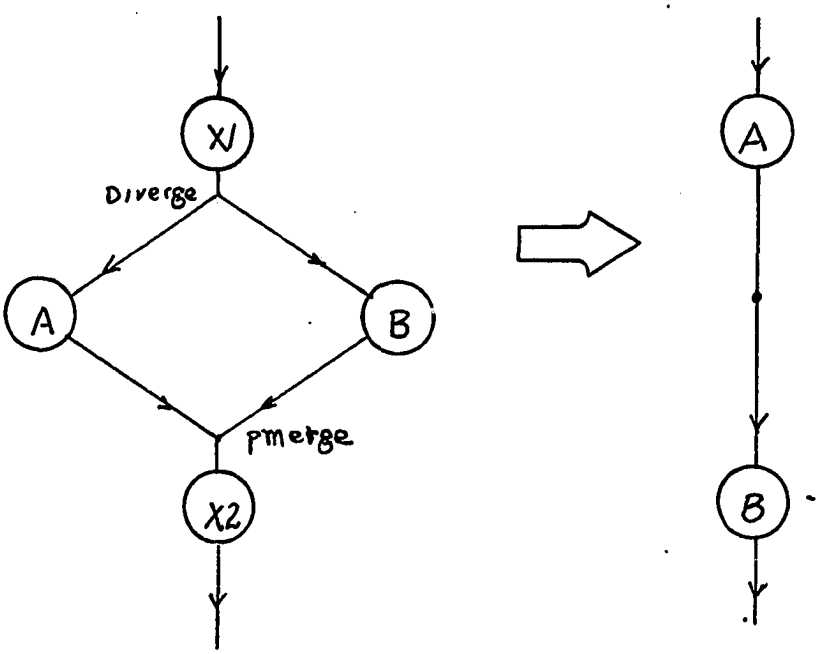


Figure 20. Rule PS

The conditions can be expressed as:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

$D(A) \cap R(B) = 0;$

$D(A) \cap D(B) \cap R(rest) = 0$

The parameters for the original graph are the following:

$time = T(X1) + t(X2) + \max(t(A), t(B));$

$cost = c(A) + c(B) + c(X1) + c(X2)$

The resulting sequential equivalent is characterized by:

$time = t(A) + t(B);$

$cost = c(A) + c(B) - \alpha.c(Bus)$

The extra time required by the computations is:

$dt = t(A) + t(B) - t(X1) - t(X2) - \max(t(A), t(B))$

From the expression, it is clear that the application of the rule does not necessarily produce an increase in time: nodes X1 and X2 used to load the local memory of B may take so much time, that their elimination may compensate for the extra time required by the serial computation of A and B.

The savings in cost can be expressed as:

$dc = c(X1) + c(X2) + \alpha.c(Bus)$

If the bus used by B is shared by some other computation ($\alpha = 0$) then it can not be eliminated from the system and no savings on this account are achieved.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

III.3.3 Rule SPSO (Serial Parallel Serial Case 0).- This transformation tries to achieve a time saving by initiating a computation at an earlier time, Fig. 21.

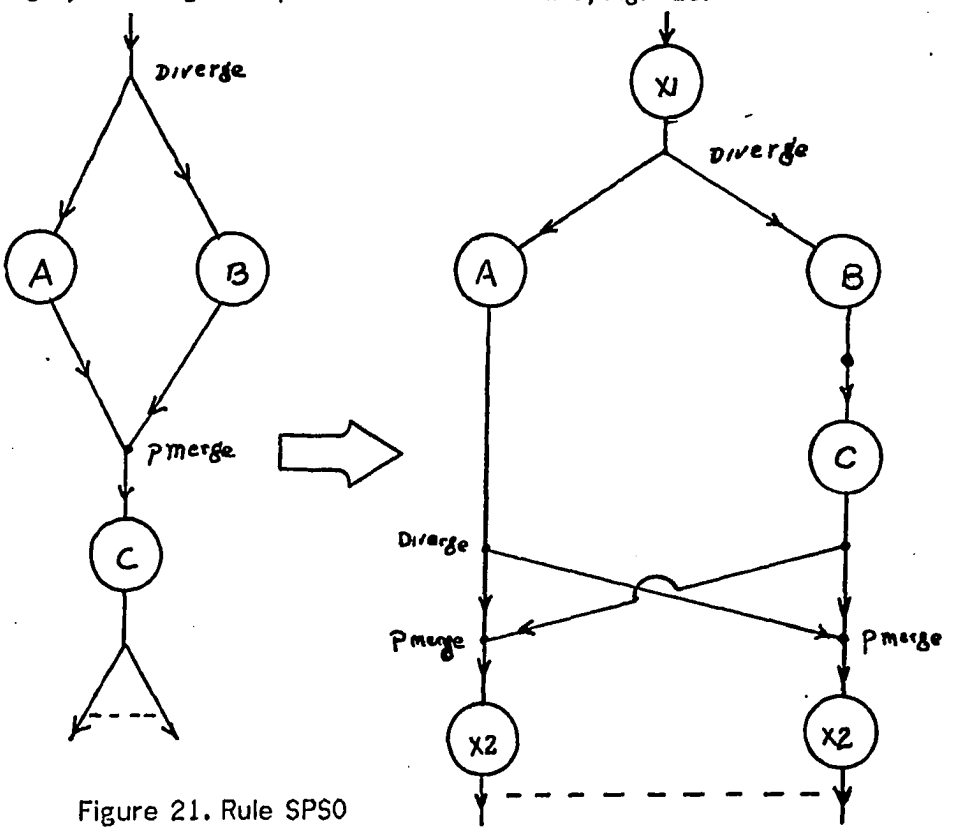


Figure 21. Rule SPSO

The conditions are:

$D(A) \cap R(C) = 0;$

$D(B) \cap R(C) \neq 0;$

$D(A) \cap D(C) \cap R(rest) = 0$

The original time and cost values are:

$time = \max(t(A), t(B)) + t(C);$

$cost = c(A) + c(B) + c(C)$

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The new time and cost-values are:

time = t(X1) + max(t(A), t(B) + t(C)) + t(X2);

cost = c(X1) + c(A) + c(B) + c(C) + n.c(X2)

and finally, the gain in time and the extra cost are:

dt = max(t(A), t(B)) + t(C) - t(X1) - max(t(A), t(B) + t(C)) - t(X2);

dc = c(X1) + n.c(X2)

III.3.4 RULE SPS1 (Serial Parallel Serial Case 1).- This is a combination of rules SP and SPS0, Fig. 22.

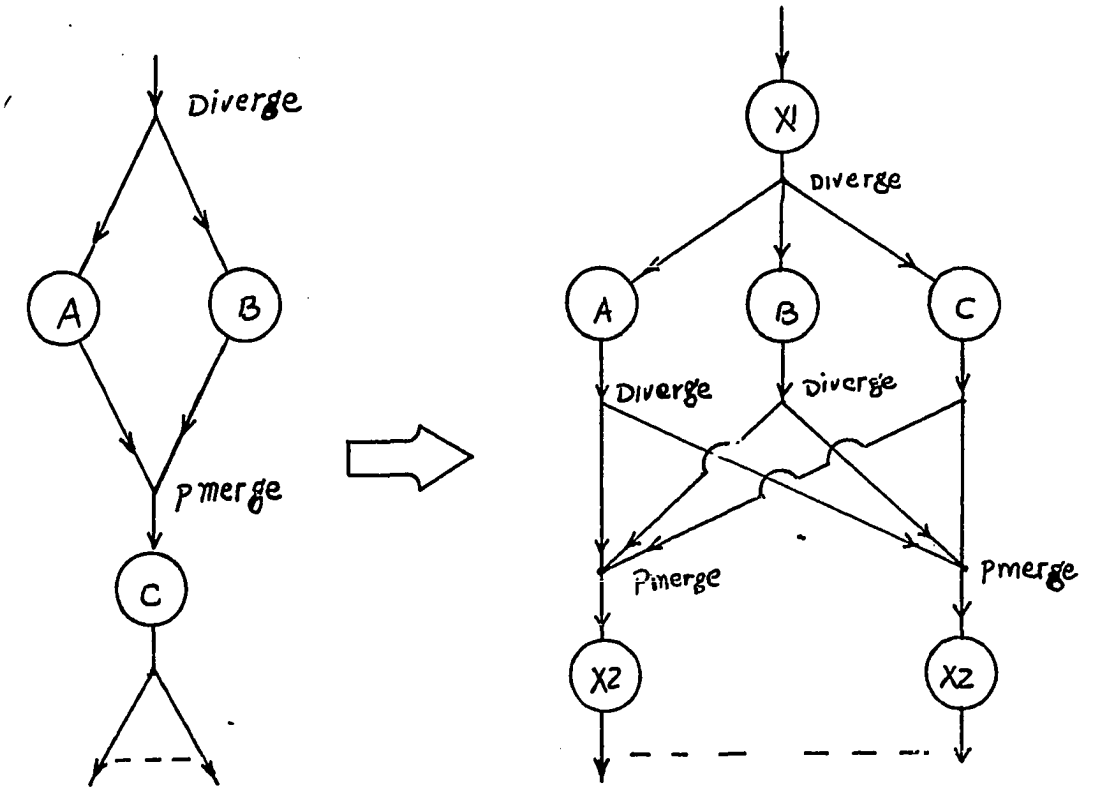


Figure 22. Rule SPS1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The conditions are:

$D(A) \cap R(C) = 0$;

$D(B) \cap R(C) = 0$;

$D(A) \cap D(C) \cap R(rest) = 0$;

$D(B) \cap D(C) \cap R(rest) = 0$

The original time and cost values are:

$time = max(t(A), t(B)) + t(C)$;

$cost = c(A) + c(B) + c(C)$

The new time and cost values are:

$time = max(t(A), t(B), t(C)) + t(X1) + t(X2)$;

$cost = c(X1) + n.c(X2) + c(A) + c(B) + c(C) + \alpha.c(Bus)$

The savings in time and the extra cost are:

$dt = max(t(A), t(B)) + t(C) - max(t(A), t(B), t(C)) - t(X1) - t(X2)$;

$dc = c(X1) + n.c(X2) + \alpha.c(Bus)$

where $\alpha = 0,1$ depending on the availability of the bus required by C. Notice that this rule can always be replaced by an application of rule SPS0. In particular, if $t(A) \geq t(B) + t(C)$ then SPS0 will produce a new configuration with the same time savings and without the cost of the extra bus.

III.3.5 Rule SPS2 (Serial Parallel Serial Case 2).- This is a mirror image (by inverting

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the direction of the control flow and performing the appropriate substitution of control operators) of rule SPS0, Fig. 23.

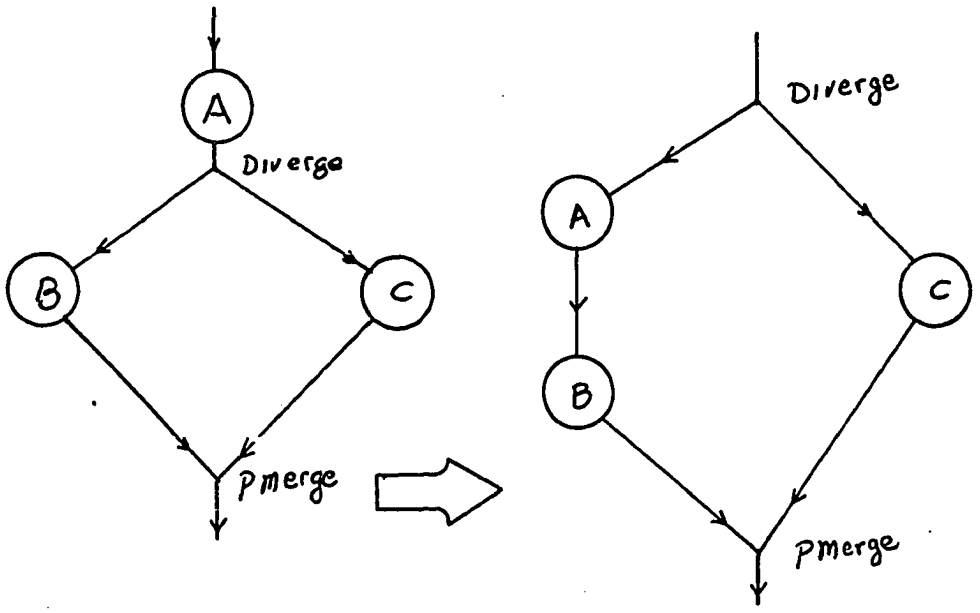


Figure 23. Rule SPS2

The conditions are:

$D(A) \cap R(C) = 0;$

$D(A) \cap D(C) \cap R(rest) = 0$

The original time and cost are:

$time = t(A) + \max(t(B), t(C));$

$cost = c(A) + c(B) + c(C)$

The new time and cost values are:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

time = max(t(A) + t(B), t(C));

cost = c(A) + c(B) + c(C)

The time gains can be expressed as:

dt = t(A) + max(t(B), t(C)) - max(t(A), t(B), t(C))

The following two rules are of a different nature. They are oriented towards a simplification of the control flow structure and they do not deal with any of the variable sets. These two rules eliminate data independent control operations (serial merge, parallel merge, and diverge).

III.3.6 Rule CK0 (Compact Control Case 0).- This rule eliminates control operators when they are not needed i.e. single entry merge (parallel and serial) operators and single exit diverges. These three cases of the rule are shown in Fig. 24. This rule always produces an improvement, both in cost and time.

III.3.7 Rule CK1 (Compact Control Case 1).- This rule collapses sequences of control operators of the same type, Fig. 25. This rule may or may not represent a net gain. We are only replacing sets of, say, parallel merges by a single parallel merge with more inputs. Depending on the particular RT set used, there may be limitations on the number of such inputs. In that case a tree of operators would have to be used. Optimal trees could be predefined and their use in a physical implementation would be

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

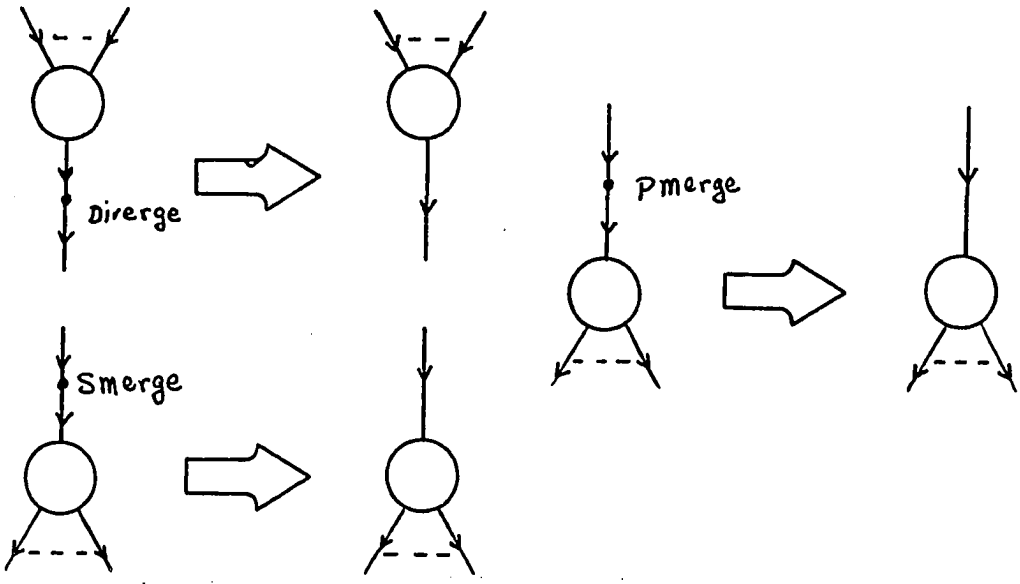


Figure 24. Rule CK0

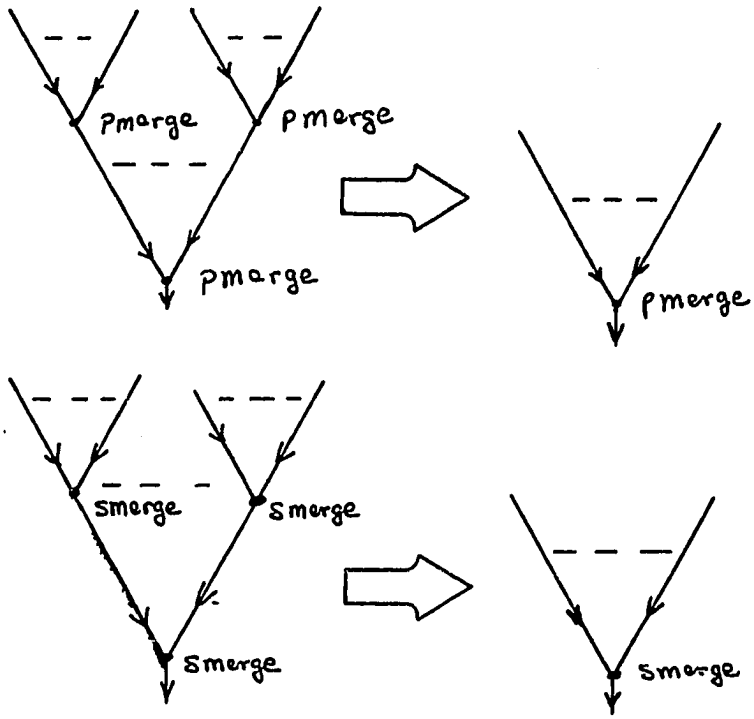


Figure 25. Rule CK1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

helped by the application of this rule.

III.4 COMPLETENESS OF THE TRANSFORMATIONS

We have presented a series of graph transformations aimed towards an increase in speed or a reduction in cost by restructuring the control flow graph. It is clear from the nature of the transformations that there is a limit in the variability that can be introduced in the flowchart. All the alternative designs that can be obtained by applications of the seven transformations must perform the computation in a similar way.

The following list, obtained from [Bell72b], p. 155, shows some of the techiques that can be used in digital systems design to effect changes in some specified dimension. The list is only partially reproduced, the book includes other dimensions, like reliability, generality, modifiability etc, that are outside the scope of this thesis.

1. To increase speed
    - 1.1 Compute in parallel
        - 1.1.1 – Array parallelism
        - 1.1.2 – Pipeline parallelism
        - 1.1.3 – Functionally independent parts (concurrency)
    - 1.2 Use table look-up
    - 1.3 Unwind control loops
    - 1.4 Use multiple representations
2. To decrease cost
    - 2.1 Share facilities
        - 2.1.1 – Subroutines
        - 2.1.2 – Extract memory and share data operations
        - 2.1.3 – Merge control paths
    - 2.2 Use software control

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

It is clear that the transformations belong to the set of techniques that increase speed by computing in parallel (1.1.3) and reduce cost by sharing facilities (2.1.3). Special cases of 1.1.1, array parallelism, could be handled if the algorithm is described as a sequence of (identical) operations performing on different elements of the array. In any event, the large number of techniques that can be applied to improve a design covers a wider design space than could be obtained from the control flow transformations. Moreover, even among pure control flow graph transformations there are several possibilities that were not included in this work. They correspond to what is known as code motion optimization [Rust72]. The main characteristic of these optimizations is the movement of computations from frequently executed paths to less executed paths. The conditions for this transformations can also be expressed in terms of the variable sets defined in chapter II, but instead of serial/parallel transformations we are interested in the conditions for commutativity of two computations:

$D(A) \cap R(B) = 0;$

$D(B) \cap R(A) = 0;$

$D(A) \cap D(B) \cap R(rest) = 0$

(R(rest) stands for the required set of the rest of the program)

Although from the previous discussion it can be observed that the transformations allow us to explore only a subset of the possible designs for a given algorithm, it is nevertheless feasible and desirable to study the transformations as a set and analyze some measure of "completeness" of the set.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Let us assume that there are three computations, A, B, and C to be executed. The completeness of the transformations will be shown by observing all possible ways that these three computations can be organized and how the transformations can be used to find them, starting from an arbitrary initial solution.

The first case analyzed is that in which all three computations are independent from each other. This case presents the maximun number of alternative solutions. The other cases studied are those in which some prespecified dependency in the order of execution is given; the possible solutions in these cases are proper subsets of the possible solutions in the fully independent case.

The following notation will be used as a shorthand to indicate the possible solutions (organization of the computations):

X->Y indicates that computation Y is performed immediately after computation X,

X//Y indicates the concurrent computation of X and Y.

Parenthesis are used to group parts of a shorthand expression, for instance, (X->Y)//Z indicates the concurrent execution of the sequence X->Y with Z.

There is a finite number of ways that three computations, A, B, and C can be organized in a flowchart. The maximun number of such solutions is found in the case of three fully independant computations. These solutions are shown in Fig. 26 and the corresponding flowcharts appear in Fig. 27.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Solution	Shorthand expression
1	A->B->C
2	A->C->B
3	B->A->C
4	B->C->A
5	C->A->B
6	C->B->A
7	(A//B)->C
8	(A//C)->B
9	(B//C)->A
10	C->(A//B)
11	B->(A//C)
12	A->(B//C)
13	(A->B)//C
14	(A->C)//B
15	(B->C)//A
16	(B->A)//C
17	(C->A)//B
18	(C->B)//A
19	A//B//C

Figure 26. 3-Block Computations - Shorthand

Given any initial solution, the rules can be used to find alternative solutions. The possible derivations are shown in Fig. 28. The figure organized in a different way appears as a transition matrix in Fig. 29.

