THE UNIVERSITY OF MICHIGAN

SYSTEMS ENGINEERING LABORATORY

Department of Electrical and Computer Engineering
College of Engineering

SEL Technical Report No. 58

THE OPTIMAL DESIGN OF CENTRAL PROCESSOR DATA PATHS

by

George A. McClain

under the direction of
Professor Keki B. Irani

April 1972

Under contract with
Rome Air Development Center
Research and Technology Division
Griffiss Air Force Base, New York
Contract No. F30602-69-C-0214

ACKNOWLEDGMENTS

A number of people made significant contributions to my progress in this research by taking time from their own problems to listen to mine and discuss them with me. Surely to be included here are the members of my committee: Professors Galler, Meyer, and Westervelt. I would like all of them to know how appreciative I am of the interest and cooperation they extended to me.

I would especially like to thank my chairman, Professor Keki B. Irani. This research would not have been completed without his encouragement at many difficult periods during the research. I am very grateful for the long hours that he spent discussing this work with me, providing guidance and helpful suggestions, and carefully reading and rereading this dissertation.

This research was sponsored by the Rome Air Development Center at Griffiss Air Force Base under Contract Number F30602-69-C-0214. In addition, an IBM Resident Study Fellowship provided me with considerable financial support during the early years of my studies.

Most of the typing was done by Ms. Kass Barker and I wish to give her many thanks. Her constant patience throughout a number of frantic weeks was greatly appreciated.

Again, thank you all very much.

George McClain
April 1972

To

LINDA SUE

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

TABLE OF CONTENTS (Continued)

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

This list includes the principal symbols used in this report with a short note about their meanings. The order is approximately that in which they are introduced in the text.

o_i	an architecture operator
O	the set of architecture operators
o'_i	a hardware operator
O'	the set of hardware operators
z_i	a variable name
Z	a set of variable names
$\bar{e}[X, y]$	an expression whose input parameter set is X and whose operator is y
$e[X, Y]$	any expression whose input parameter set is contained in X and whose operator is an element of Y
w	the number of bits in a variable, register, port, etc.
s	a statement in the language
ρ	a partial ordering of a statement set
S	a statement group consisting of a set of statements and a partial ordering of the statements
$\mathcal{A}$	an architecture
r	an architecture register

R	a set of architecture registers
p	an architecture input port
P	a set of architecture input ports
q	an architecture output port
Q	a set of architecture output ports
I	an architecture instruction
$\mathcal{I}$	a set of architecture instructions
C	the architecture cost limit
φ	an instruction weighting factor
τ	an instruction time limit
$\mathcal{A}^c$	a compiled architecture
I^c	a compiled instruction
$\mathcal{I}^c$	a set of compiled instructions
g_{ij}	the jth algorithm for operator i
g	a sequence of algorithms
G_i	the set of algorithms for operator i
G	the algorithm library
u	a hardware unit
r'	a hardware register
R'_i	the set of hardware registers of unit i
p'	a unit output port or data path input port
P'_i	the set of output ports of unit i

P'_0	a set of data path input ports
q'	a unit input port or data path output port
Q'_i	the set of input ports of unit i
Q'_0	a set of data path output ports
C'_i	the cost of unit i
f'	a unit function
F'_i	the set of functions of unit i
δ	a delay value for a statement
$\vec{\delta}$	the accumulated delay value for a statement in a statement group
H	the hardware library
D	a data path
U	a set of units
M	a static connection matrix
C'	the cost of the data path
$M_n(i,j,k,l)$	a gate
$M(i,j,k,l)$	a bus, i.e., a set of gates
$\vec{M}$	a dynamic connection matrix
$\vec{M}(i,j,k,l)$	a particular gate of $M(i,j,k,l)$
F	the static function set
$\vec{F}$	the dynamic function set
ψ	a data path cycle

t_ψ	the execution time for cycle ψ
T	the data path cycle time
I'	a hardware instruction
$\mathcal{J}'$	a set of hardware instructions
WAIET	the weighted average instruction execution time
Φ	the reciprocal of WAIET
$h_i (i=1, \ldots 6)$	maps from a compiled architecture to a data path and hardware instruction set

ABSTRACT

OPTIMAL DESIGN OF CENTRAL PROCESSOR DATA PATHS

by
George A. McClain

Chairman: Keki B. Irani

The research described in this paper deals with the design automation of the central processor of a digital computing system. The principal aim has been to lay the groundwork for a new and unexplored area in the field of computer design. While conventional design automation systems provide the user with a convenient method of describing his design, we explore the possibility of giving the user the ability of describing the desired computer architecture along with the available building blocks with which it is to be implemented.