It can be shown that the transitive closure of the matrix contains no empty positions, i.e. any solution can be derived from any other solution. The proof is very simple: Any solution 1-18 can be transformed, by one or two applications of rule SP, into solution 19, and solution 19 can be transformed into any other solution by one or

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

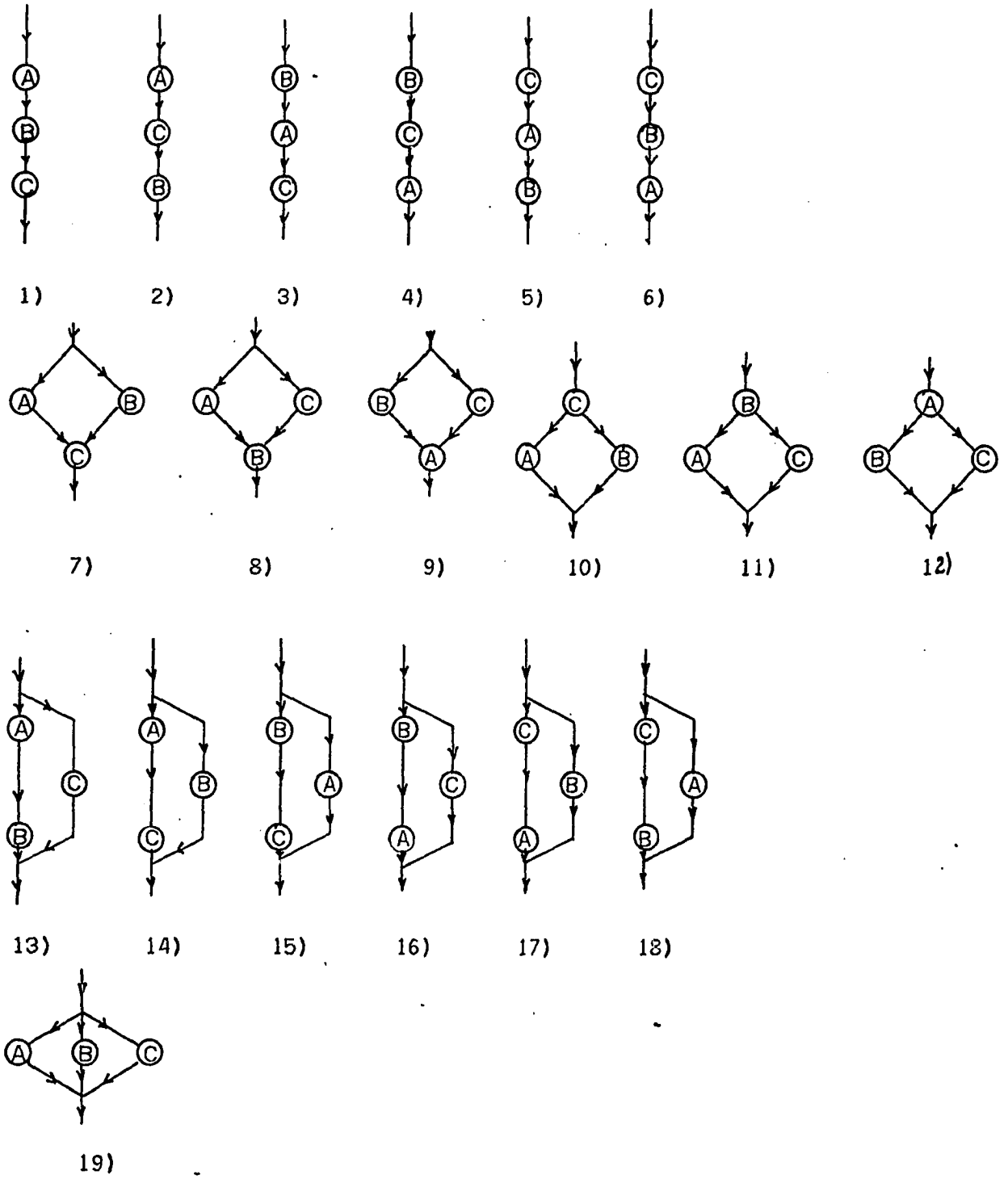


Figure 27.3 – Block Computations – Flowcharts

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Solution	Rule	Derived Solution(s)
1	SP	7,12,13
2	SP	8,12,14
3	SP	7,11,16
4	SP	9,11,15
5	SP	8,10,17
6	SP	9,10,18
7	PS	1,3
7	SP	19
7	SPS0	14,15
7	SPS1	19
8	PS	2,5
8	SP	19
8	SPS0	13,18
8	SPS1	19
9	PS	4,6
9	SP	19
9	SPS0	16,17
9	SPS1	19
10	PS	5,6
10	SP	19
10	SPS2	17,18
11	PS	3,4
11	SP	19
11	SPS2	15,16
12	PS	1,2
12	SP	19
12	SPS2	13,14
13	PS	1,5
13	SP	19
14	PS	2,3
14	SP	19
15	PS	1,4
15	SP	19
16	PS	3,6
16	SP	19
17	PS	4,5
17	SP	19
18	PS	2,6
18	SP	19
19	PS	7,8,9,10,11,12,13,14,15,16,17,18

Figure 28. 3—Block Computations – Derivations

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	/						SP					SP	SP						
2		/						SP				SP		SP					
3			/				SP				SP					SP			
4				/					SP		SP				SP				
5					/			SP		SP							SP		
6						/			SP	SP								SP	
7	PS		PS				/							SPS0	SPS0				SP,SPS1
8		PS			PS			/					SPS0					SPS0	SP,SPS1
9				PS		PS			/							SPS0	SPS0		SP,SPS1
10					PS	PS				/							SPS2	SPS2	SP
11			PS	PS							/				SPS2	SPS2			SP
12	PS	PS										/	SPS2	SPS2					SP
13	PS				PS								/						SP
14		PS	PS											/					SP
15	PS			PS											/				SP
16			PS			PS										/			SP
17				PS	PS												/		SP
18		PS				PS												/	SP
19							PS	PS	PS	PS	PS	PS	PS	PS	PS	PS	PS	PS	/

Figure 29. 3 – Block Computations – Transition Matrix

two applications of rule PS. The proof also shows that rules SP and PS constitute a necessary and sufficient set. The other rules simply reduce the number of steps required to derive the solutions.

Certain constraints, derived from data dependencies among the computations, could be specified, in which case the number of possible solutions decreases. These constraints will be specified as follows:

X=>Y indicates that X must be computed before Y (with some, possibly, intermediate computations),

X///Y indicates that X and Y must be performed along concurrent paths.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The non-trivial constraints (i.e. those that imply more that one possible solution) and the possible solutions are shown in Fig. 30. The figure indicates that the constraints can be classified in three groups: cases 1 through 6, cases 7,8, and 9,

Case	Conditions	Possible solutions
1	A=>B	1,2,5,8,12,13
2	A=>C	1,2,3,7,12,14
3	B=>A	3,4,6,9,11,16
4	B=>C	1,3,4,7,11,15
5	C=>A	4,5,6,9,10,17
6	C=>B	2,5,6,8,10,18
7	A=>B,A=>C	1,2,12
8	B=>A,B=>C	3,4,11
9	C=>A,C=>B	5,6,10
10	A///B	7,10,14,15,17,18,19
11	A///C	8,11,13,15,16,18,19
12	B///C	9,12,13,14,16,17,19

Figure 30. 3-Block Computations - Constraints

and cases 10, 11, and 12. Due to the regularity among the constraints in a group we will analyze only one case of each group since the results intuitively apply to the other cases in the group.

Let us consider the constraint A=>B (representative of the first group). The possible solutions and the solutions derived from them are shown in Fig. 31.a, the transition matrix is shown in Fig. 31.b and the strongly connected network shown in Fig. 31.c is just another representation of the derivation of the solutions. As in the

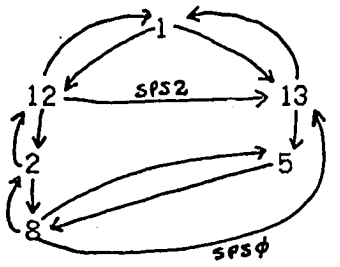
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Solution	Rule	Derived solution(s)
1	SP	12,13
2	SP	8,12
5	SP	8
8	PS	2,5
8	SPS0	13
12	PS	1,2
12	SPS2	13
13	PS	1,5

(a)

	1	2	5	8	12	13
1					SP	SP
2				SP	SP	
5				SP		
8		PS	PS			SPS0
12	PS	PS				SPS2
13	PS		PS			

(b)



(c)

Figure 31. Constraint A=>B

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

fully independent case, rules SP and PS constitute a necessary and sufficient set. It can be proved by deleting (in Fig. 31.c) the directed arcs from solution 12 to solution 13 (rule SPS2) and from solution 8 to solution 13 (rule SPS0); the resulting network is still strongly connected by applications of rules SP and PS.

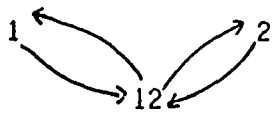
A similar analysis of constraint A=>C,A=>B (representative of the second group) and constraint A///B (representative of the third group) is shown in Figs. 32 and 33 respectively. Deleting the arcs corresponding to rules other than SP or PS

Solution	Rule	Derived solution(s)
1	SP	12
2	SP	12
12	PS	1,2

(a)

	1	2	12
1	/		SP
2		/	SP
12	PS	PS	/

(b)



(c)

Figure 32. Constraint A=>B, A=>C

(Figs. 32.c and 33.c) still leave strongly connected networks.

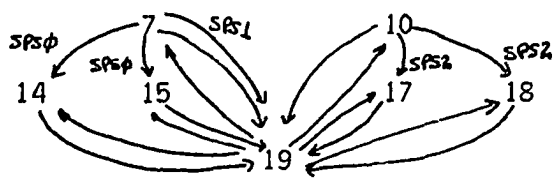
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Solution	Rule	Derived solution(s)
7	SP	19
7	SPS0	14,15
7	SPS1	19
10	SP	19
10	SPS2	17,18
14	SP	19
15	SP	19
17	SP	19
18	SP	19
19	PS	7,10,14,15,17,18

(a)

	7	10	14	15	17	18	19
7	/		SPS0	SPS0			SP,SPS1
10		/			SPS2	SPS2	SP
14			/				SP
15				/			SP
17					/		SP
18						/	SP
19	PS	PS	PS	PS	PS	PS	/

(b)



(c)

Figure 33. Constraint A///B

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The overall conclusions are: 1) the set of transformations is complete in the sense that they derive all possible solutions from an arbitrary initial solution, under different constraints, and 2) the set PS,SP is the only necessary and sufficient set of transformations; the other transformations simply reduce the number of steps required to obtain the alternative solutions.

III.5 - RTM DESIGN EVALUATION

Once a transformation has been performed, the cost and time requirements in the new configuration must be measured to ascertain whether an improvement has been achieved. This evaluation is clearly dependent on the module set used, and is performed by ad-hoc routines, independent of the graph transformations. This section describes the evaluation procedure for RTM systems.

The initial assumption is the existence of a behavioral description of the object being designed i.e. a graph representation. In this representation the elements (control and data operators) are not bound to specific physical components. The evaluation is performed by applying a series of binding algorithms that map this behavioral representation into a physical (RTM) representation, which is then measured*.

The order in which this binding process is performed is dictated by the interconnection rules of the RTM set. All data operators and carriers are connected to

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the bus, which not only takes part in the data operations but also has control functions, being responsible for the generation of some control signals. This feature of RTM requires that the binding of the buses precedes the binding of the data modules. Since in RTM concurrency of operations is equivalent to the number of available buses the selection and binding of the buses must follow the binding of the control structure.

The order of binding is the following:

1) Selection and identification of the control modules. Binding of control modules.

2) Selection and identification of the buses required to implement the control flow structure. Binding of buses.

3) Selection and identification of the data modules that are to be attached to each bus. Binding of data modules.

III.5.1 Control binding.- This first step consists of assigning physical components to the control operators used in the graph model. This is a straighforward procedure since there is a one-to-one correspondance between graph control operators and some

---

* It should be pointed out that the design space we are dealing with is really an evaluation space, not a structural space. In other words, for each point in the evaluation space there may be more than one structure but we only analyze one, the so to speak, "canonical" implementation of the flowchart.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

RTM control modules (Kbranch, Kdiverge, Ksmerge, and Kpmerge). However, control operators needed in RTM to direct the data operations must be added. These control operators (Kevokes) are not present in the graph model. The data operations require different numbers of Kevokes (steps), depending on the type of variables involved and the complexity of the data operation. This process is similar to a macro-expansion where to each data operation corresponds a template of Kevokes connected to data modules (still unbound). Since the costs and times are known for each RTM operation, the evaluation of the basic blocks is a straightforward process. This evaluation reflects the time required by the operation, but only the cost of the control part (since the data modules and buses are, in general, shared by the computations their evaluation is performed later, at a global level).

III.5.2 Bus binding.- The selection and identification of the number of buses required by a given control flow graph is performed by a "coloring" algorithm applied to the graph. The main idea is to mark blocks that are executed in the same bus, with the same color. A detailed example is worked out in Appendix 2.

The algorithm works in three steps:

1) An initial coloring is performed, by which subgraphs of the flowchart are assigned colors. The criterion used is to maximize the size of the subgraph, provided that no concurrent activities are performed inside. This initial number of colors provides the initial (upper bound) number of buses.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

2) Conflicting colors are identified and this is reflected in a conflict table. Two colors conflict whenever any two blocks, one of each color, are either immediate successors of a diverge control operation, or immediate predecessors of a parallel merge control operation. The two colors clearly must correspond to different buses.

3) A pair of non conflicting colors is chosen and the colors are "merged" (they become one and the same). The conflict table is updated to reflect this fact (any color that conflicts with either of them will conflict with both after this merging). The criterion used in the selection of the two colors to be merged is the minimization of the number of common variables (to reduce the interbus communication and replication of components). The pair of non conflicting colors with the highest level of commonality is selected as the candidate pair. This process is repeated until no more pairs of non conflicting colors are found. The number of colors left is the number of buses.

III.5.3 Data binding.- Once the buses have been bound, the process of binding the data components is simple. The blocks of the flowchart have been "colored" by the bus on which they are to operate. Thus, the units needed on each bus can be determined by examining the requirements. Although this process is simple due to the specialization of RTM components, some cost/time trade-offs, not implemented in EXPL, can be effected on the data part. This data optimization (along a given bus) is reflected in the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

selection of optimal numbers of functional units (DMgpa). The criterion used is the trade-off between the transfer of variables to and from the functional units and the cost of using additional units, taking advantage of their local memory capabilities. This problem is somewhat similar to the register allocation problem [John73].

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

IV. THE EXPL SYSTEM

This chapter presents the characteristics of the EXPL system. Section IV.1 describes the language, ISP, that is used to define the behavior of the initial solution. Section IV.2 describes the data structures used to represent the alternative solutions. Section IV.3 describes the organization of EXPL and the command facility available to the user. Finally, section IV.4 describes the characteristics of the design space search and the heuristics used.

IV.1 – THE ISP COMPILER

In order to provide a clean interface between the user and EXPL, a high level hardware description language, ISP [Bell71a], is used to describe the behavior of the object being designed. It allows the definition of the *data carriers* (memories) and the *data operations* performed on the data, as well as the order in which these operations are carried out.

The following paragraphs describe some of the features of the language. A more detailed description can be found in [Bell71a].

Data carriers are defined in ISP by a name and description of the structure, given by bracketed lists of names (if names are associated with the subcarriers) or constants, much like array declarations in Algol, e.g.:

Ac<0:11>

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Ac is the name of a carrier, a register 12 bits wide, named from 0 to 11 (from left to right). The ":" or range operator is used to denote an abbreviated list of elements.

For descriptive purposes there is an abbreviation or alias operator, "\", which is used as a delimiter for a list of names, all of which are thus made equivalent, e.g.:

Accumulator\Ac<S,1:11>

is a valid definition of the carrier, but now it can be referred to as either "Accumulator" or "Ac" indistinctly. Square brackets are used to specify those dimensions where the accessing is done through some "addressing" (switching) schema:

Memory\Mp[0:7777↓8]<0:11>

The memory consists of 4096 words, each of 12 bits. Constants are, by default, decimal numbers, unless otherwise specified by the "↓" (base) operator.

Elements are specified by "names" (numbers do not indicate relative positions), therefore, it is legal to describe a 7 bit register as:

R<A,15,B,13,11,9:10>

the only concession to the use of numbers as both names and position indicators is by using the ":" (range) operator, whereby, the abbreviated list consists of the bounds and all integers in between, with the implication that these consecutive integers also name consecutive (from left to right) elements.

Data operators work on *data types*. Associated with the data types we find carriers, the physical components used in storing and transmitting the data types. The data operators take their inputs (data type carriers), operate on the data, and present

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the result as output (the resulting data type carriers). Data operator complexity vary from a simple transfer path to combinational networks to more complex transformations involving sequences of simpler operations.

Data operators in ISP include:

- Concatenation (□)
- Boolean (¬,∨,∧,⊕,≡)
- Arithmetic (+,−,*,÷)
- Relational (=,≠,<,≤,>,≥)
- Transfer (←)

Since all operations result in modification of bits in memories (permanently or otherwise), each action takes the following form:

memory-expression ← *data expression*

the *data-expression*, patterned after standard mathematical notations, describes the transformations (if any) and the information pattern that is to be placed in the memory described by the *memory-expression*, e.g.:

Ac←Ac∧Mp[EA]

Expressions can be followed by a modifier, providing more information about the meaning and interpretation of the operands and operators. An *expression modifier* consists of a data-type or an operation mode enclosed in curly brackets "{" and "}", e.g.:

L□AC←L□AC*2 {SHIFT LEFT}

A←B+C {2's COMPLEMENT}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

In ISP concurrency of actions is the rule rather than the exception. This is reflected in the use of the ";" as a delimiter for lists of concurrent actions. Sequencing is expressed by using the term *next* as a delimiter for lists of sequential actions. Complex concurrent and sequential activities can be described in terms of simpler activities using "next", ";", "(", and ")", e.g.:

IR←Mp[Pc] single action

IR←Mp[Pc];Pc←Pc+1 concurrent actions

IR←M[Pc];Pc←Pc+1; next State←1 action sequence of two steps, the first one consists of 2 concurrent actions.

(IR←Mp[Pc];Pc←Pc+1; next State←1);(Ac←0;MQ←Ac) concurrent action sequences

Conditional action sequences can be defined in terms of conditional action sequences. Parenthesis are used to indicate the scope of the conditional activities:

(OP=2⇒Mp[EA]←Mp[EA]+1; next (Mp[EA]=0⇒Pc←Pc+1))

A more formal definition of a subset of ISP can be found in Appendix 3. This subset, implemented at CMU, allows the description of simple digital systems. The main restrictions are:

- Memories can have up to 2 dimensions (i.e. words and bits)
- No user defined operators are allowed
- Expression modifiers are ignored

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

- Memory-expressions accept a single element along the word dimension and one set of contiguous elements along the bit dimension (i.e. no scattered access is allowed).

The reason for these limitations is that the compiler is intended to be used as a teaching aid in a Register Transfer Level Design course, and the features implemented are those efficiently implementable in RTM's.

The intermediate code of the compiler is, however, independent of the RTM set and was defined such that there is a one-to-one mapping between the elements of the intermediate language and the elements of the graph model. This organization allows the use of the ISP compiler as a front end for the EXPL system. The user can, however, provide his own problem description in the intermediate language without using the compiler. This intermediate level program is written as a BLISS10 [Wulf71] program. Each of the data and control operators used in the graph model is represented by a routine call, with the appropriate parameters. When the program is executed the routines called build the internal tables (Symbols and Statements) that define the graph. This facility to by-pass the compiler allows us to experiment with designs that can not be described in ISP. This facility was also used during the development and testing of EXPL, before the compiler was implemented.

IV.2 - THE DATA STRUCTURES

This section describes the format and type of information stored in several

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

internal tables, used to represent the flowcharts and the applicable transformations. The examples shown in the figures correspond to the RTM multiplier used in chapter V.

- Symbol table.- Figure 34 shows a picture of a symbol table. It contains the characteristics of the variables used and it is produced by the ISP compiler. The MTYPE field indicates the module type assigned to the variable i.e. array, register, flag, or constant. The FLAGS are used for internal purposes only. The PNAME field contains the variable name as declared by the user or the value in the case of constants. The INDEX field is the internal name of the variable, this is the identification used by the other tables when referring to a variable (or a constant).

- Statement table.- Contains an entry for each of the data and control operations. The entries include pointers to the symbol table and lists of predecessor and successor statements. This is the object code produced by the compiler. Figure 35 shows a small statement table. The INDEX field is the position of a statement in the table. Other tables use this index to access selected statements. The LABEL field contains the user defined labels (used in the ISP program or in the intermediate language input); it is kept for debugging only. The DEST, SRCE1, and SRCE2 field contain the symbol table indices of the Destination, Source1 and Source2 variables. The BLOCK field contains the Block Table index of the block where this statement is located. The PREDECESSOR and SUCCESSOR fields are lists of statement indices.