This leads to an investigation of the relationships between the architecture of a computing system and various hardware designs which implement that architecture. By developing a model of a central processor and a language for describing the architecture, we can study the manner in which the architecture influences the data path design and develop algorithms that will yield an optimal data path for a specified performance or cost.

The three inputs of this model are:

1) the architecture which describes the computer as the programmer sees it,

2) the algorithm library which gives a set of possible translations of the operators used in the architecture description language,

3) the hardware library which gives a set of hardware units to be used as the building blocks in constructing a data path.

These two libraries allow a large number of data paths to be constructed which implement the given architecture. The problem is to select the one data path that gives the lowest weighted average instruction execution time but which does not exceed a specified cost limit.

In order to evaluate a particular data path and architecture implementation, we need to know: the cost of the data path, the number of cycles required for each instruction, and the duration of a data path cycle.

The cost of the data path is taken to be the total cost of the individual units. The computation of the minimum number of cycles per instruction and the duration of each cycle are rather complex computations, but the techniques to compute these values have been developed and are presented in this dissertation.

Obviously a problem of this scope implies serious computational difficulties if a working implementation is to be developed. In constructing a system of computer programs to illustrate the potential of this research, we have chosen to assume independence among a few key parameters in order to compute estimates of the performance and cost of various sub-portions of data paths. With these assumptions

we can no longer guarantee an absolute optimal solution, but we find instead an optimal in a reduced solution space. The assumptions enable us to use our programs to find solutions for some substantial architecture descriptions in reasonable execution times. A number of examples are presented which illustrate both the model of a computer and the programming implementation we have developed.

CHAPTER I

INTRODUCTION

1.1 AN INTRODUCTION TO DESIGN AUTOMATION

There are a large number of separate activities in the design of a digital computer system. Over the last decade efforts have been made to apply design automation techniques to most of these areas. Breuer [3,4] and Breuer, et al. [5] describe the entire spectrum of design automation of computers. The various areas are outlined with a discussion of the goals, direction of research, and inherent problems in each of these, as well as summing up the state of the art. Other authors, Gerace [16] and Metze and Seshu [22] in particular have good introductory material in their papers about automating the logical design phase of computer design.

For some problems of design automation, a great deal of progress has been made to date. Areas where design automation aids are taken for granted now are overall system simulation, small card layout, circuit design, back panel wire routing, placement and partitioning of logic, logic simulation, and documentation of the design. The engineer has been relieved of a considerable amount of tedious design work by these programs, and facilities for checking and cross-referencing of the design have reduced significantly the number of errors reaching the hardware stage of development. The ability of design automation programs to drastically reduce the number of errors in a designer's final specifications for hardware has been, in fact, a crucial step in the successful development of systems using large scale integrated circuits. The difficulty, cost, and delay of making engineering changes in a design using a large scale integrated circuit technology would surely discourage manufacturers from developing this technology if the engineers would have to work without the benefit of current design automation tools.

Detailed knowledge of the work in these areas is not relevant to this research and no specific references are included in the bibliography to these very successful areas in design automation research. One rarely encounters a new paper in these areas today. The majority of papers being published in recent years are concerned with the problem of unifying the present system of design automation procedures and further easing the task of the designer by providing him with a standardized, man-oriented language in which he can express his design.