- Block table.- Contains an entry for each basic block. Entries in the block table include pointers to successor, predecessor, postdominator, and predominator

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
SYMBOL TABLE -- SYTABLE
INDEX,MTYPE,FLAGS        PNAME
    0     0        1     DUMMY
    1     0        1     OVFLG
    2     0        1     ZFLAG
    3     0        1     NFLAG
    4     0        1     PFLAG
    5     0        1     VFLAG
    6    44       10     COUNT
    7    44       10     P
    8    44       10     MPD
    9    42      100     8
```

Figure 34. Symbol Table

```
STATEMENT TABLE -- XTABLE
INDEX,LABEL,    OP, DEST,SRCE1,SRCE2,BLOCK,   FLAGS,NESTING, SUCCESSOR STATEMENTS,
     (BUS)                                                   PREDECESSOR STATEMENTS
    0      KSTART    0     0     0     0      110     1     ----
                                                             ----

    1        MOVE    6     9     0     0     1001     1     ( 2)
                                                             ----

    2     1   KSM    0     0     0     1      110     8     ----
                                                             ( 11  1)

    3        TEST    0     7     0     1        1     8     ----
                                                             ----

    4         KBR    0     2     0     1     1010     8     ( 6  5)
                                                             ----

    5     2 RSHFT    7     7     0     2     1101     8     ( 8)
                                                             ( 4)

    6     3   ADD    7     7     8     3     1101     8     ( 7)
                                                             ( 4)

    7       RSHFT    7     7     0     4     1101     8     ( 8)
                                                             ( 6)

    8     4   KSM    0     0     0     5      110     8     ----
                                                             ( 7  5)

    9        DECR    6     6     0     5     1001     8     ( 10)
                                                             ----

   10        TEST    0     6     0     6      101     8     ----
                                                             ( 9)

   11         KBR    0     2     0     6     1010     8     ( 2 12)
                                                             ----

   12   511 KSTOP    0     0     0     7     1110     1     ----
                                                             ( 11)
```

Figure 35. Statement Table

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

blocks, as well as lists of required, source, and destination variables. Figure 36 shows an example of a block table. The HEAD and TAIL fields contain the statement table entries of the head and tail operations, respectively. The PREDECESSOR, SUCCESSOR, PREDOMINATOR, and POSTDOMINATOR fields contain the respective sets of blocks. The SOURCE, DESTINATION, and REQUIRED fields contain the respective sets of variables. To allow greater control on the level of detail at which the transformations are tested and applied the user can specify the ***block size***. Blocks can be of two types, ***short blocks*** containing at most one data operation statement per block, and ***long blocks***, that group sequences of data operation statements in a single block. In the present implementation block sizes are required to be of the same type, either ***short*** or ***long*** (the largest possible sequence of data operation statements), no mixed types are allowed.

– Connectivity matrix.– A Boolean array indicating the connectivity between any two blocks, Fig. 37. This matrix is computed from the immediate predecessor and immediate successor lists kept in the block table.

– Distance matrix.– An array containing the minimal distance (number of edges) between any two blocks, Fig 38. This matrix is computed simultaneously with the connectivity matrix, using a modified version of Warshall's algorithm [Wars62].

– Interval table.– This table, Fig, 39, contains a hierarchical partitioning of the graph using the interval construction of Allen [Alle70]. Although they are not used in EXPL, the variable sets of the intervals are also computed.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BLOCK TABLE -- BTABLE

INDEX	HEAD	TAIL	BUS	FLAGS	PREDECESSOR BLOCKS, SUCCESSOR BLOCKS PREDOMINATOR BLOCKS, POSTDOMINATOR BLOCKS SOURCE SET, DESTINATION SET REQUIRED SET
0	0	1	0	0	---- (1,) ---- (1,5,6,7,) ---- (6,) ----
1	2	4	0	0	(0,6,) (2,3,) (0,) (5,6,7,) (7,) ---- (7,)
2	5	5	0	0	(1,) (5,) (0,1,) (5,6,7,) (7,) (7,) (7,)
3	6	6	0	0	(1,) (4,) (0,1,) (4,5,6,7,) (7,8,) (7,) (7,8,)
4	7	7	0	0	(3,) (5,) (0,1,3,) (5,6,7,) (7,) (7,) (7,)
5	8	9	0	0	(2,4,) (6,) (0,1,) (6,7,) (6,) (6,) (6,)
6	10	11	0	0	(5,) (1,7,) (0,1,5,) (7,) (6,) ---- (6,)
7	12	12	0	0	(6,) ---- (0,1,5,6,) ---- ---- ---- ----

Figure 36. Block Table

BLOCK CONNECTIVITY MATRIX -- CCMATRIX

	0	1	2	3	4	5	6	7
0=	0	1	1	1	1	1	1	1
1=	0	1	1	1	1	1	1	1
2=	0	1	1	1	1	1	1	1
3=	0	1	1	1	1	1	1	1
4=	0	1	1	1	1	1	1	1
5=	0	1	1	1	1	1	1	1
6=	0	1	1	1	1	1	1	1
7=	0	0	0	0	0	0	0	0

Figure 37. Connectivity Matrix

BLOCK DISTANCE MATRIX -- DMATRIX

	0	1	2	3	4	5	6	7
0=	0	1	2	2	3	3	4	5
1=	0	4	1	1	2	2	3	4
2=	0	3	4	4	5	1	2	3
3=	0	4	5	5	1	2	3	4
4=	0	3	4	4	5	1	2	3
5=	0	2	3	3	4	4	1	2
6=	0	1	2	2	3	3	4	1
7=	0	0	0	0	0	0	0	0

Figure 38. Distance Matrix

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
INTERVAL TABLE -- ITABLE
INDEX,   FLAGS, HEAD, DEGR, INTRVL, BLOCKS IN INTERVAL,
                                    PRED. BLOCKS,
                                    SUCC. BLOCKS,
                                    EXIT BLOCKS
                                    SOURCE VARIABLES,
                                    DESTINATION VARIABLES,
                                    REQUIRED-VARIABLES
  0       11      0     0     9    (0,)
                                    ----
                                    (1,)
                                    (0,)
                                    ----
                                    (6,)
                                    ----

  1       11      1     0     8    (1,)
                                    (0,6,)
                                    (2,3,)
                                    (1,)
                                    (7,)
                                    ----
                                    (7,)

  2     1000      2     0     0    (2,)
                                    (1,)
                                    (5,)
                                    (2,)
                                    (7,)
                                    (7,)
                                    (7,)

  3     1000      3     0     0    (3,)
                                    (1,)
                                    (4,)
                                    (3,)
                                    (7,8,)
                                    (7,)
                                    (7,8,)

  4     1000      4     0     0    (4,)
                                    (3,)
                                    (5,)
                                    (4,)
                                    (7,)
                                    (7,)
                                    (7,)

  5     1000      5     0     0    (5,)
                                    (2,4,)
                                    (6,)
                                    (5,)
                                    (6,)
                                    (6,)
                                    (6,)

  6     1000      6     0     0    (6,)
                                    (5,)
                                    (1,7,)
                                    (6,)
                                    (6,)
                                    ----
                                    (6,)

  7     1000      7     0     0    (7,)
                                    (6,)
                                    ----
                                    (7,)
                                    ----
                                    ----
                                    ----

  8     1000      1     1     0    (1,2,3,4,5,6,7,)
                                    (0,)
                                    ----
                                    (7,)
                                    (6,7,8,)
                                    (6,7,)
                                    (6,7,8,)

  9        0      0     2     0    (0,1,2,3,4,5,6,7,)
                                    ----
                                    ----
                                    (7,)
                                    (6,7,8,)
                                    (6,7,)
                                    (7,8,)
```

Figure 39. Interval Table

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
REGION TABLE -- RTABLE
INDEX,   FLAGS,FATHER,ENTRY,EXIT,  BLOCKS IN REGION,
                                   OUTHER REGIONS,
                                   IMMEDIATE INNER REGIONS
   0     1100 -----     1     6    (1,2,3,4,5,6,)
                                   ----
                                   ----
```

Figure 40. Region Table

- Region table.- This table, Fig. 40, contains the strongly connected regions of the graph. The regions are used to extract the level of loop nesting of the basic blocks. The regions are obtained from the intervals by using an algorithm described by Allen.

- Group table.- This table, Fig. 41, contains all possible subgraphs used as parameters by the testing and transformation routines. It includes information about required, source, and destination variables. The groups are created from the basic blocks. The first set of groups consists of the block themselves. These groups are made to "grow" by adding, one at a time, the successor blocks of the blocks already in the group (eliminating duplicated groups and groups with more than one entry block). This process is repeated, creating larger groups from the previously found groups.

- Case table.- This table, Fig. 42, contains information about all possible cases of transformations that can be applied to the graph. There is one entry for each possible application of a rule. The entries contain the rule identification and the groups involved. This table also contains some preliminary evaluation of the effect of the rule. This information is a "best" case evaluation. i.e. the least cost added and the maximun time gained.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

GROUP TABLE -- GTABLE

INDEX,	FLAGS,	HEAD,	DEGR,	BLOCKS IN GROUP, PRED. BLOCKS, SUCC. BLOCKS, EXIT BLOCKS SOURCE VARIABLES, DESTINATION VARIABLES, REQUIRED VARIABLES
0	0	6	1	(6,) (5,) (1,7,) (6,) (6,) ---- (6,)
1	0	5	1	(5,) (2,4,) (6,) (5,) (6,) (6,) (6,)
2	0	4	1	(4,) (3,) (5,) (4,) (7,) (7,) (7,)
3	0	3	1	(3,) (1,) (4,) (3,) (7,8,) (7,) (7,8,)
4	0	2	1	(2,) (1,) (5,) (2,) (7,) (7,) (7,)
5	0	1	1	(1,) (0,6,) (2,3,) (1,) (7,) ---- (7,)
6	0	6	2	(6,7,) (5,) (1,) (6,7,) (6,) ---- (6,)
7	0	5	2	(5,6,) (2,4,) (1,7,) (6,) (6,) (6,) (6,)
8	0	3	2	(3,4,) (1,) (5,) (4,) (7,8,) (7,) (7,8,)
9	0	1	2	(1,3,) (0,6,) (2,4,) (1,3,) (7,8,) (7,) (7,8,)
10	0	1	2	(1,2,) (0,6,) (3,5,) (1,2,) (7,) (7,) (7,)
11	0	5	3	(5,6,7,) (2,4,) (1,) (6,7,) (6,) (6,) (6,)
12	0	1	3	(1,3,4,) (0,6,) (2,5,) (1,4,) (7,8,) (7,) (7,8,)
13	0	1	3	(1,2,3,) (0,6,) (4,5,) (2,3,) (7,8,) (7,) (7,8,)
14	0	1	4	(1,2,3,4,) (0,6,) (5,) (2,4,) (7,8,) (7,) (7,8,)
15	0	1	5	(1,2,3,4,5,) (0,6,) (6,) (5,) (6,7,8,) (6,7,) (6,7,8,)
16	0	1	6	(1,2,3,4,5,6,) (0,) (7,) (6,) (6,7,8,) (6,7,) (6,7,8,)
17	0	1	7	(1,2,3,4,5,6,7,) (0,) ---- (7,) (6,7,8,) (6,7,) (6,7,8,)

Figure 41. Group Table

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
POSSIBLE CASES -- PRTABLE
INDEX,RULE,VALUE,    GRP1,VAL1,      GRP2,VAL2,      GRP3,VAL3,      FLAGS
    0 SP   (0,0)      14  (51,23800)  7  (30,18280)  0  (0,0)             0
    1 SP   (0,0)      14  (51,23800)  1  (16,14760)  0  (0,0)             0
```

Figure 42. Cases Table

The initial description given to EXPL consists of the symbol table and the statement table. The other tables are created by a bootstrap process from the statement table.

Since the transformations affect the connectivity of the graph, by redefining the pointers in the statement table, once a transformation is applied the other tables must be updated to represent the new graph. These tables are actually recomputed, rather than patched.

This process of creation of the tables is performed in two steps. In the first step, the blocks, intervals, and regions are computed. These tables contain all the information needed to evaluate the design. The second step creates the group and case tables. They are used to find the successor alternatives. The creation of this second set of tables is time consuming and can be avoided if the design is rejected after the evaluation.

IV.3 EXPL ORGANIZATION

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

An initial description of EXPL was presented in section I.4. The current section provides additional details. The system is centered around the set of tables described in section IV.2. The interface between the system and the user is given by a driver module that interpretes user commands. These commands are used to direct the creation of the tables. Associated with each data structure there are two routines. These routines are used to create and display the tables in a readable format. Besides these, each table has associated with it specific routines that perform table dependent actions.

The main commands are:

{SET | TRADE} parameter

These commands are used to define global parameters used by the individual routines. The parameter defines modes of behavior for the routines, including goals used by the heuristic search (i.e. cost/speed trade offs).

CREATE table-name

This command executes the routine used to create a particular table. Since the tables must be created in a certain order, a null table-name is interpreted as a sequence of CREATE commands that bootstrap the building of the tables in the following order: blocks, intervals, regions, groups, and cases.

DISPLAY table-name table-entry

This command displays selected entries in any of the tables. The output of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

command is directed to the user's terminal. The table-entry can be a single integer or a bound pair (integer : integer). The latter is used to display consecutive entries.

PRINT {table-name | GRAPH}

This command is similar to DISPLAY. The entire table is printed on the output device specified by the OPEN command. A null table-name is interpreted as a sequence of PRINT commands that print all the system tables. This comand is useful if the amount of output is large (the output can be directed to a line printer) or if the user wants it for later use (as a disk file). The GRAPH parameter is used to print the flowchart representation, thus providing a graphical description, sometimes more useful than the tabular representation.

{OPEN | CLOSE} file-specs

These two commands are used to direct the output of the printing routines to user selected devices. The file-specs are the usual PDP-10 "dev:filename.ext".

{TEST | DO} rule-name group-list

These two commands test and apply transformations on the graph. The TEST command executes a routine associated with the rule. This routine will return a yes/no answer depending on the applicability of the transformation on the groups given as parameters. The DO command executes the routine that performs the reconnection of the graph.

VALUE {table-name table-entry | SYSTEM}

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

This command invokes the RTM evaluation routines. These routines return the cost and time requirements of statements, blocks, intervals, regions, groups, or the entire graph.

{DUMP | RESTORE} node-name

These two commands are used to store and retrieve the tables that describe a design. They provide a simple way of creating and using a data base of designs. The nodes are store as disk files with the following specifications: "DSK:node-name.NDE".

WALK node-id

This command invokes a user written routine that performs a heuristic search of the design space. One such program exists in the system as the default search algorithm. It is described in the next section. This default algorithm can be replaced and the system assembled together again. The user can write such a program without a thorough knowledge of the inner details of the system, or the algorithms used. The program would consist of a set of calls to the system routines, simulating a sequence of user commands. The values returned by the system programs can be used to select alternative actions (sequences of simulated commands). In other words, the program can set goals and global parameters; it can create, save, and restore the tables that describe a design; it can evaluate designs and perform alternative actions, depending on the values.

The organization of the system is flexible and allows it to be used as an open

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

system, where the user drives every facility from a time sharing terminal, or as a closed system by providing a suitable "WALK" routine that will operate without interaction with the designer.

IV.4 THE HEURISTIC SEARCH

Due to the interaction between transformations it is a difficult task to formalize the optimization (improvement of alternative structures) as a mathematical optimization problem. The main difficulty is the fact that transformations apply to subgraphs of arbitrary size and, as a consequence, transformations in a given alternative structure may or may not be feasible or desirable in structures derived from it. It is also the case that new cases of transformations become feasible or desirable only after a specific sequence of transformations has been applied.

IV.4.1 The design space exploration.- The design space exploration can be characterized as a space-search problem [Nils71]. The characteristics of the methods used are:

- A data structure is used to represent *states* that resemble some physical property of the problem being solved. The data structures allow easy computations by *operators* that transform one state description into another.

- Operators are (partial) functions whose domain and range are states. They

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

transform state descriptions into new state descriptions.

- Goal states are defined and used to indicate the end of the search when they are reached.

In EXPL, the state description is given by the collection of tables described in section IV.2. The operators are the set of graph transformations. The goal states are defined by the user in terms of cost and time trade offs.

The design space is represented by a time/cost diagram. Alternative structures are represented by points in the diagram. Except for the original solution, all points are derived, by transformations, from other points in the space. These relationships will be made explicit by drawing vectors from the parent nodes to their immediate (i.e. one transform removed) descendents.

The exploration of the design space in EXPL is performed by a group of heuristic routines that produce alternative designs in a goal oriented fashion, the goal being specified by the designer. Ideally, the goal is to find an alternative structure whose position in the design space is as close as possible to the origin (0 cost and 0 time). This ideal case is, however, not easily found in real solutions. The usual case is that the least expensive solution is not the fastest and vice versa. This characteristic provides a rough classification of the design objectives into two classes: minimal cost and minimal time.

Although a designer's aim can be classified according to these objective functions

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

it may be the case that the real objective is more complicated in nature, namely, some combination of time and cost. For instance, the objective could be something like: "the fastest alternative structure not costing more than x dollars".

For simplicity, the subspace of acceptable solutions will be defined by a set of straight line segments whose slopes reflect the objective functions. In the simples case, a single straight line, orthogonal to the main goal axis would be used to divide the space in two halves. Only those solutions that lie in the semispace containing the origin are considered acceptable. These solutions represent improvements along the design goal.

More complex constraints can be described by using lines of the form $\$ = -m.T + b$, where m is a parameter indicating how many dollars the designer is willing to pay for each time unit saved (if time is the primary goal) or how many time units the designer is willing to sacrifice for each dollar saved (if cost is the objective). An example, Fig. 43, will clarify this description.

Assume that the primary objective is a reduction in time, and that the designer wants a time/cost trade-off of at most m dollars for each time unit improvement. Furthermore, assume that the original design is characterized by $1 and T1. The "acceptable trade-off" subspace would thus be delineated by two line segments: one parallel to the cost axis starting from (T1,$1) to (T1,0), and the other through (T1,$1) with slope $-m$. Assume that by studying the control flow and data dependencies in this original structure, four transformations are available which yield four alternative solutions derived from the original one: A,B,C,D.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

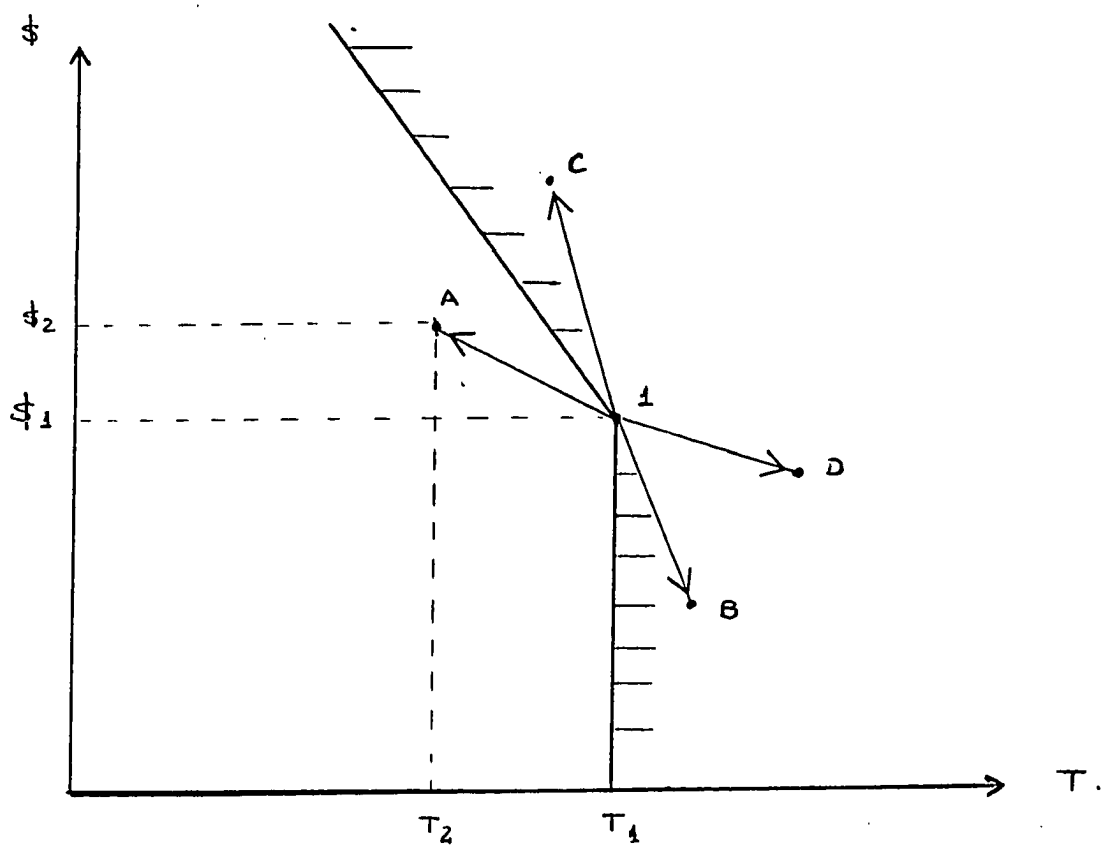


Figure 43. Design Space Reduction

By dividing the space according to the trade-off lines, alternatives B, C, and D can be rejected because their characteristics are not within the acceptable subspace (i.e. they take more time or the decrease in time costs too much). The alternative left, A, represents improvement in time while the cost to achieve the improvement is under the designer's threshold.

The process can now be applied to A in an identical manner. Design A is taken as the new initial solution and a new "acceptable trade-off" subspace is defined by a

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

line segment (T2,$2) to (T2,0) and a line with slope −m through (T2,$2). Since in some cases more than one alternative can be left for further exploration, this process takes the form of a tree walk where the nodes represent alternative solutions and the edges are the transformations applied. In some instances, identical structures can be obtained by different sequences of transformations and the exploration of the design space is a graph walking process. In any event, a path ends when no alternative solutions worth exploring can be reached from a given point. When all possible paths have been explored the end nodes are measured against the primary objective and the best one chosen.

IV.4.2 Design space reduction.– Depending on the nature of the problem, it may be the case that an excessive number of possible successor structures can be found. To reduce the space two types of pruning techniques are used.

The first type, *strategic*, accepts or rejects alternative structures based on global analysis of the cost and time requirements. This type of pruning has been described in the previous paragraphs. This strategic pruning can be applied only after the successor structure has been obtained, at least to a point where it can be evaluated, i.e., only after the computation of the block, interval, and region tables. This partial computation of the full state description can still represent an excessive amount of overhead, which could be avoided by a closer look at the transformation that generates the successor alternative. This second type of pruning, *tactical*, is based

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

upon the preliminary evaluation of the effect of the rule, stored in the case table. Depending upon this preliminary evaluation, a transformation can be rejected even before it is applied. Since this evaluation is based on a best case situation, the system can confidently reject a transformation knowing that its application would not produce a structure acceptable to the strategic pruning routines. This tactical pruning is a case analysis of the proposed transformations. It consists of a series of tests performed on the time requirements of the subgraphs (groups) involved. No tactical pruning is done based on cost requirements since the cost of a structure is heavily determined by that of the data part, determined at a global level (i.e. only after the transformation is applied). Other applications of the tactical decisions include the replacement of instances of rule SPS1 by instances of rule SPS0. The conditions for this replacement were described in section III.3.4. Tactical decisions are also used to eliminate transformations when the best possible gain in time is less than some threshold (1% by default, although the user could specify his own).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

V. EXPERIMENTAL RESULTS

In this chapter several examples of the use of EXPL are examined to give some indication of the characteristics of the system as a design aid as well as the characteristics of the design space for RTM's.

The first example deals with the design of an 8-bit multiplier using the familiar shift and add algorithm. This is a rather simple example and was used extensively by Bell et. al. [Bell72b] to describe the RTM design trade-offs. The authors of the book performed an almost exhaustive search of the variations that could be obtained from the basic algorithm. Some of the techniques used, like the use of table look-up, sharing of functional units, loop unwinding, pipelines etc., could not be achieved by the use of the transformations described in this thesis, as was pointed out in chapter III. Nevertheless, the system was able to obtain an extra solution that was not contemplated in [Bell72b]. Although this additional solution is neither the fastest nor the cheapest, its discovery, by straightforward application of the transformations bears out the usefulness of a system like EXPL, even in small problems.

The second and third examples deal with the design of the control for a conveyor-bin system. Example 2 searches the design space starting from a sequential description of the algorithm and its goal is to find a faster implementation. The design space in this example is larger (number of points) than that of the multiplier and shows some characteristics of the design space and the effect of the the strategical pruning. Example 3 deals again with the conveyor-bin system, but in this case the initial

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

solution was modified to include many concurrent activities. This third example was processed by EXPL using as goal the least expensive design. The design space presents many more points than the previous examples raising issues about the efficiency of blind searches and how they can be avoided by enabling or disabling sets of applicable transformations.

V.1 AN RTM MULTIPLIER

The multiplier is to be used as part of a larger system, like a subroutine. The data consists of two 8-bit integers, and the result is a 16-bit integer. Two variables, MPD (multiplicand) and P (multiplier) are used. The product is developed in P. In order to improve its efficiency, the algorithm uses an unconventional data format, namely, the multiplicand is assumed to be stored in the leftmost eight bits of the MPD register, while the multiplier is in the rightmost eight bits of the P register. Using this format, it is possible to shift the multiplier (in order to select the next multiplier bit) and the product (so that the partial results can be added properly) in one step. A counter, C, is used to keep track of the number of times the basic shift and add cycle has been performed.

V.1.1 The initial solution.- The following ISP program describes the algorithm. The flowchart obtained from this program is shown in Fig. 44.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
MPD<15:0>
P<15:0>
C<15:0>

MULTIPLIER := (
        C←8; next
        L1 := (  (¬P<0>⇒P←P/2);
                 (P<0>⇒P←(P+MPD)/2); next
                 C←C−1; next
                 (C≠0⇒L1)
             )
        );
```

Several issues regarding the correctness of the algorithm and the handling of improper data are described in the reference. For our purposes we can be satisfied with the algorithm as described. The basic solution graph has a simple RTM implementation. Since no concurrency exists in the algorithm, a single bus system is sufficient. A DMgpa is required to perform the data operations i.e. adding, subtracting, and shifting. A Mconstant is required to provide the constant 8, used to initialize the counter C. Three registers from a Mscratchpad are used to hold the data and result. The control modules are also allocated in a straightforward method. The initial RTM solution is shown in Fig. 45. The figure shows the type of templates used by the evaluation routines. It is apparent from the figure the type of improvements that could be obtained by optimizing the use of the local memory of the DMgpa module and by building into the system a greater knowledge about the RTM set. This point was mentioned before, in chapter IV, but it is worth discussing again. The dashed lines in Fig. 45 indicate the set of RTM operations assumed by the templates. Test and branch operations are implemented as a sequence of two steps:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

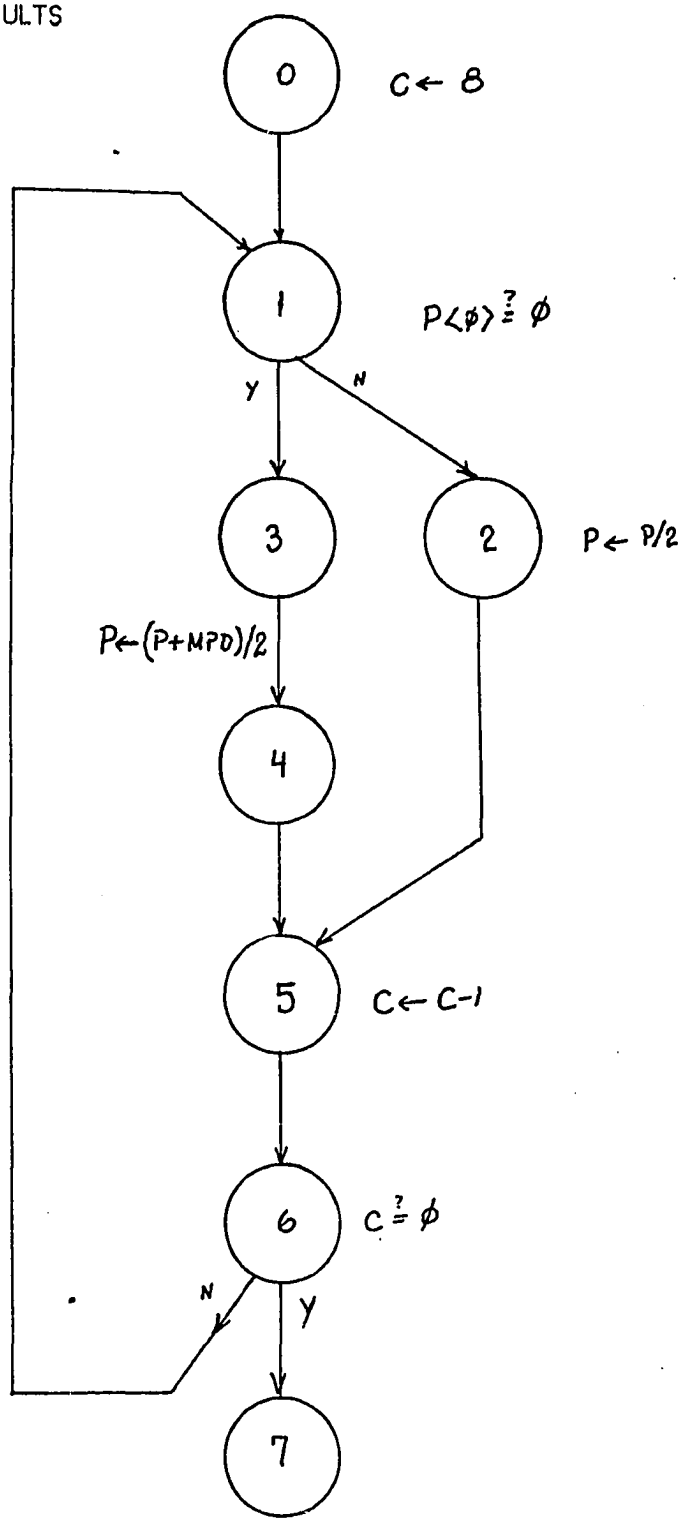


Figure 44. Multiplier Flowchart

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

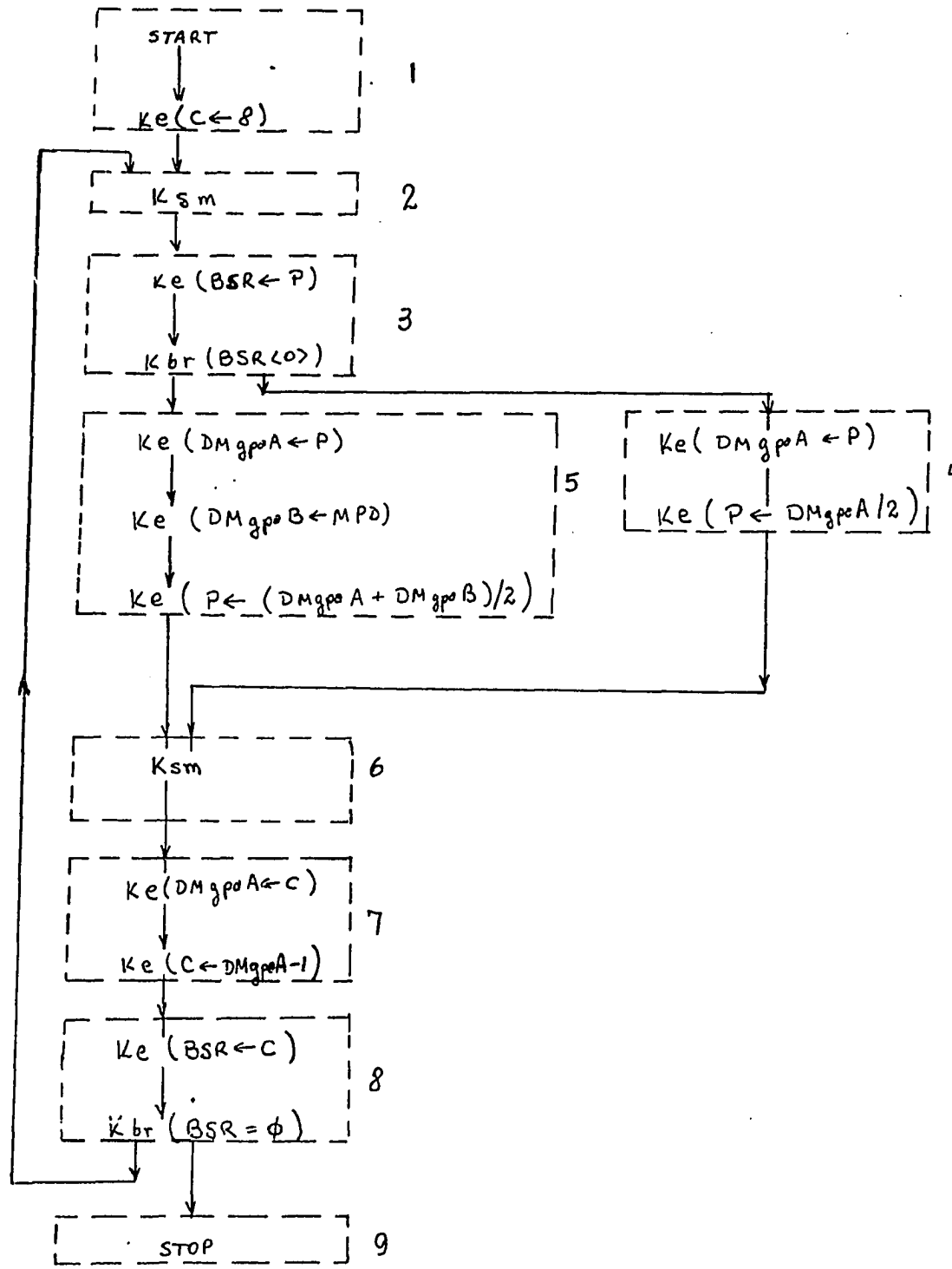
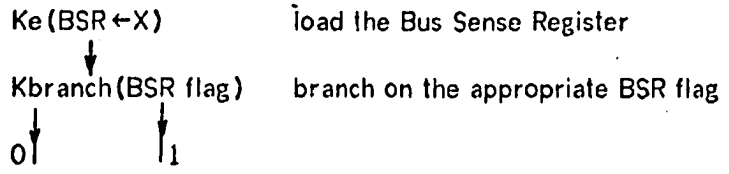
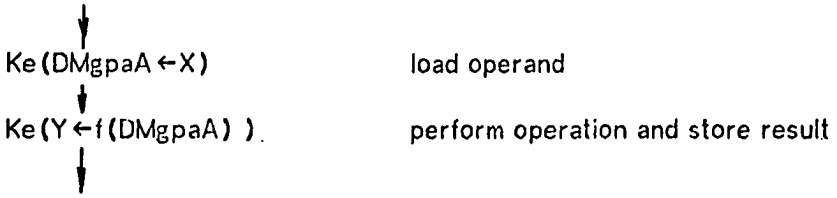


Figure 45. RTM Multiplier – Templates

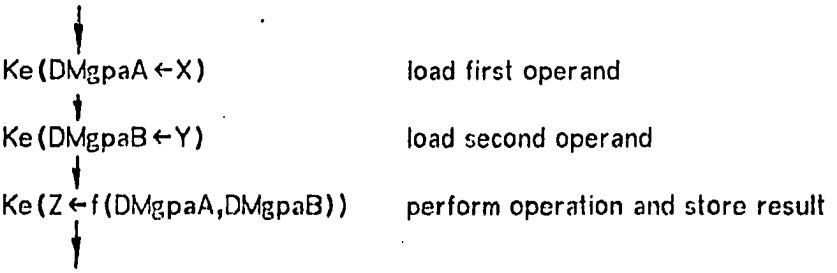
Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.



Arithmetic and logic operations are implemented always as a sequence of two or three steps, depending on the number and type of the operands. Thus, for unary operations the template is:



For binary operations the template is:



Now is perhaps clear what was meant by a "naive" implementation of the flowchart. For instance, the first operation of box 8, Ke(BSR←C), can be eliminated since in RTMs the result of any operation that involves the bus always leaves the last value transmited in the BSR. In other words, the value of C, computed in the last operation of box 7 is already available in the BSR. A different type of optimization can

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

be obtained by making use of the DMgpa registers. The first operation of boxes 4 and 5 indicate the loading of the DMgpaA register with the value of P. If we replace the first operation of box 3 , Ke(BSR←P), by Ke(DMgpa←P), then we can eliminate the two similar operation in boxes 4 and 5. In this case we are using the local memory (DMgpaA) to hold a value during an intermediate step, the Kbranch operation. In this optimization we are using again the fact that the BSR is automatically loaded. This type of optimization can also be described in terms of code motion, in this case, the Ke(DMgpaA←P) operations on both branches of the Kbranch module are moved upwards, beyond the branch operation, and then combined with the Ke(BSR←P).

V.1.2 Variations in the basic algorithm.- Several alternatives in the implementation of the basic algorithm can be found. Examination of the flowchart (Fig. 44) indicates that the computation of the loop count (block 5) does not depend on the shifting and adding steps (blocks 1,2,3, and 4). They do not have any variables in common. Thus, the decrement of the loop counter can be performed in parallel with the elementary multiplication step. Fig. 46 shows this application of rule SP.

The resulting graph shows that the testing of the loop count, although independent of the multiplication steps, can not be performed in parallel with the decrement of the loop count. This fact rules out a transformation like rule SPS1. However, rule SPS0 is a possible transformation, by which the testing of the loop counter is performed after the decrement, but both concurrently with the basic multiplication step. The resulting graph is shown in Fig. 47.

No more transformations can be applied to the graph, and it represents an

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

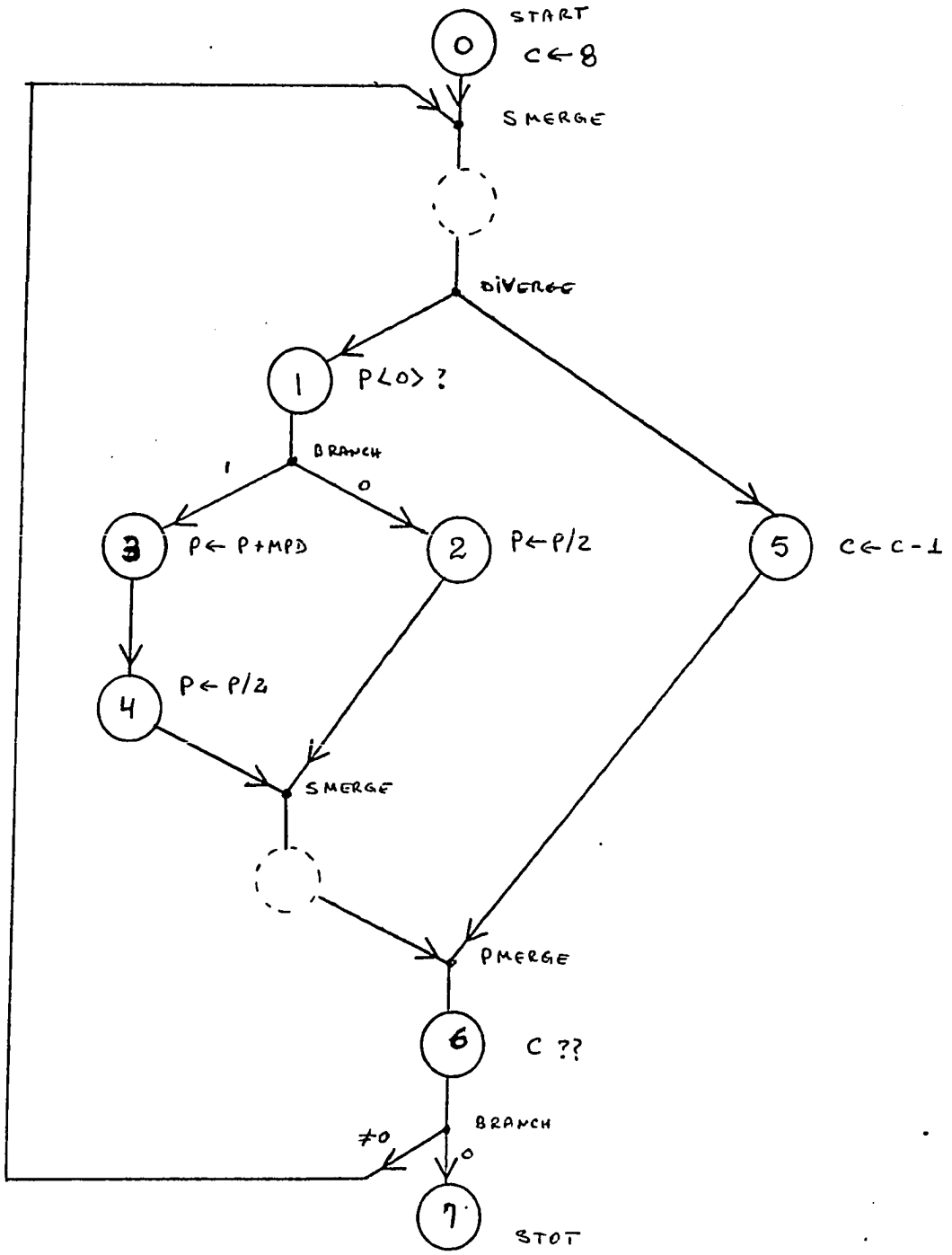


Figure 46. Multiplier Transformations – 1

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

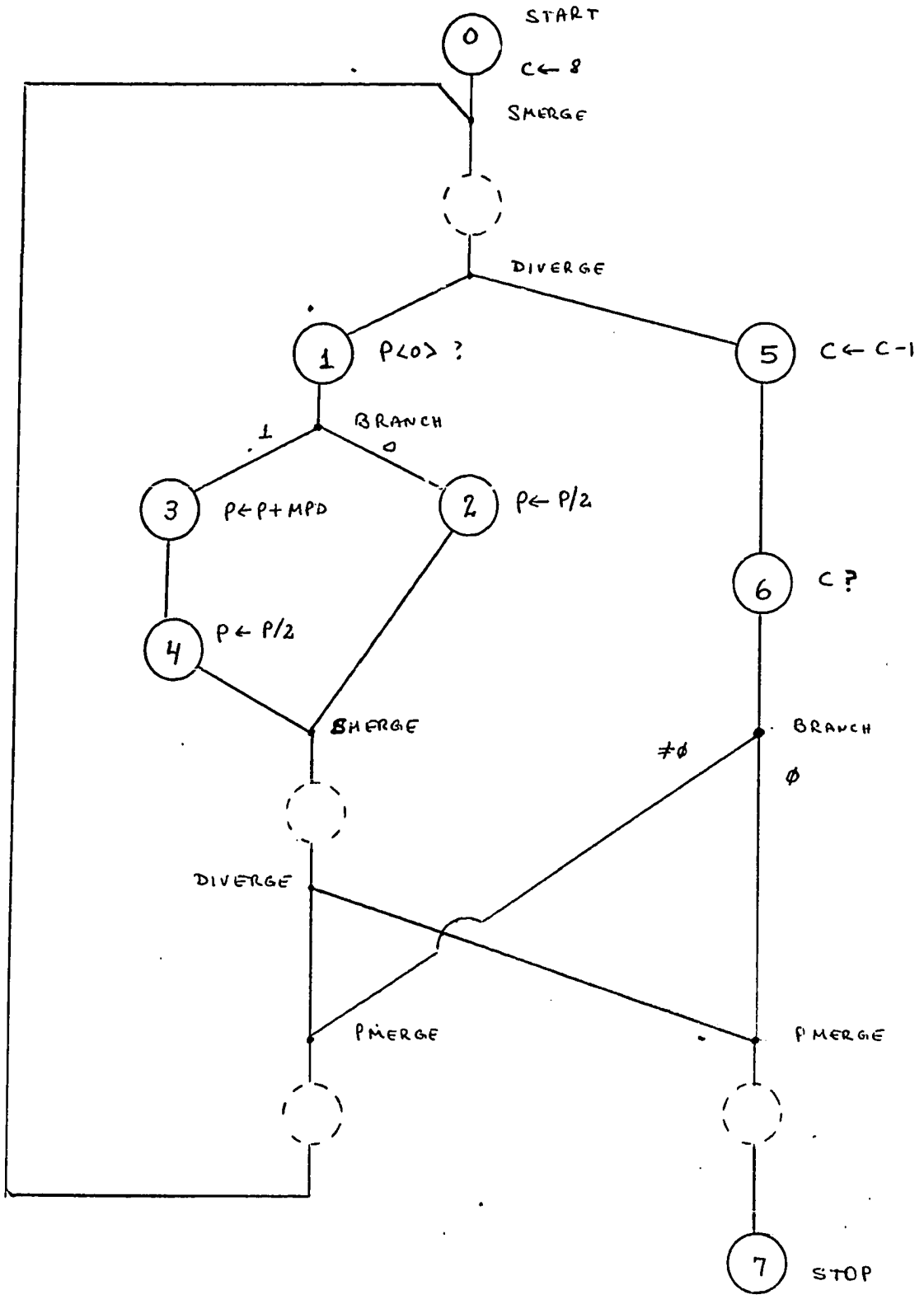


Figure 47. Multiplier Transformations – 2

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

optimal time implementation (we are not considering here specific RTM dependent optimizations, for instance, the use of the local memory of the DM modules as permanent storage for P, MPD, and C). Although it took two steps (transformations) to arrive to this final graph, we could have taken blocks 5 and 6 of the original graph as a group, and arrived to the same final graph in one step (one application of rule SP).

These series of graph transformations have taken us from an original solution, to two additional design alternatives. Both represent an improvement in the time required to perform the basic algorithm. The design space, explored automatically by the EXPL heuristic search routines, presents these alternatives as points in a design space. These alternatives and the paths followed are shown in Fig. 48.

Point 0 corresponds to the initial solution (Figs. 44 and 45). Point 2 corresponds to the flowchart shown in Fig. 46, this is the solution that was not covered in [Bell72b]). Point 1, coincident with point 3, corresponds to the flowchart shown in figure 47.

It can be argued that point 2 does not present any kind of reasonable alternative between points 0 and 1. This is probably so, but only because the cost of the buses and functional units in RTM is by far the predominant element in the total system cost. The transition from point 2 to point 3 is the trade-off between the time of the branch operation versus the cost of the Kparallel.merge module.

A trace of the exploration is shown in Fig. 49. It depicts the identification of

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

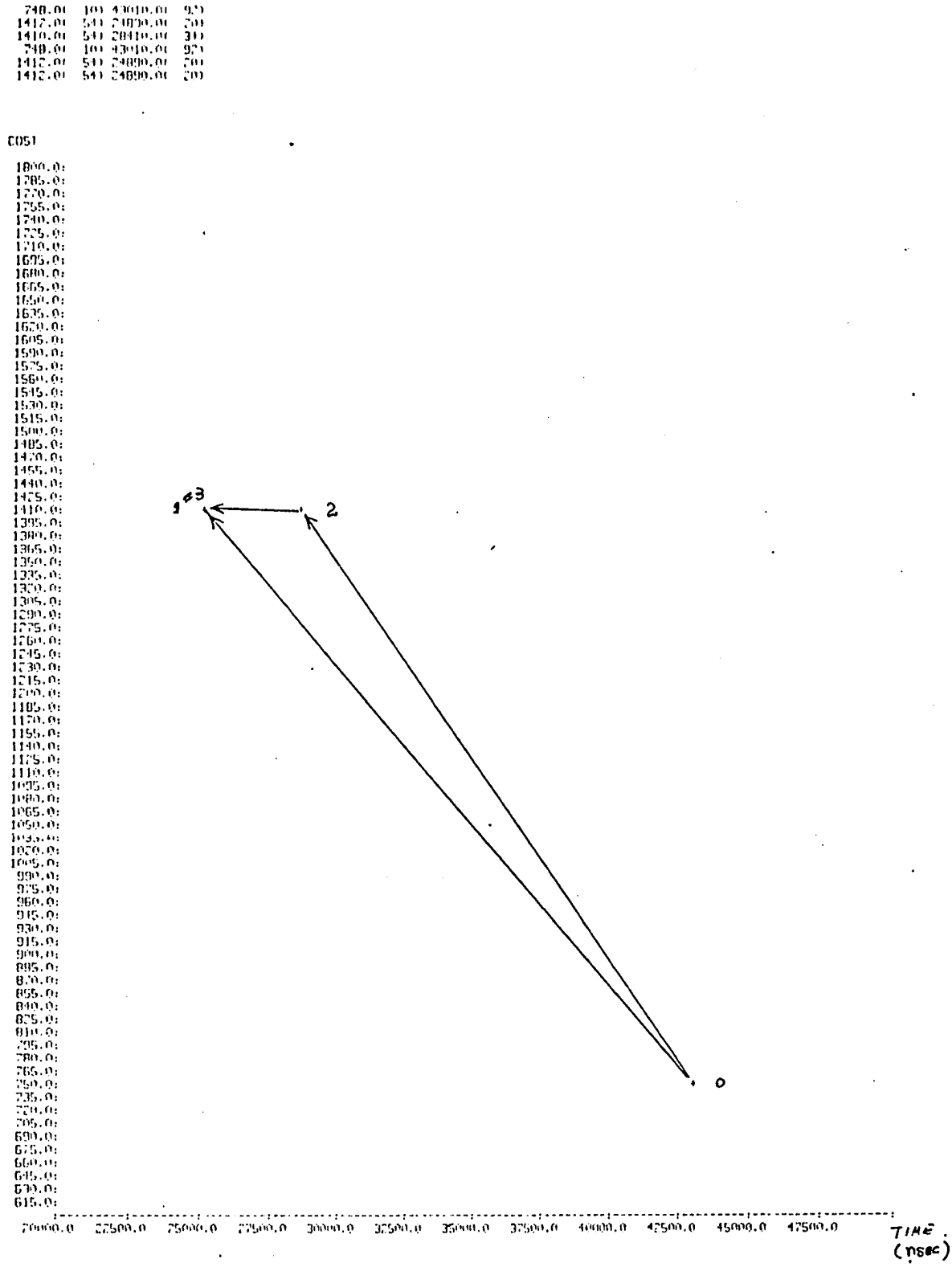


Figure 48. Multiplier Design Space

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
VAR;ALLRULES;SPEED;TRADE=1000.00;SHORT;LOCAL;


00100 IN STMT=12;BLK=7;INT=9;REG=0;GRP=17;CSE=1
              ;CST=748;TME=43010;DST=21879.01
    0 SP  (0,0)      14  (51,23800)  7  (30,18280)  0  (0,0)          10

CK0  2
CK0  8

00101 IN STMT=16;BLK=11;INT=13;REG=0;GRP=41;CSE=-1
               ;CST=1412;TME=24890;DST=13151.01
00101  OUT

00100  BACK
    1 SP  (0,0)      14  (51,23800)  1  (16,14760)  0  (0,0)          10

CK0  2
CK0  8
CK0 16

00102 IN STMT=14;BLK=10;INT=12;REG=0;GRP=33;CSE=2
               ;CST=1410;TME=28410;DST=14910.01
    1 PS0 (0,0)       1  (14,14680) 27  (51,23800)  0  (0,0)           1
****DELETED

    0 PS0 (0,0)      27  (51,23800)  1  (14,14680)  0  (0,0)          10

CK0 12
CK0 14
CK0 15

****REJECT  STMT=12;BLK=9;INT=11;REG=0
              ;CST=748;TME=43010;DST=21879.01
    2 SPS0(0,0)      27  (51,23800)  1  (14,14680)  6  (16,3600)       10

CK0 14
CK0 15

00103 IN STMT=16;BLK=12;INT=14;REG=0;GRP=45;CSE=-1
               ;CST=1412;TME=24890;DST=13151.01
00103  OUT

00102  BACK
00102  OUT

00100  BACK
00100  OUT

BEST NODE=00101
              STMT=16;BLK=11;INT=13;REG=0;GRP=41;CSE=-1
              ;CST=1412;TME=24890;DST=13151.01
```