The standard practice today is to have the engineer, who is doing the logical design of the computer, express his design by drawing symbolic blocks on his work sheet which represent various gates, flip-flops, etc., and then draw lines between them to represent the connections. These work sheets often contain such an enormous amount of detail that the overall structure is quite obscured. What is conceptually a simple register appears in these documents as perhaps twenty pages of AND's, OR's, and NOT's, and no one but the designer himself is likely to understand (or for that matter care) about the fine, detailed structure. Authors such as Gerace [16], Gorman [17], Schlaeppi [24], and Shorr [25] point out that the detailed design of the hardware is specified implicitly but precisely by a definition of the facilities a processor has (registers, memories, and I/O busses) and exactly what operations and transformations occur on each cycle. A statement of the form $A + B \rightarrow C$ where A, B, and C are registers and + is understood as 2's complement addition can precisely define the hardware and interconnections as well as the sequencing and gating controls.

These authors are attempting to define languages which will allow the design to be done by specifying the flow chart or micro-program for the processor. Then there should be no need to draw out a register as a group of logic elements when it is defined in a complete and more easily understood manner by listing the transfers which involve the register. This not only simplifies the design process and improves communication between designers but promises great improvements in logical simulation and provides good, standardized documentation of the system earlier than is presently possible. (The designers can eventually generate good documentation but this, being separate documentation, tends to be done later after the design tasks are over.)

These languages differ widely in their form. Some authors have based their work on ALGOL [6, 7, 8,17], FORTRAN [22], Iverson Language [13, 19], or a register transfer type language [1, 2, 11, 25] but the requirements for the languages are invariably the same. The language must be flexible enough to allow wide variation in the amount of detail presented. Intricate timing relationships in certain facets of the hardware and parallel operations in the data path require a precise, detailed language, but often the specific implementation is either obvious or else is not critical and the language should allow the designer to express himself in broad terms with the translator automatically filling in the details.

A representative example of this approach is the work reported by J. A. Darringer [7, 8]. He describes a language derived from ALGOL-60 by the addition of a few specialized statements and constructions. In the resulting language the designer can declare the computer facilities for his design and describe the instruction set as operations on these facilities. The language provides the user with two standard sets of operators for defining instructions: register operations, such as AND and OR; and arithmetic operators, such as ADD and MULTIPLY. The language is sufficiently compact that a small computer can be completely described on a single page.

This description can then be used to simulate the desired computer for the designer. The designer can also evolve his design by replacing arithmetic operators with algorithms using register operators and eventually reach a description which is entirely in terms of the register operators. Darringer's language processor has the capability to extract and tabulate the details of the design from this description and this in turn can be implemented directly into hardware.

In general, it is assumed by these authors that the designer will understand quite clearly what the data path of the processor will be, what algorithms will be used, what branch and decision points occur in the sequencing of instructions, and what the execution time of each instruction will be. The languages only provide an efficient and convenient means of expressing this information.

Some important insight into the aims of these various efforts is gained by noticing the total lack of discussion of the optimizing strategies used in the various proposed translators. This results from the simple fact that no true optimization is required in these hardware compilers. Various logic implementations of adders or registers may be examined to find the cheapest. Also standard logic simplification techniques are employed, but parameters such as the width, delay, and interconnection of each data path unit, as well as the specific control sequencing are rigidly defined in the input specification either explicitly or implicitly. The designer has specified the design in such detail that there is no variable in the system organization which can be used for optimizing. Also there is no need to include cost or performance objectives for the design. The designer knows quite accurately what the resulting cost and performance will be.

At present, none of the systems described in the current literature has reached the level of development where it can be regarded as a production system. Various degrees of success have been reported and the progress seems encouraging, but these systems still being used for research rather than design.

Metze and Seshu [22] discuss the possibility of a language that expresses the processor architecture (the computer as it appears to the programmer) without implying detailed algorithms to implement it. This language would then be translated or compiled into a design with the aid of catalogues of algorithms and hardware. Their paper treats some of the general problem areas but offers little more than a definition of the objectives. The paper indicates that their work is still in the most preliminary stages, and Gorman [17] expresses doubt that this approach can be successful now.

More recently Bell and Newell [1,2] have defined the Instruction Set Processor language (ISP) which can be used to express the architecture of a central processor in a convenient and compact notation. In this language the amount of detail required is quite flexible and the user is able to express detail where it is desired, but omit it when it is not needed for an understanding of the design. An ISP description of a computer is intended to be a condensed, machine-readable description of the programming manual for the computer although it was developed primarily for use in describing computer systems in a text.