Figure 49. Multiplier Design Space Exploration – Trace

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the node under examination, the size of the tables describing the node, the cost and time requirements, the rules being considered for application, the result of the tactical and strategical prunning, and finally, the identification (and characteristics) of the best solution. The first line of the trace displays the global parameters that are used by the system routines.

V.2 A CONTROLLER FOR A CONVEYOR-BIN SYSTEM

Briefly, the algorithm performs the controlling function for a conveyor carrying items to be sorted into bins.

"A production process creates items to be placed in 63 bins. When an item is placed on the conveyor, it is given an item number which identifies the bin into which it is to be placed. Faulty items are also placed on the conveyor and transfered to a reject bin. It is assumed that the conveyor moves in discrete steps, and after each step, a new item is placed on the conveyor and all items move up one position. If items opposite new bin positions have the corresponding item number, they are ejected into the corresponding bins. A 128 word array, M, is used to keep the state of the conveyor (the item numbers on the conveyor) and the bins (the number of items in a bin). M[1:63] contain the bin counts. M[65:127] correspond to the 63 positions of the conveyor. M[0] corresponds to the reject bin. Instead of simulating the movement of the conveyor by shifting the words M[65:127], a Conveyor- Head- Position\CHP pointer is used to point to the head of the conveyor

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

at any given time. CHP is stored in M[64], and can be considered as a modulo 63 counter. New items enter the conveyor at the cell pointed at by CHP" [Bell72b].

V.2.1 The initial solution.– The algorithm is described in ISP and its graph model is shown in Fig. 50. Notice that in this example the nodes correspond to sequences of one or more operations.

```
L1 := ( ADVANCE.CONVEYOR.FLAG ⇒
        M[64] ← M[64] – 1 ; next
        (M[64] = 64 ⇒ M[64] ← 127) ; next
        T ← M[64] ; next
        M[T] ← ITEM.NUMBER ; next
        I ← 1 ; next
        L2 := (BIN ← I ; next
                (BIN = M[T] ⇒
                        (M[I] = 0 ⇒ FULL.BIN.FLAG ← 1 ; next
                                M[0] ← M[0] + 1 ; next
                                (M[0] = 30 ⇒ OVER.30.ALARM ← 1) );
                        (M[I] < 0 ⇒ EJECT.TO.BIN.SIGNAL ← 1);
                        (M[I] > 0 ⇒ M[I] ← M[I] – 1 ; next
                                EJECT.TO.BIN.SIGNAL ← 1) ) ; next
                T ← T + 1 ; next
                (T > 127 ⇒ T ← 65) ; next
                I ← I + 1 ; next
                (I = 64 ⇒ L2, L1) )
```

Register T (temporary) holds the physical number of the conveyor cell being considered. The controller waits for the Advance Conveyor signal from the conveyor, then decrements CHP and inserts the item number in the corresponding cell. It then checks the 63 positions of the conveyor to find those items that may be ejected into the bins. The bin counts are assumed to be initialized to the maximum number of items the bin can hold. If this number reaches 0, the items addressed to the bin are rerouted

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

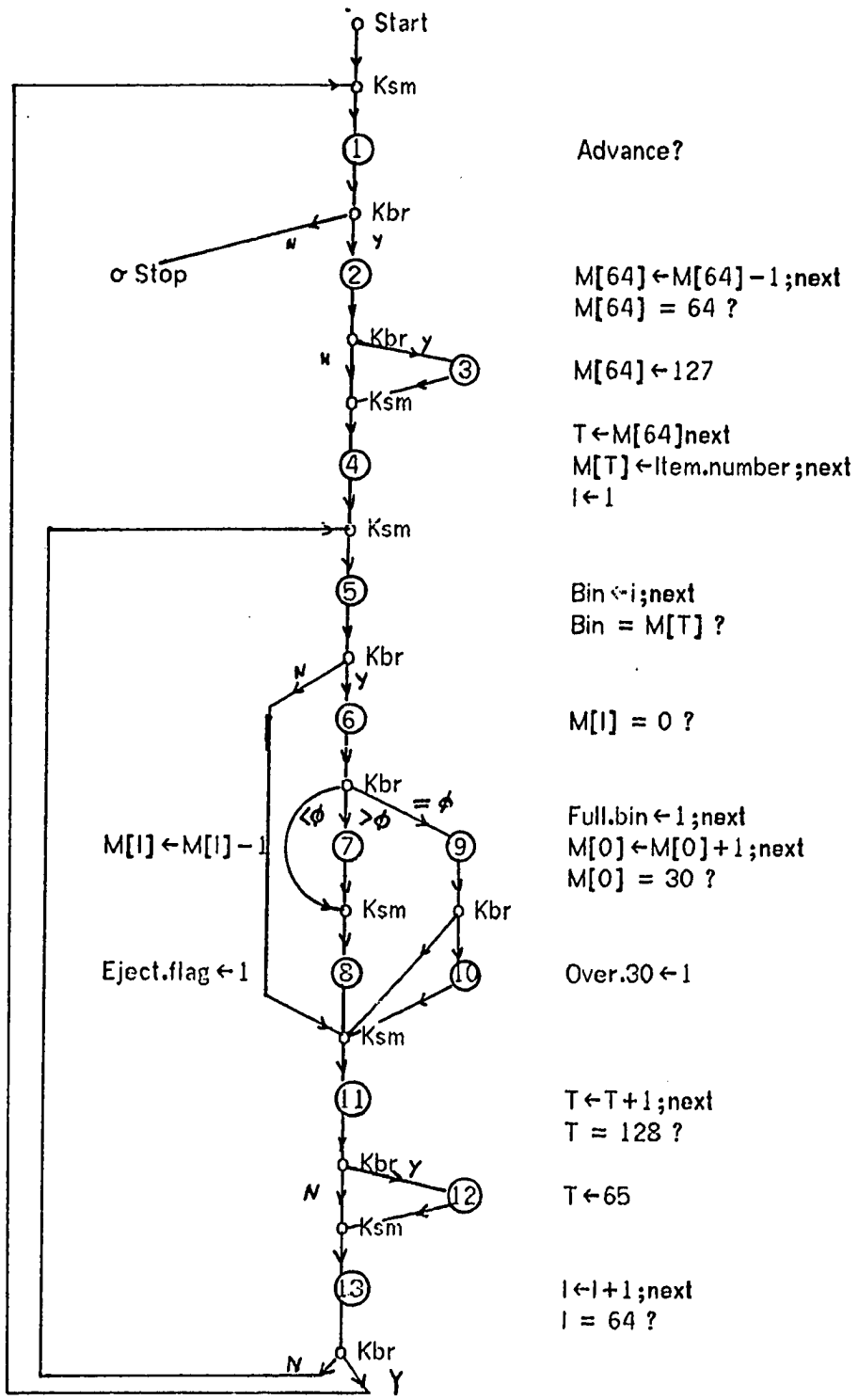


Figure 50. Controller for a conveyor–bin system

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

to the reject bin. An initial bin count given by a negative number indicates an infinite capacity bin.

V.2.2 Alternative implementations.- Several alternative implementations can be derived from this example. They are rather simplistic due to the compactness of the algorithm, but they are nevertheless appropriate to show the design space and its exploration, Fig. 51. First, assume that the fastest solution is sought. All the applicable transformations deal with the increment and testing of variables T and I (nodes 11,12, and 13 of the flowchart), and their concurrent execution with the main computation (nodes 5,6,7,8,9, and 10).

The best solution (timewise) is given by point 4 in the design space. In this solution, the main body of the algorithm (5,6,7,8,9,10) is computed in parallel with the increment and testing subprocess (11,12,13) as a whole. Other alternative points are also shown in the diagram (points 1,2,3,5). Several things can be noticed in the design space diagram; for instance, point 2, the parallel computation of (5,6,7,8,9,10), (11,12), and (13) is reached in two ways: First, (5,6,7,8,9,10,11,12) is performed in parallel with (13), point 1, and then the larger computation is performed as (5,6,7,8,9,10) in parallel with (11,12). The other way of reaching point 2 is by computing (5,6,7,8,9,10) in parallel with (11,12,13), point 4, and then transforming the smaller subgraph into (11,12) in parallel with (13). Notice furthermore, that points 2 and 4 present the same time value. In this case, EXPL uses the cost as a tie breaker parameter. Figure 52 shows the trace of the design space exploration. The trace shows several characteristics of the exploration: 1) Point 2, when reached from

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

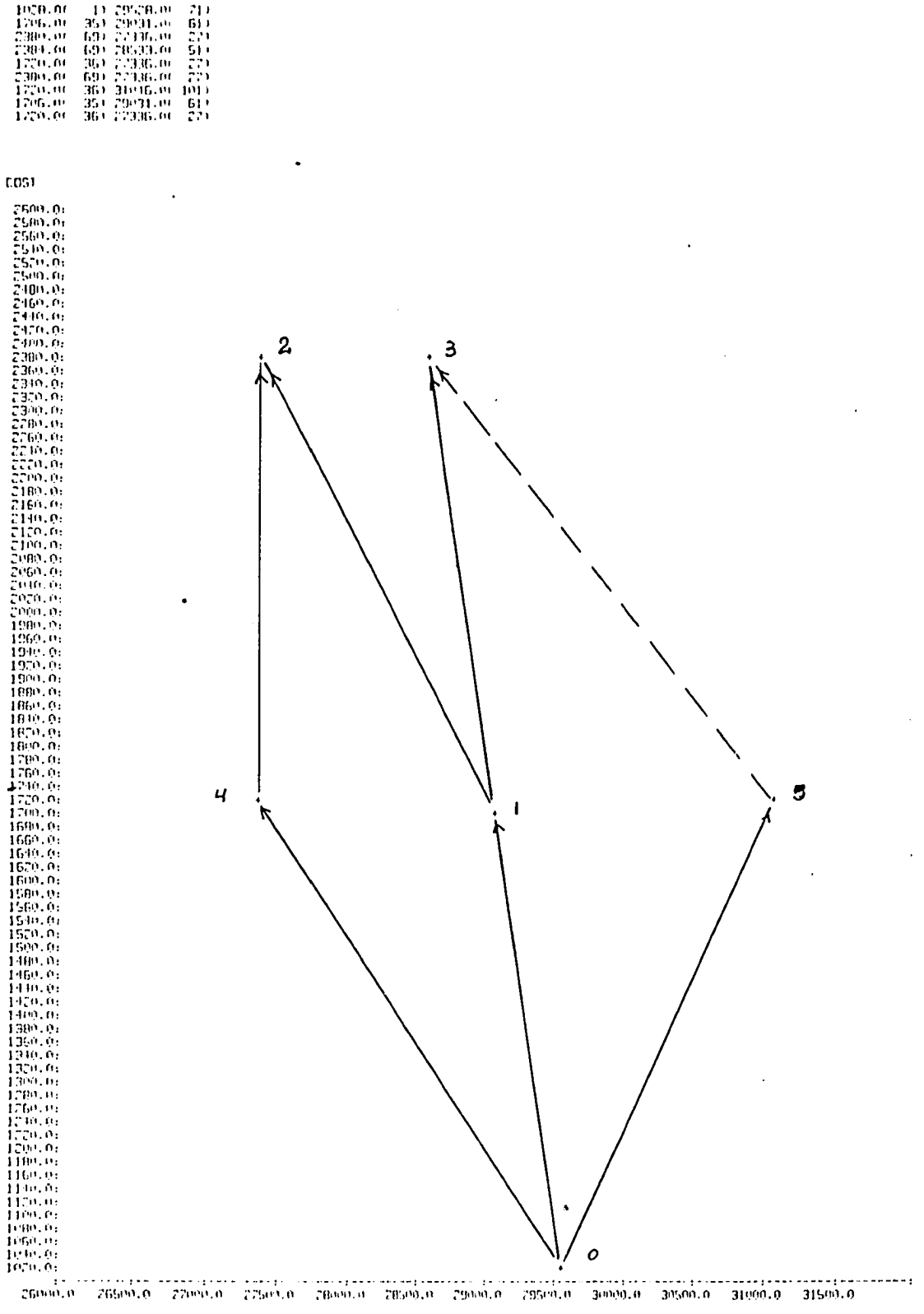


Figure 51. Conveyor – Bin Design Space

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

point 1, is accepted since the trade-off involved is less than the limit (100$/microsecond), but when reached from 4 it is rejected since there is no gain in speed. 2) Point 5, derived from point 0, presents a loss in both cost and time with respect to its predecessor and therefore it is rejected. 3) If point 5 had been accepted, the system would have found that point 5 could be transformed into point 3 which has an acceptable trade-off between cost and speed; this is indicated by a dotted line in Fig. 51.

This example illustrates several characteristics of the design space and the transformations:

- Concurrency does not always increase speed (e.g.: transition from point 0 to point 5)
- Points can be reached via several paths but not all of them will be acceptable (e.g.: paths 0-4-2, 0-1-2)
- Rejection of a solution implies rejection of any solution derived therefrom (the rejection of point 5 prevents the system from finding the alternative route to point 3)

V.3 A MODIFIED CONVEYOR BIN CONTROLLER

This is a variation of the system described in example 2. The algorithm is the same, but the initial solution includes some amount of concurrency. Other modifications include a different allocation of the variables and arrays used. Thus, CHP is declared as an independent register and the bin count and conveyor arrays are declared as

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

```
VAR;ALLRULES;SPEED;TRADE=100.00;LONG;LOCAL;

00200 IN STMT=42;BLK=14;INT=18;REG=1;GRP=45;CSE=3
              ;CST=1028;TME=29528;DST=148161.41
    0 SP  (42,1487)  43  (222,26662)  0 (30,2539)   0  (0,0)          10

CK0 14
CK0 38

00201 IN STMT=46;BLK=18;INT=22;REG=1;GRP=75;CSE=1
              ;CST=1706;TME=29031;DST=146015.30
    0 SP  (42,1487)  58  (183,22938) 25 (39,3724)   0  (0,0)          10

CK0 32
CK1 45
CK1 46
CK0 47
CK1 42

00202 IN STMT=47;BLK=19;INT=23;REG=1;GRP=141;CSE=-1
              ;CST=2380;TME=27336;DST=137876.90
00202  OUT

00201  BACK
    1 SP  (42,1487)  58  (183,22938)  2 (30,2539)   0  (0,0)          10

CK0 32
CK0 47
CK1 42

00203 IN STMT=49;BLK=20;INT=24;REG=1;GRP=99;CSE=-1
              ;CST=2384;TME=28533;DST=143864.19
00203  OUT

00201  BACK
00201  OUT

00200  BACK
    1 SP  (84,2974)  41  (185,22951) 21 (67,6250)   0  (0,0)          10

CK0 14
CK0 33

00204 IN STMT=46;BLK=18;INT=22;REG=1;GRP=93;CSE=0
              ;CST=1720;TME=27336;DST=137546.88
    0 SP  (0,0)      15  (35,3698)   0  (30,2539)   0  (0,0)          10

CK0 36
CK1 45
CK1 46
CK0 47
CK1 42

****REJECT  STMT=47;BLK=19;INT=23;REG=1
              ;CST=2380;TME=27336;DST=137876.90
00204  OUT

00200  BACK
    2 SP  (42,1487)  41  (185,22951)  2 (30,2539)   0  (0,0)          10

CK0 14
CK0 33

****REJECT  STMT=46;BLK=17;INT=21;REG=1
              ;CST=1720;TME=31046;DST=156097.81
    3 SP  (0,0)      13  (37,3711)   0  (30,2539)   0  (0,0)          10

CK0 33
CK0 38

 LOOP STMT=46;BLK=18;INT=22;REG=1
              ;CST=1706;TME=29031;DST=146015.30
00200  OUT

BEST NODE=00204
              STMT=46;BLK=18;INT=22;REG=1;GRP=93;CSE=0
              ;CST=1720;TME=27336;DST=137546.88
```

Figure 52. Conveyor–Bin Design Space Exploration – Trace

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

independent arrays.

```
L1 := ( ADVANCE.CONVEYOR.FLAG ⇒
        CHP ← CHP - 1 ;next
        (CHP = 0 ⇒ CHP ← 63) ;next
        CONVEYOR[CHP] ← ITEM.NUMBER; T ← CHP; I ← 1 ;next
        L2 := (BIN ← I ;next
                (BIN = CONVEYOR[T] ⇒
                        (BIN.COUNT[I] = 0 ⇒ FULL.BIN.FLAG ← 1 ;next
                                BIN.COUNT[0] ← BIN.COUNT[0] + 1 ;next
                                (BIN.COUNT[0] = 30 ⇒ OVER.30.ALARM ← 1) );
                        (BIN.COUNT[I] < 0 ⇒ EJECT.TO.BIN.SIGNAL ← 1);
                        (BIN.COUNT[I] > 0 ⇒ BIN.COUNT[I] ← BIN.COUNT[I] - 1 ;next
                                EJECT.TO.BIN.SIGNAL ← 1) ) ;next
                T ← T + 1; I ← I + 1 ;next
                (T > 127 ⇒ T ← 65) ;next
                (I = 64 ⇒ L2, L1) )
```

The extra concurrency appears in lines 4 and 13 of the ISP description:

```
CONVEYOR[CHP] ← ITEM.NUMBER; T ← CHP; I ← 1 ;next
.....
        T ← T + 1; I ← I + 1 ;next
```

Figure 53 shows the design space exploration, starting from this initial solution, and using as goal the most economic alternative. As could be expected, the solution is the one with the least concurrency, i.e. the initial solution of example 2. The difference in time and cost is due to the different type of variables used.

This example presents a large number of design space points, mainly because of the multiple cases in which rule PS can be applied. Although this example is in a sense

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

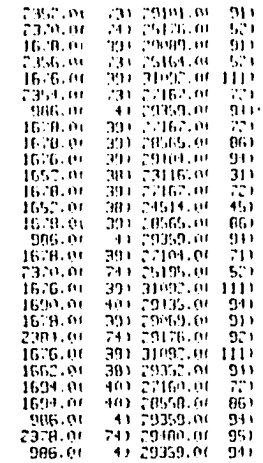
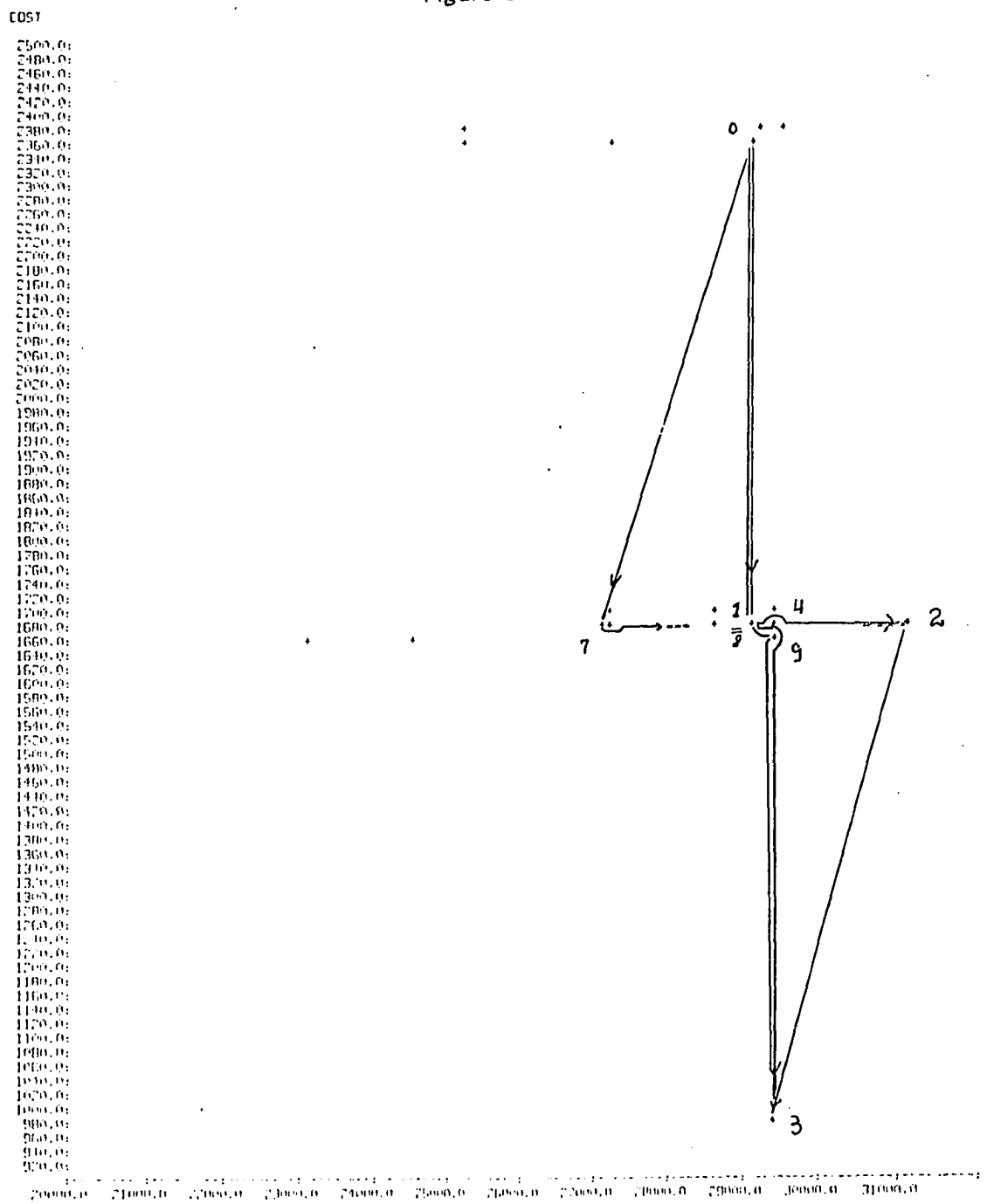


Figure 53. Modified Conveyor – Bin Design Space



Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

artificial (it is impractical to use multiple buses to perform in parallel simple operations like T←T+1 and I←I+1) it can be used to describe some characteristics of the RTM design space.

The first detail that can be observed in Fig. 53 is the regularity of the grouping of points along bands of similar cost, separated by cost gaps of constant size. The gap size is given by the cost of the RTM buses (together with one DMgpa and one set of registers). The first band consists of 6 points concentrated between $2300 and $2400; it represents the set of 3-bus systems. A second band, of 12 points, is concentrated between $1600 and $1700; it corresponds to the 2-bus systems.Finally, we have a 1-point band corresponding to the sequential solution i.e. the 1-bus system.

A second observation that can be made is that there are two 2-bus systems that are faster than any 3-bus system. This is explained by the fact that a 3-bus system will assign different buses for the computations involving T and I and, since the main computation uses both of them for array accessing, they must be copied onto the main bus for each execution of the inner loop.

The third fact that can be observed is the multiplicity of paths leading to the goal. The paths are shown as vectors between points in the paths. The reader could have guessed that all the vectors represent parallel to serial transformations (rule PS).

The overall lesson that can be learned from this example is that in RTM systems concurrency is always expensive and that it does not always imply a faster execution.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The same example was processed a second time, using the same goal (cost reduction) but enabling only the PS transformations (all other rules were ignored). The design space, shown in Fig. 54 presents, as could be expected, a dramatic reduction in the number of points explored before reaching the goal. Thus it can be concluded that to obtain the least expensive RTM system enabling only the PS rule is a viable technique. It can not be concluded, however, that to obtain gains in speed we should disable rule PS, as the example shows (in order to get to the 2-bus region we must eliminate one of the buses by applying PS).

V.4 TIMING MEASURES

The examples described above were processed with the timing routines enabled. This section present some of the timing measures as well as some comments about the ratio of groups to blocks for the examples.

The following table shows the timing measures for the examples. The times are given in seconds and the figures in parenthesis indicate the percentage of a measure over the total time required to process an example. The ACTUAL TIME figures represent the time required for a run, as measured by the system. The GROUP and CASES figures represent the total time required to create the *group* and *case* tables respectively. These tables, according to the strategic pruning, were created only for those points in the design space that were worth exploring i.e. that presented some improvement over its predecessor. The PROJECTED TIME figures indicate the time that

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

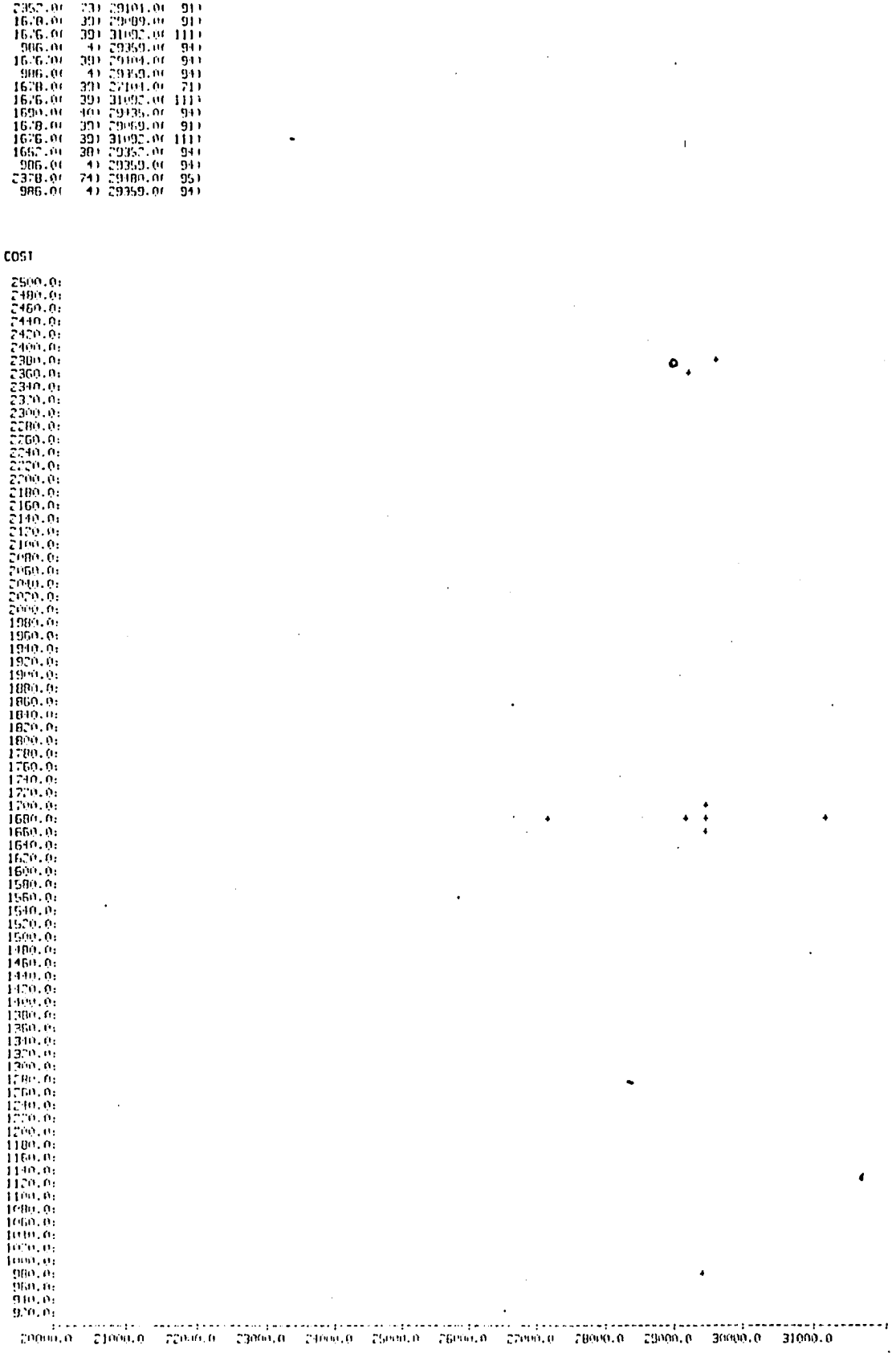


Figure 54. Modified Conveyor – Bin Design Space – Rule PS Only

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

would have been required had the strategic pruning been applied after the complete generation (i.e. including the groups and cases) of points that were rejected. The number of such points is given as EXTRA POINTS.

		ACTUAL TIME			PROJECTED TIME		
EXAMPLE	TOTAL	GROUP	CASES	TOTAL	GROUP	CASES	EXTRA
1	11.7	1.2 (10%)	3.2 (28%)	12.7	1.5 (12%)	4.0 (31%)	1
2	98.6	7.6 (8%)	76.4* (78%)	141.2	12.1 (12%)	114.5 (80%)	3
3	98.4	10.9 (11%)	39.0 (40%)	176.4	28.9 (16%)	99.0 (55%)	17
3PS	54.1	7.4 (14%)	22.6 (41%)	64.3	10.1 (16%)	30.1 (47%)	3

* This example presents an anomaly in the time required to compute the *cases*. This is due to one point in the design space that presents a very large number of groups, as will be shown later in the section.

Figure 55 shows a plot of the number of blocks versus the number of groups for those points in the design spaces that were explored (i.e. those points whose group and case tables were computed). The figure indicates that the number of groups tends to grow linearly with the number of blocks. The average ratios #group/#block for the examples are: 3.3 (example 1), 5.0 (example 2), 3.9 (example 3). The high ratio for example 2 is influenced by the high number of groups (141) that were found in one of the design space points (point 2). Although this increase in the number of groups with the number of blocks is intuitive, the number of cases derived from the groups bears no relationship to the number of groups, as the following table shows:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

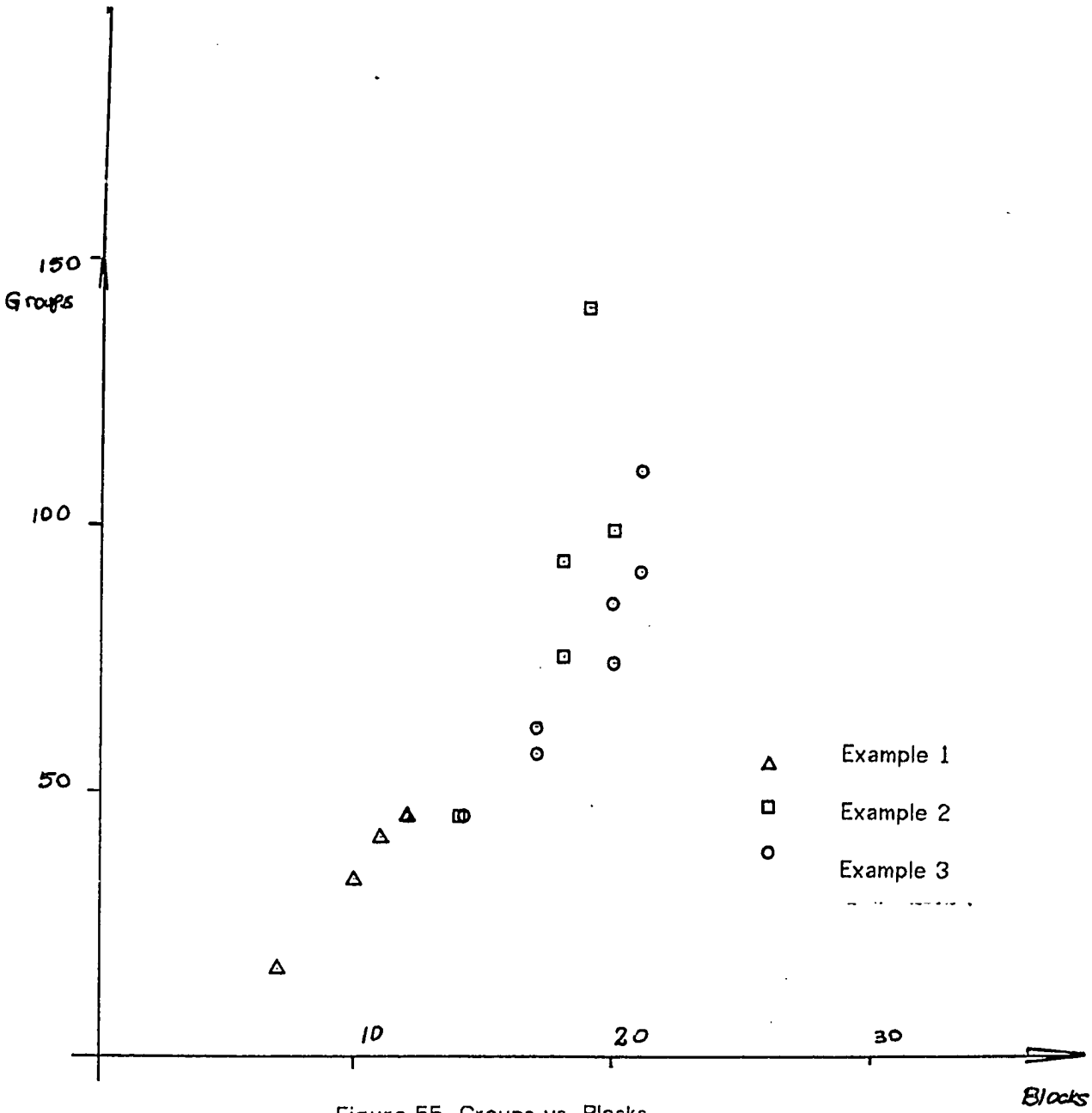


Figure 55. Groups vs. Blocks

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

EXAMPLE	POINT	BLOCKS	GROUPS	CASES	COMMENTS
1	0	7	17	2	Original solution
	1	11	41	0	Best solution
	2	10	33	3	
	3	12	45	0	Same as point 1
2	0	14	45	4	Initial solution
	1	18	75	2	
	2	19	141	0	
	3	20	99	0	
	4	18	93	1	Best solution
3	0	21	91	14	Initial solution
	1	20	74	11	
	2	17	57	8	
	3	14	45	2	Best solution
	4	17	62	5	
	5	21	110	3	
	6	20	85	3	
	7	20	74	10	
	8	20	74	10	
	9	17	62	4	

These figures tend to indicate that the algorithm used to partition the graph in groups is not very efficient. How to obtain graph partitions (groups) that are both small in numbers and rich in cases is a problem that requires further study before we attack larger problems.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

VI. CONCLUSIONS

The programs implemented as the EXPL system were designed to illustrate the principles of a computer aided design method based on the graph representation of alternative designs, all of which implement the same basic behavior or algorithm specified by the designer. By using a simple set of parameters, cost and time, to quantify the alternative designs it was possible to implement a design space search for better alternatives to a given initial solution, obtained by straightforward compilation of an ISP program.

During the implementation and testing of EXPL several ideas were tried and accepted or discarded, as they proved more or less valuable for our purposes. Section VI.1 deals with some features of the particular implementation of EXPL written by the author. Sections VI.2 and VI.3 deal with the methodology for register transfer level design, its extensions and shortcomings. Section VI.4 is a summary of the conclusions.

VI.1 – THE IMPLEMENTATION

A point must be made that the effort involved in the implementation would not have been successful without the use of a high level language. The implementation of EXPL (not counting the ISP compiler) took a single programmer (the author) 7 months to complete, from the initial conception to the completion of the project. EXPL is written

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

in BLISS [Wulf71] and consists of 15 software modules. The object code amounts to approximately 20,000 words (instructions) on a PDP-10. Two debugging aids proved to be very helpful during the implementation, DDT [DEC67] and SIX12 [John73b]. In fact, during the process of implementation and testing they were routinely loaded as part of the system, and appropriate commands were added to EXPL to allow the programmer to work at different levels of abstraction, namely, EXPL itself via the command language, the BLISS run time environment via SIX12, and the machine code via DDT. A timing package, TIMER [Newc73], was used during the development of EXPL to identify those sections of code where additional effort in the coding would increase the efficiency of the system. Again, commands were defined in EXPL to enable or disable the timing facility at will, thus reducing the overhead involved in a continuous measurement.

The internal data structures were organized as tables of fixed sizes. The decision was based on the availability of enough core on the PDP-10 and saved the author the effort of building a space allocation/garbage collection package. In hindsight, a list oriented organization could have been more efficient. The reconnection of the graphs involved modifications of values in the *statement* table that rendered the rest of the tables invalid, thus requiring their complete regeneration. A list oriented structure would have helped this process by simply redefining the appropriate pointers (some regeneration would still be necessary, but to a lesser extent). The table oriented organization had the additional inconvenience that the identification of the elements on a given table (the index) would change between versions (points in the design space).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Thus, the interactive user is sometimes forced to obtain new listings (via the PRINT and DISPLAY commands) to observe the alternative designs as they are generated.

Set structures and operations are used extensively in EXPL. The sets are represented as PDP-10 words, where each bit can be turned *on* or *off* to indicate the presence or absence of an element in a set. This representation allows the use of the full set of PDP-10 logical operations to perform operations like set intersection, set union, set complement, set coverage, etc. in an efficient manner. A limitation in the current system is that sets are limited to 36 elements (a PDP-10 word has 36 bits). This could be improved by recoding a few *routines* and *macros* used to implement the set operations.

The organization of the system, the data structures used, and the optimization features of BLISS cooperated to make EXPL a fast system, albeit a large one (20K of instructions, 8K of data). The system when used in an interactive manner provides a response time measured in seconds, even for expensive commands like the creation of the *case* table.

The use of a cleanly defined intermediate language, a set of calls to BLISS routines, to provide the real input to EXPL (the graph corresponding to the initial solution) had two effects. First of all, it allowed the testing of EXPL independently of the ISP compiler, which was in fact implemented after EXPL was completed. This avoided the complexity of debugging a larger system. The second effect is that the system is not limited to the use of initial solutions (the behavioral specifications) describable in ISP.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

The last two features, a fast response time and a two level capability for describing the behavior of a digital system, together with a useful command language contribute to make EXPL a research tool that is both efficient and easy to use.

VI.2 – A METHODOLOGY FOR RT LEVEL DESIGN

Chapter IV presented the characteristics of the heuristic design space search and the algorithms used to reduce the exploration. The implementation of the particular space search used in EXPL was dictated by two considerations. The first is the difficulty in describing the goal with any precision i.e. by a specific cost/time pair of values, as opposed to vaguer statements like "fastest" or "cheapest". A specific cost/time point given as goal would imply two things: either the designer is just expressing a wish or the designer knows a solution with these characteristics. In the latter case, the design space exploration becomes unneccessary. In the former case, it is not known whether or not there exists such a point, and unless an exhaustive search is performed (more about this, later) the search routines will report a failure if the point is not reached. This could be improved by specifying a region instead of a single point, but the basic problem still remains. The second consideration that went into the implementation of the space search was the impracticality of exhaustive searches, even for problems like those in the examples. For instance, the conveyor–bin used in section V.3 has, at every point, an average branching factor (number of alternative successors) of 4.1 (from a minimum of 2 – the final solution, to a maximun of 14 at node 0 – the initial solution).

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Since that for a given point in the design space there may exist a multiplicity of alternative points derived from it, and that there may be several paths (sequences of transformations) leading to the best design, it is desirable to be able to find short paths to the goal. The purpose of the heuristic search is to make more efficient use of the resources of the system. Thus, short paths tend to increase the efficiency of the design process itself. The strategies used in EXPL are oriented towards this goal, for instance, transformations that yield very small improvements are rejected by the tactical pruning routine. A major improvement can be obtained, in EXPL, by building into the system a larger knowledge of the design space. This could be used, for instance, to allow temporary violations of design constraints if, by going to a point that is worse that its predecessor, the system finds a shorter path to the goal (or even to a better solution). EXPL is deficient in this respect, for the strategic pruning is based on a local testing of improvement and the rejection of a point in the design space masks out points derived therefrom. An improvement would be achieved by increasing the look ahead capability (1-deep in the present system) to, say, 2 or 3 points (transformations) removed, before a path is rejected.

The RTM evaluation routines perform an important role in the system. The criteria on which a point is accepted or rejected is based on the evaluation of the RTM system that would be used to implement the design alternative under consideration. The RTM evaluation tries to be fair, to the same extent, to all points in the design space. The evaluation is based on the cost and time requirements of a "naive" RTM implementation, that is, no RTM dependent optimizations are performed on the structure. For each data

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

and control operation a template of RTM components is used, much like a macro expansion in a programming language. This expansion takes into account the specific operation and the types of variables used (e.g. memories, registers, flags). The only optimization allowed is the use of the DMgpa capability to perform a shift right (/2) operation in the same step as any other operation, thus saving both in cost (the Kevokes) and time. A useful expansion to EXPL would be the encoding of RTM optimizations that would take into account the use of the data part resources, for instance, DMgpa's and their local memory capability. Another type of optimization, along the same lines, would be the reallocation of variables to different memory modules than those explicitly declared by the designer as part of the behavioral description.

EXPL, by using the current RTM evaluation routines, assumes that a more clever implementation of the RTM data part would be reflected in a proportional reduction in cost or time in all the design space points, in other words, it assumes that the real design space is similar to the one explored, with a certain shrinkage of the scales. The validity of this assumption is being tested as part of ongoing PhD research at CMU by Satish Rege [Rege73]. The approach used by Rege, not restricted to the use of RTM's, is based on the exploration of the design space of the data flow structures that are implied by a given control structure. The research reported here and Rege's can be put in terms of the work by Dennis and Patil [Denn70], in the following form: EXPL explores the design space of the different realizations of a (control) *precedence graph* corresponding to a *computation schema*. Rege's work explores the design space of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

different realizations of a *data flow graph* corresponding to a *computation schema*. A combination of the current EXPL capabilities with Rege's results would result in a more powerful design system, where each point in the design space is characterized by a given cost/time evaluation resulting from a given control structure and an optimal data flow structure.

VI.3 – EXTENSIONS

Chapter II presented several PMS configurations that can be used to characterize building blocks from which digital systems can be constructed. The different PMS types dictate the conditions for topological reorganization of the graph. Thus, in principle, EXPL can be extended to handle non-RTM systems by replacing the current test routines (and, of course, the evaluation routines) by others that reflect the conditions of the PMS type of the new building blocks.

A possible application is the use of a design system capable of designing micro-programmed machines. Two types of such applications can be observed. First, the design system can be used to optimize micro-programs for an existing machine. This application would aim to find the best combination of micro-commands that can be executed concurrently, say, in a horizontal micro-programmed machine. It would take into account the number and types of micro-commands that can coexist in the same control word. Costs would be given by the length of a microprogram, times would be computed from the requirements of the individual micro-commands. The second

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

application in micro-programming would be the specification of the micro-controller itself, the width of the control words, and the concurrency of the different functional units.

Although an application of this type has an immediate appeal due to the current vogue of micro-programming applications, it is yet to be seen whether a general design system can compete with human designers without extensive tuning of the design programs for each specific application.

Flow analysis and graph transformation techniques, similar to those implemented in EXPL, could be used as design aids for implementing systems out of Computer Module (CM) networks [Bell73]. Basically, a CM is a Processor-Memory pair with several I/O ports. There is no central shared memory as in normal multiprocessor networks and the CMs communicate via a virtual memory mechanism that maps (via the I/O ports) virtual addresses into a large physical address space, distributed among the CMs in the network.

Since each CM has its own processing unit, with higher capabilities (i.e. PDP-8 like) than a RT level module, trade-offs will have to be defined also at a higher level. The transformations would aim to partition and map the graph representation of behavior into a CM network. Transformations would be applied to reduce the cost for the time requirements by using more or less CMs, and by reorganization of the network. The trade-offs would include not only the cost of the modules and the time required by the computation assigned to them, but also the cost of sharing variable via

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

the virtual memory mechanism. Since CMs are not intended to be used in conjunction with large switches like C.mmp [Wulf72], transfer of variables among modules implies a possible routing through several intermediate modules with a consequent degradation in performance. The partition and assignment of computations to modules must take into account the physical location of the variables. The CMs themselves are somewhat oblivious to the fact that variables may reside far apart from the module that uses them. This is a function of the virtual memory mechanism.

From a practical point of view, a system like EXPL can be a valuable design tool, not only through the application of serial/parallel trade-offs as stressed throughout the thesis, but also, by extending the system to perform other tasks: code optimization (e.g. code motion, code elimination, register allocation), design checking (e.g. detecting improper use of variables), and testing (e.g. building a simulator inside the system).

VI.4 - SUMMARY

The analogy drawn between the programming and RT levels served a dual purpose. First of all, it allows us to represent the behavior of an RT system as a program flowchart without having to stress the fact that algorithms are hardwired instead of interpreted by an instruction set processor. This was reflected in the use of a graph model whose elements have been present in one form or another in programming languages. Thus the model itself is not original, its use to define structural

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

transformations on a hardware system is, however, new. The second advantage is that we can use the experience in code optimization developed over the years, with the added advantage that we are not constrained to a fixed machine organization.

EXPL was implemented as a research tool and the main emphasis was given to provide the user with a set of capabilities by which different design strategies can be tested. EXPL can be used as an interactive system using the command language or as a closed system by providing an appropriate WALK routine to perform the design space exploration. These capabilities together with the user specification of goals present, however, only one half of the picture. A more useful system must allow the designer to specify the building blocks from which the final products can be built. The specifications of the RTM components (costs, speeds, and interconnection rules) are provided to EXPL by means of software tables and routines. A whole area of research concerning the description of hardware building blocks is wide open. Built-in descriptions, as in EXPL, can only be a temporary solution. They tend to burden the designer with peculiarities of the machine and the language used, as well as the particular way the main body of the system is implemented.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

REFERENCES

[Alle70] Allen, F.E.: "Control Flow Analysis". Sigplan Proceedings, Vol. 5, No. 7, July 1970, pp. 1-19.

[Barb73] Barbacci, M.R.: "A Comparison of Register Transfer Languages for Describing Computers and Digital Systems". Department of Computer Science, Carnegie-Mellon University, March 1973. PB-221591.

[Bell71a] Bell, C.G., A. Newell: "Computer Structures: Readings and Examples". McGraw-Hill Book Company, New York, 1971.

[Bell71b] -----, A.G. Jordan, and J.F. Traub: "Register Transfer (RT) level: Components, Representations, and Design Techniques". Proposal from the Departments of Electrical Engineering and Computer Science, Carnegie-Mellon University, to the National Science Foundation (Computer Systems Design Program), CMU proposal No. 1582, October 1971.

[Bell72a] -----, J.L. Eggert, J. Grason, and P. Williams: "The Description and Use of Register Transfer Modules (RTM's)". IEEE-TC, Vol. C-21, No. 5, May 1972, pp. 495:500.

[Bell72b] -----, J. Grason, and A. Newell: "Designing Computers and Digital Systems". Digital Press, Digital Equipment Corporation, 1972.

[Bell73] -----, R.C. Chen, S.H. Fuller, J. Grason, S. Rege, and D.P. Siewiorek: "The Architecture and Applications of Computer Modules: A Set of Components for Digital Design". IEEE Computer Society International Conference, COMPCON73, March 1973.

[Bern66] Bernstein, A.J.: "Analysis of Programs for Parallel Processing". IEEE-TEC, Vol. EC-15, No. 5, October 1966, pp. 757:763.

[Breu72] Breuer, M.A.: "Recent Developments in the Automated Design and Analysis of Digital Systems". Proceedings of the IEEE, Vol. 60, No. 1, January 1972, pp. 12-27.

[Clar67] Clark, W.A.: "Macromodular Computer Systems" (an introduction to a set of 6 papers on the subject). AFIPS Conference Proceedings, Vol. 30, SJCC 1967, pp. 335-402.

[Darr69] Darringer, J.A.: "The Description, Simulation, and Automatic Implementation of Digital Computer Processors". PhD thesis, Department of Electrical Engineering, Carnegie-Mellon University, May 1969.

[DEC71] Digital Equipment Corporation: "PDP-16 Computer Design Handbook". 1971.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[DEC67] -----: "PDP-10 Reference Handbook". 1967.

[Denn70] Dennis, J.B., S.S. Patil: "Computation Structures". Notes for subject 6.232, Department of Electrical Engineering, Massachusetts Institute of Technology, 1970.

[Fish70] Fisher, D.A.: "Control Structures for Programming Languages". PhD Thesis, Department of Computer Science, Carnegie-Mellon University, May 1970.

[Frie10] Friedman, T.D., S. Yang: "Methods Used in An Automatic Logic Design Generator (ALERT)". IEEE-TC, Vol. C-18, No. 7, July 1969, pp. 593:614.

[Full73] Fuller, S.H., D.P. Siewiorek: "Some Observations on Semiconductor Technology and the Architecture of Large Digital Systems". Report on the Workshop on the Architecture and Applications of Digital Modules, Department of Computer Science, Carnegie-Mellon University, June 7-8, 1973.

[Huen73] Huen, W.H.: "Representation of RT Control". PhD thesis, Department of Computer Science, Carnegie-Mellon University, December 1973.

[John73a] Johnsson, R.K.: "A Survey of Register Allocation". Department of Computer Science, Carnegie-Mellon University, May 1973.

[John73b] -----, C.B. Weinstock, W.A. Wulf, and T. Lane: "SIX12 Reference Manual". Department of Computer Science, Carnegie-Mellon University, 1973.

[Ople65] Opler, A.:"Procedure Oriented Language Statements to Facilitate Parallel Processing". CACM Vol. 8, No. 5, May 1965, pp. 306:307.

[Nils71] Nilsson, N.J.: "Problem-solving Methods in Artificial Intelligence". McGraw-Hill Book Company, New York, 1971.

[Newc73] Newcomer, J.M.: "BLISS Timer Package - User's Manual". Department of Computer Science, Carnegie-Mellon University, March 1973.

[Pota69] Potash, H.: "A Digital Control Design System". PhD thesis, Report No. 69-21, School of Engineering, University of California, Los Angeles, May 1969.

[Rege73] Rege, S.: Private communication.

[Rosi69] Rosin, F.R.: "Contemporary Concepts of Microprogramming and Emulation". Computing Surveys, ACM, Vol. 1, No. 4, December 1969, pp. 197:212.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

[Rust72] Rustin, R.(ed): "Design and Optimization of Compilers". Courant Computer Science Symposium 5. Prentice-Hall Inc. 1972.

[Wars62] Warshall, S.: "A Theorem on Boolean Matrices". JACM, Vol. 9, No. 1, Jan. 1962, pp. 11-12.

[Wulf71] Wulf, W.A., D.B. Russell, and A.N. Haberman: "BLISS: A Language For Systems Programming". CACM, Vol. 14, No. 12, December 1971, pp. 780:790.

[Wulf72] -----, C.G. Bell: "C.mmp - A Multi-Mini-Processor". AFIPS Conference Proceedings, Vol. 41, Part II, FJCC 1972, pp. 765:777.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX I - REGISTER TRANSFER MODULES

CONTROL MODULES

Evoke\Ke module.- The Ke module (Fig. A1.1) is the basic control module. Its function is to evoke a single register transfer operation in the data part (D, M, and T modules), with or without bus intervenlion. The operation sequence of a Ke module is as follows:

1) The Ke module receives a signal, called *activate*, at its input link. This signal activates the module.

2) The Ke sends a signal, called *evoke operation*, to the data part of the system.

3) When the operation is completed, the bus control module sends a *done* signal back to the Ke module.

4) The Ke passes control on to the next control module by generating the *activate next* signal.

2-way branch\Kb2.- The Kb2 module (Fig. A1.2) provides for the branching of control flow based on the condition of a boolean variable. Each time a Kb2 is activated (activate signal), it initiates either one of the subsequent control paths (activate next signals), depending on whether the boolean input is true (1) or false (0).

8-way branch\Kb8.- The Kb8 module (Fig. A1.3) is similar to the Kb2, but it has three boolean inputs and eight possible activate-next control outputs. Each one of these corresponds to one of the eight possible combinations of the three boolean inputs.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

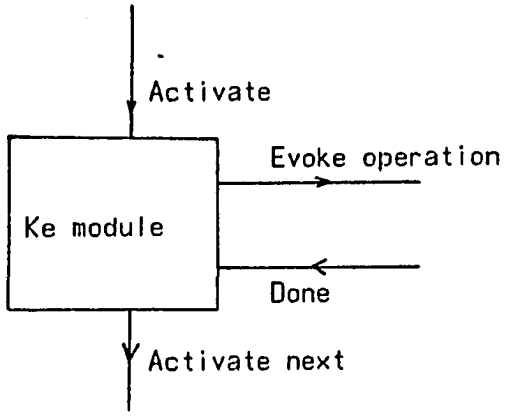


Figure A1.1 - Ke module

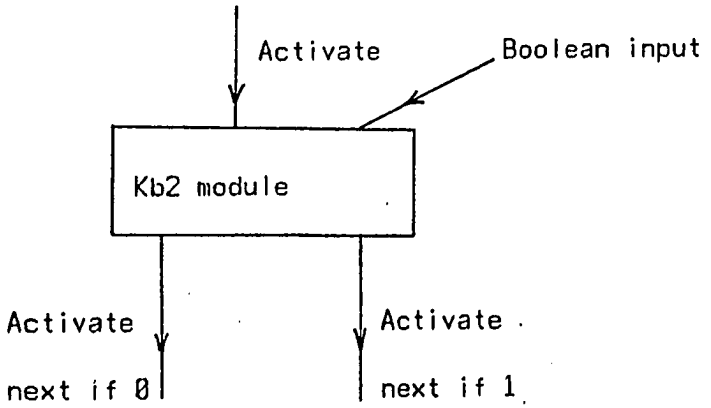


Figure A1.2 - Kb2 module

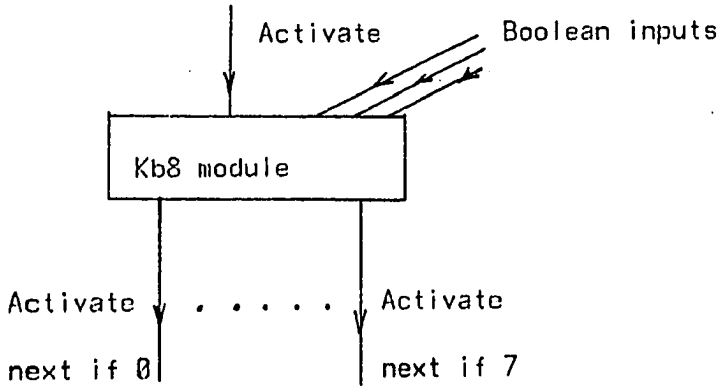


Figure A1.3 - Kb8 module

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

Diverge\Kdiv.– The Kdiv module (Fig. A1.4) is an implicit module. It consists of a fan–out of a wire carrying an activate signal. It is used whenever two or more concurrent control paths are to be activated.

Serial merge\Ksm.– The Ksm module (Fig. A1.5) allows the merging of several control paths, only one of which should be active at a time. If any one of the activate signals is present, it will produce the activate next output.

Parallel merge\Kpm.– The Kpm (Fig. A1.6) allows the merging of concurrent control paths. When all activate inputs have been evoked the Kpm generates the activate next signal.

Bus sense and termination\Kbus.– This module (Fig. A1.7) is a centralized control module that governs the action of a bus. It has a Bus Sense Register\BSR which always contains the result of the last register transfer that took place via the bus. The functions provided by the Kbus include:

1) Monitors all register transfer operations via the bus sequencing signals and supplies the operation completion signal (done) to the K modules.

2) Provides for manual control of an RTM system via a set of switches.

3)Provides a reset signal which initializes the modules when power is turned on.

4) Allows sense lights to be connected to the BSR for monitoring purposses.

5) Provides a constant, 0, to be put on the bus.

6) Allows for transfer of data to the BSR.

7) Provides the following boolean outputs which are available after each control step:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

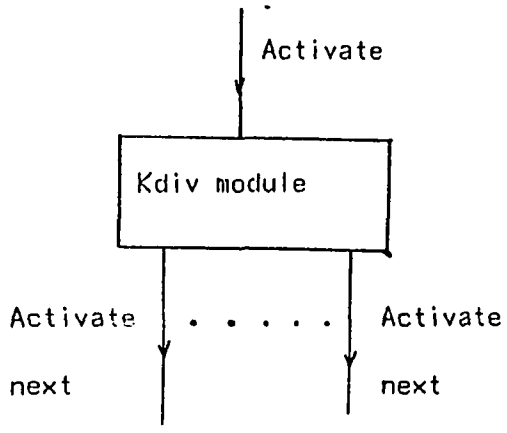


Figure A1.4 - Kdiv module

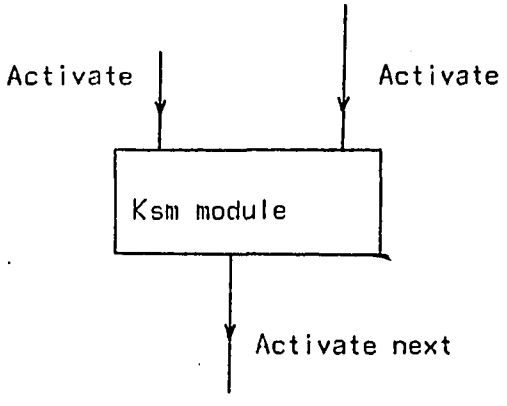


Figure A1.5 - Ksm module

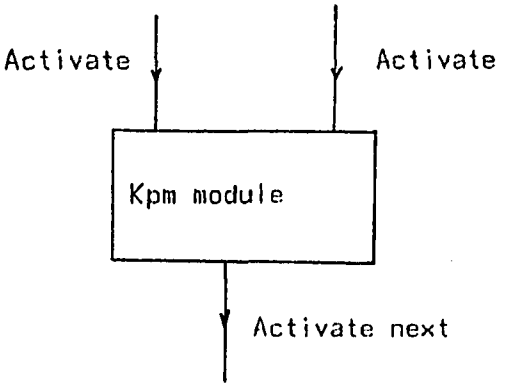


Figure A1.6 - Kpm module

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

BSR=0; BSR>0; BSR<0
all 16 bits of the BSR
overflow bit

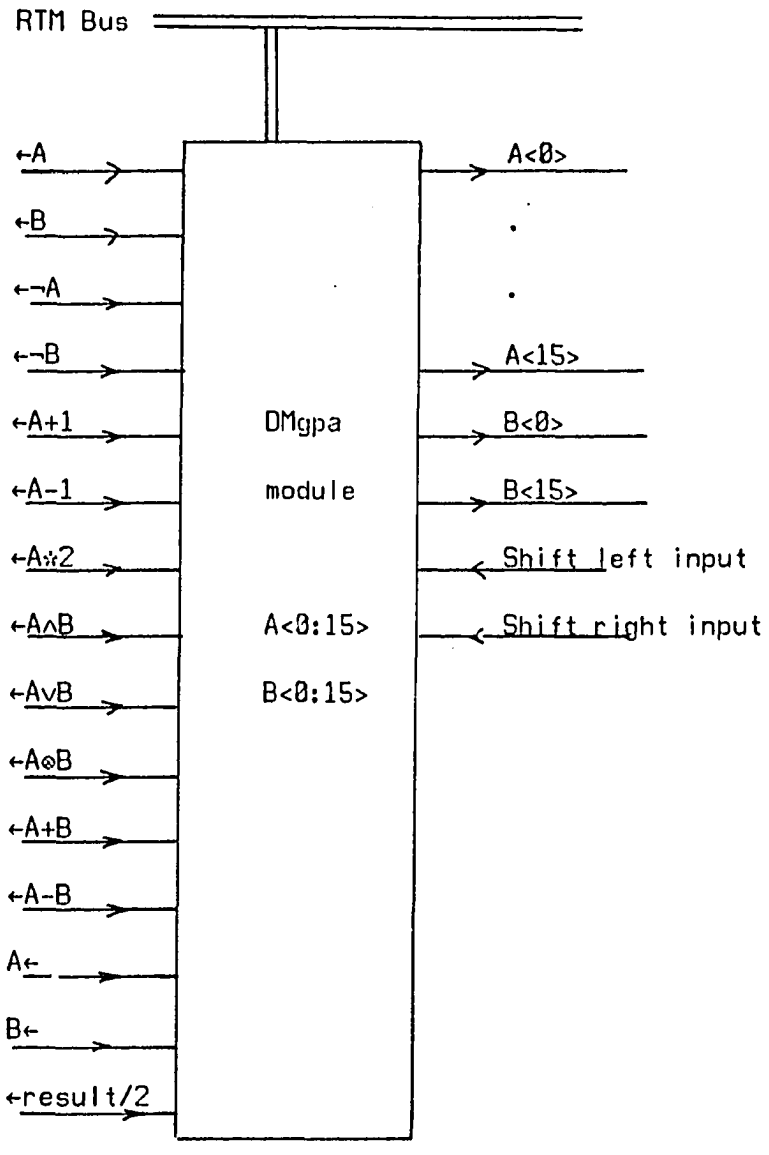


Figure A1.11 - DMgpa module

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

DATA MODULES

The basic units of information (data types) are the 16 bit, 2'complement integer, the 16 bit Boolean vector, and the single bit boolean variable. The RTM set provides several storage units capable of handling these basic data types.

Flag\DMflag.- This module (Fig. A1.8) consists of a 1 bit register, F, capable of storing a boolean variable. The operations allowed are:

F←0; F←1; F←¬F; F←data;

Array\Marray.- There are two versions: a 1024 and a 256, 16 bit words of random access, read/write memory. The operation is similar in both versions and consists of two steps. In the first step, a memory address register is loaded (MA←Bus). The second step invokes the desired data transfer operation (Marray[MA]←Bus or Bus←Marray[MA]).

Scratchpad\Mscp.- This memory module (Fig.A1.10) is an array of 16 registers. During operation a register is selected by an individual signal accompanied by a read or write command (Bus←Si or Si←Bus). Since it does not require a memory address decoding its operation is faster than that of the Marray modules.

Several other memory modules are available, with special capabilities, but those presented so far are sufficient for the time being.

General Purpose Arithmetic\DMgpa.- This is the main computational element of the RTM set (Fig. A1.11). The DMgpa carries out arithmetic and logical operations. It

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

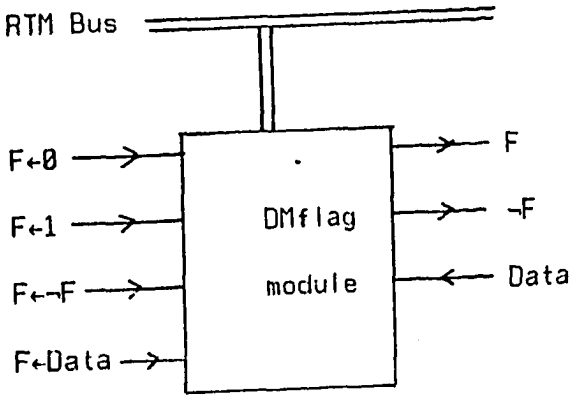


Figure A1.8 - DMflag module

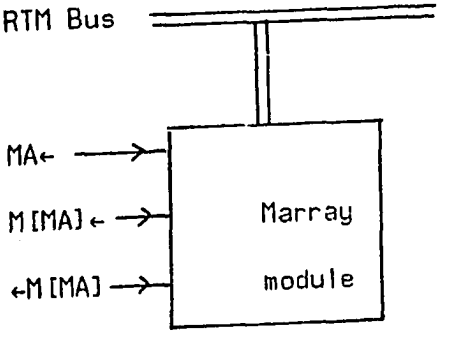


Figure A1.9 - Marray module

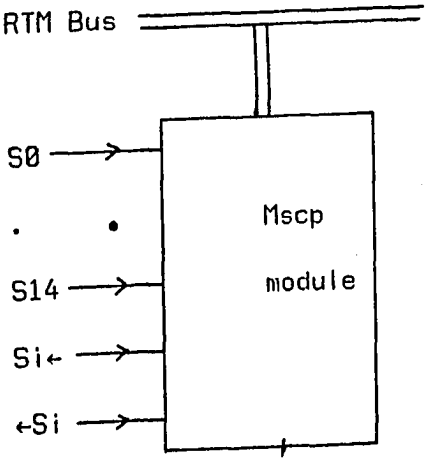


Figure A1.10 - Mscp Module

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

has two registers, A and B, used for temporary storage of operands and results. The DMgpa performs its operations under request of Ke modules. The operations are the following:

Bus ← A
Bus ← B
Bus ← ¬A
Bus ← ¬B
Bus ← A + 1
Bus ← A – 1
Bus ← A * 2
Bus ← A ∧ B
Bus ← A ∨ B
Bus ← A ⊕ B
Bus ← A + B
Bus ← A – B
A ← Bus
B ← Bus

A special operation can be invoked concurrently with any of the above:
Bus ← result/2 (the result of the other operation is shifted right).

Besides the control (evoke operations) inputs, there are several data lines that can be used in the DMgpa operation:

- End bit inputs for the shift operations (Bus ← A * 2 and Bus ← result/2): Shift left input, Shift Right Input.

- All 16 bits of register A and bits 15 (sign) and 0 of the B register are available as boolean outputs of the DMgpa.

- The overflow bit of the BSR (in the Kbus module) is set to the carry out bit in arithmetic operations or the bit shifted out in shift operations.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

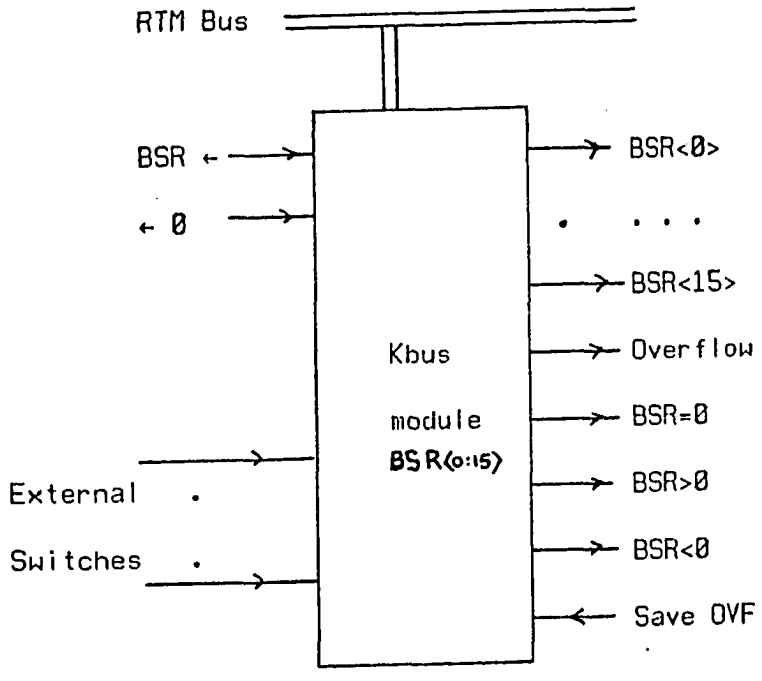


Figure A1.7 - Kbus module

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

RTM EVALUATION TABLE

CONTROL	COST($)	TIME(nsec) path delay
Kb2	7	40
Kb8	20	40
Kdiv	0	0
Ksm2	1	30
Ksm4	2	10
Kpm2	1	30
Kpm4	2	10
Kbsr	145	70
Kevoke	7	60

DATA (NSEC)	COST($)	READ TIME (NSEC) (bus ←)	WRITE TIME (←bus)
DMflag	13	400	300
DMgpa	315	415(1)	290
Mc4	10	220	---
Mc24	320	1000	---
Mscp	200	270	600
Marray(2)	500	2200	1600
Marray(3)	1500	2200	1600
Maddress	0(4)	---	270
Tgpi	100	190	280

(1) Includes any data operation
(2) 256 words
(3) 1024 words
(4) part of Marray

The time for any register operation is given by:

time = read time + write time + bus time (Kbsr) + k time (path delay)

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX 2 - BUS IDENTIFICATION EXAMPLE

Assume the graph shown in Fig. A2.1. The first step of the algorithm assigns colors to the basic blocks. The criteria is to find all the nodes that can be computed on a single bus. The following nodes are assigned colors as the first step: The entry block, the blocks headed by parallel merge operations, and the blocks that are immediate successors of diverge operations. The assignment is carried out, starting from these initially marked blocks, following the edges of the graphs, and assigning the same color to all the successor blocks of a marked block, unless a color has already been assigned to it. The following "colors" are identified (the blocks assigned to these colors):

C1 := {1,2,3,4}

C2 := {5,8,9}

C3 := {6,10}

C4 := {7}

C5 := {11}

C6 := {12}

Figure A2.2 shows a simplified version of the graph, in which the subgraphs are replaced by a single node, identified by its color.

The conflicting colors are identified, and the information stored in a conflict table. It contains one entry for each Kdiv or Kpm operator.

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

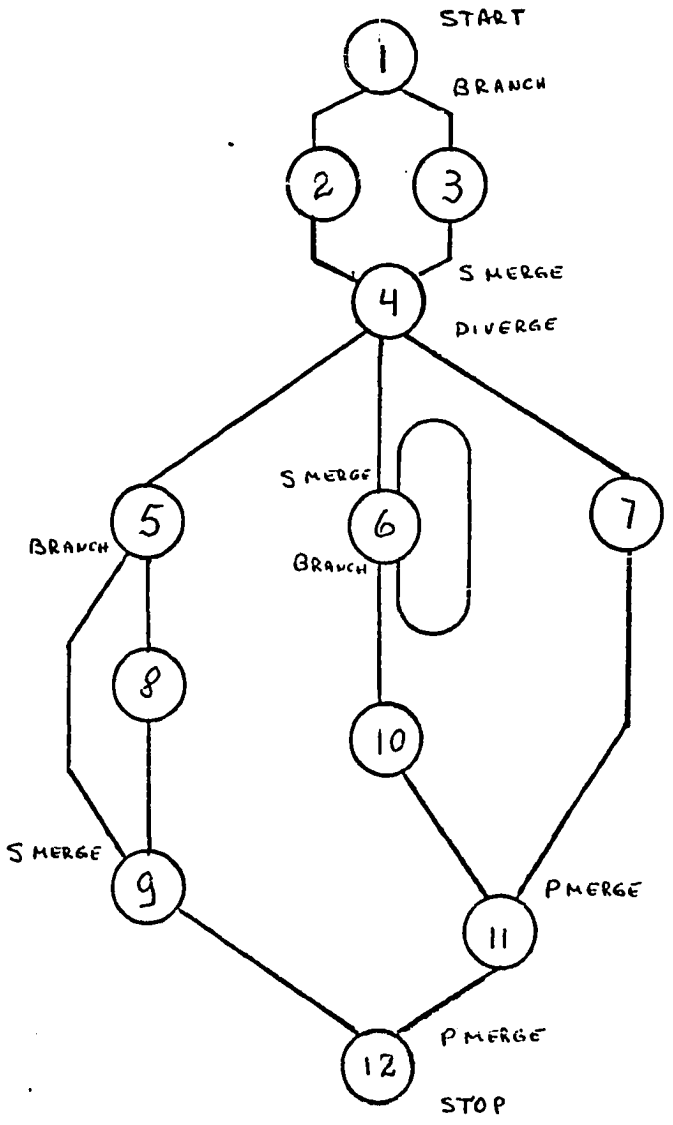


Figure A2.1 - Bus identification - Flowchart

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

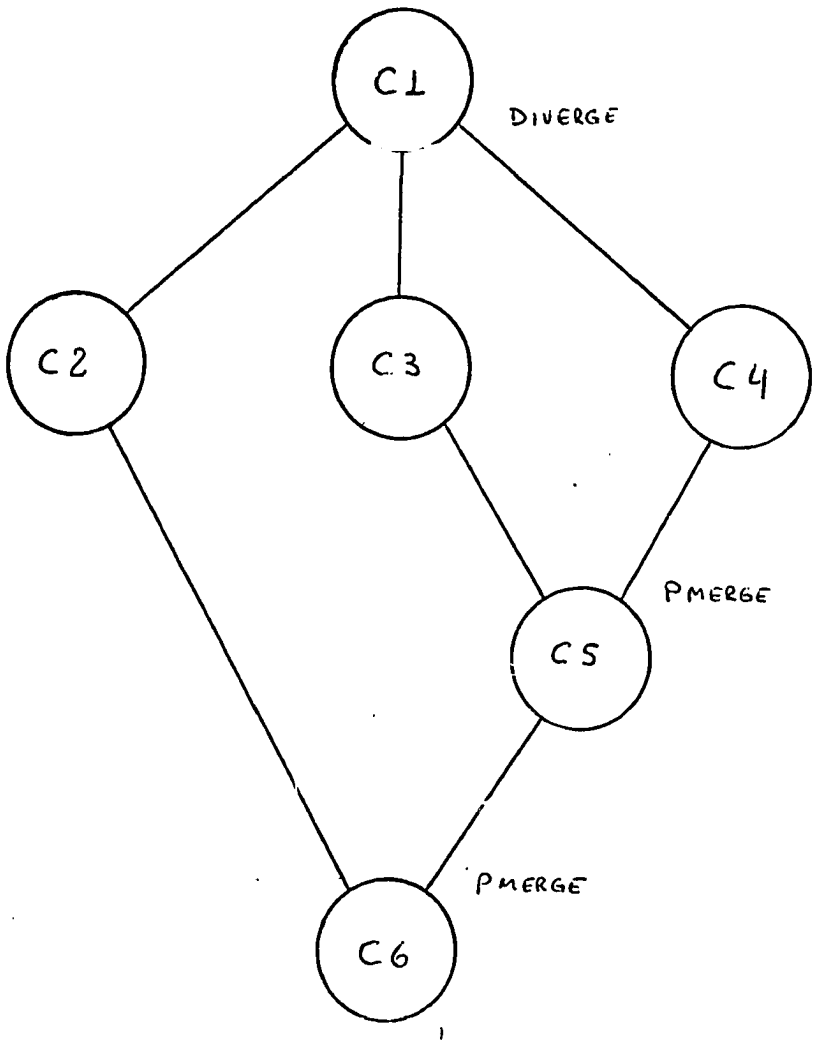


Figure A2.2 - Bus identification - Initial Colors

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

CONFLICT TABLE

1) {C2,C3,C4}
2) {C3,C4}
3) {C2,C5}

An initial alias table is defined. It contains an entry for each color.

ALIAS TABLE

1) C1
2) C2
3) C3
4) C4
5) C5
6) C6

From the alias table we can identify the following non-conflicting pairs of colors:

{C1,C2}
{C1,C3}
{C1,C4}
{C1,C5}
{C1,C6}
{C2,C6}
{C3,C5}
{C3,C6}
{C4,C5}
{C4,C6}
{C5,C6}

These non-conflicting pairs are obtained by listing all possible unordered pairs of entries from the alias table, eliminating those that appear in an entry of the conflict table. All these pairs are evaluated by counting the number of common variables between the two colors (buses). The variables are obtained from the blocks of the

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

original graph. The pair with the maximun evaluation factor, say, {C3,C6} is chosen. This is the pair of colors that we want to assign to the same bus to reduce the interbus transfer of variables. After identifying C3 and C6 as one and the same color, the conflict and alias tables are updated to reflect this situation:

CONFLICT TABLE

1) {C2,{C3,C6},C4}
2) {{C3,C6},C4}
3) {C2,C5}

ALIAS TABLE

1) C1
2) C2
3) {C3,C6}
4) C4
5) C5

The non conflicting pairs from the new alias table are found:

{C1,C2}
{C1,{C3,C6}}
{C1,C4} {C1,C5}
{{C3.C6},C5}
{C4,C5}

After evaluating this list of candidate pairs, we chose the one with the highest factor, say, {C1,C5}. The new conflict and alias tables are:

CONFLICT TABLE

1) {C2,{C3,C6},C4}
2) {{C3,C6},C4}
3) {C2,{C1,C5}}

ALIAS TABLE

1) {C1,C5}
2) C2
3) {C3,C6}
4) C4

The non-conflicting pairs are:

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

{{C1,C5},{C3,C6}}
{{C1,C5},C4}

Assume that the evaluation procedure identifies the pair {{C1,C5},C4} as the best candidate. the updated tables are:

CONFLICT TABLE

1) {C2,{C3,C6},{{C1,C5},C4}}
2) {{C3,C6},{{C1,C5},C4}}
3) {C2,{{C1,C5},C4}}

ALIAS TABLE

1) {{C1,C5},C4}
2) C2
3) {C3,C6}

Since no non-conflicting can be found, the procedure stops. The number of buses required is the number of entries in the alias table. The assignment of the buses is performed by marking the blocks of the initial graph with the alias table entry numbers of their colors. Thus:

Bus 1 := Colors C1,C2, and C5 := Blocks 1,2,3,4,7, and 11
Bus 2 := Color C2 := Blocks 5,8, and 9
Bus 3 := Colors C3 and C6 := Blocks 6,10, and 12

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

APPENDIX 3 – ISP SYNTAX

isp-definition ::= *process-definition*
process-definition ::= identifier := (*declarations action-list*)
declarations ::= DECLARE *declaration-list* ERALCED
declaration-list ::= *declaration* { ; declaration } *
declaration ::= *register-declaration* |
network-definition |
process-definition |
register-declaration ::= identifier *structure-declaration*
structure-declaration ::= { [*name-list*] | ∈ } < *name-list* > { ↑ number | ∈ }
name-list ::= *element-range* { , *element-range* } *
element-range ::= number : number |
number |
identifier
register-access ::= *single-register-access* { □ *single-register-access* } *
single-register-access ::= identifier { *structure-selector* | ∈ }
structure-selector ::= { [{ *arith-expression* | ∈ }] | ∈ }
{ < { *selector-range* | ∈ } > | ∈ }
selector-range ::= *bit* |
bit : *bit*
bit ::= number |
identifier
network-definition ::= *register-declaration* := *expression*
action ::= *register-access* ← *arith-expression* |
labelled-statement |
conditional-execute |
conditional-decode |
conditional-wait |
wait-statement |
identifier
wait-statement ::= WAIT (number)
action-list ::= *action-statement* { ; *action-statement* }*
action-statement ::= (*action-list*) { NEXT *action-statement* } * |
action-element
action-element ::= *action* { NEXT *action-statement* } * |
action { ; *action* }
labelled-statement ::= identifier := { *action* | *action-statement* }
conditional-execute ::= (*expression* ⇒ *action-list*)
conditional-decode ::= (DECODE *expression* ⇒ *action-list*)
conditional-wait ::= (WAIT *single-register-access* ⇒ *action-list*)
term ::= number |
identifier |
register-access |

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.

(*expression*

shift ::= *term* |
term ↓ { + | − } number

negation ::= *shift* |
{ + | − } *shift*

factor ::= *negation* |
factor { * | ÷ } *negation*

sum ::= *factor* |
sum { + | − } *factor*

relation ::= *sum* |
sum { = | ≠ | < | ≤ | > | ≥ } *sum*

complement ::= *relation* |
¬ *relation*

conjunction ::= *complement* |
conjunction { ∧ | ≡ } *complement*

disjunction ::= *conjunction* |
disjunction { ∨ | ⊕ } *conjunction*

expression ::= *disjunction* { *modifier* | (}

arith-complement ::= *sum* |
¬ *sum*

arith-conjunction ::= *arith-complement* |
arith-conjunction { ∧ | ≡ } *arith-complement*

arith-disjunction ::= *arith-conjunction* |
arith-disjunction { ∨ | ⊕ } *arith-conjunction*

arith-expression ::= *arith-disjunction*

modifier ::= ! comment !

Reproduced with permission of the copyright owner. Further reproduction prohibited without permission.