The approach that we take in this research is an attempt to attack the problem from a different point of view. We describe a generalized language with which the designer specifies the system architecture which the computer is to implement. It is intended that the designer has not yet developed a definite view of how the data path should best be organized. This language input is applied to a model of a central processor and analyzed to determine the optimal processor data path for that architecture. In this approach we have left essentially all the parameters of the design open for optimization. The form of the final data path design is not directly constrained by the input from the designer. We do not actually attempt to define a language for this process but only specify the characteristics that are required of the language. For the examples presented later in this report, a primitive language is informally defined and is made adequate only for the examples being considered. For a full implementation of this research, the work of Bell and Newell or Metze and Seshu would probably be quite useful in developing the needed language.

1.2 A STATEMENT OF THE RESEARCH PROBLEM

The basic problem is that of formalizing the design of a digital computer and developing algorithms and procedures which can be used in optimizing the design of the central processing unit. Specifically, we plan to develop techniques for creating the most suitable central processor data path for the system architecture which has been defined.

The term data path is used here to refer to a set of hardware logic units such as registers, adders, and counters and the interconnections between them for data transfers. System architecture means a description of the computing system as it appears to the programmer and is composed of definitions of such items as data and instruction word formats, addressing and indexing structure, and the operation of each instruction. Optimization requires determining that data path whose cost is less than the specified maximum cost and which yields the highest performance as measured by a weighted average instruction execution time.

As mentioned previously, the major success in design automation has been confined to the area of translating a detailed hardware design into a specification of the physical location of circuits and wires. The development of a data path is always done by engineers in a slow and tedious procedure.

This procedure is basically an iterative technique of making more or less intuitive changes in a data path and then determining the best implementation for each instruction on the new data path in order to evaluate the overall effect of the change on performance. Another change is made and the cycle is repeated . The engineers produce an implementation for each instruction which is usually optimal, but the time required to do this severely limits the number of different data paths which can be examined. As a result the initial data path design is most often done by a few very experienced and creative engineers and the success of the resulting computer rests heavily on their skills in making a good intuitive first pass design. The development of algorithms to perform this portion of the design would provide a powerful tool in the design process allowing faster and more thorough evaluation of alternative data path designs.

1.2.1 Basic Assumptions

On the broadest scale the number of variables relevant to the design of a computer is enormous and many of these presently are poorly defined and not well understood. In order to concentrate on the relationship between the instruction set and the data path, a number of these variables will be controlled through some basic assumptions.

The first assumption is that the architecture of the system has been established, perhaps due to some fundamental requirement for program compatibility with existing computers and hardware compatibility with existing I/O equipment. This means it is known precisely how the machine will appear to the programmer. The compromises involved in various word lengths, instruction formats, indexing and addressing schemes, etc., all of which do effect the final cost and performance of a computer system, have already been resolved.

The second assumption is that the information needed to evaluate the computing power is available. Since a weighted average instruction time is used for this calculation, the required input information will be the relative frequency of occurrence of each instruction. It will be assumed, then, that the percentage usage of each instruction is accurately known although we are aware that in practice this information may be very hard to obtain.

In addition to the performance evaluation information, a maximum allowable system cost must be supplied. The optimizing algorithms will then limit their search to data paths below this cost. This facility is provided because it is invariably the case that a processor is intended to reach a certain segment of the computer market. An absolute optimal design is rarely of interest but rather the local optimum which can be implemented at less than some fixed total cost.

Although these assumptions arise primarily because of the desire to examine the relation between the instruction set and the data path, it should be seen that they are essentially the same as those applicable to design by current techniques. In cases where these assumptions are not met because of some uncertainties in the system specifications, the optimizing algorithms developed in this study can be used to compare the various alternatives under consideration, for instance, by evaluating the cost and performance corresponding to different instruction sets or different word lengths.

1.3 OUTLINE OF THE RESEARCH

The remainder of this report is organized so that in Chapter II the model as well as the objectives of the research are presented verbally to prime the reader with the terminology and basic point of view used in the later chapters. It is frustrating and ineffective to try to define precisely all the various terms and the jargon of a computing system as they are used in this report. These terms will all be quite familiar in a general way but the exact connotation of each can best be refined by their occurrence in the context of Chapter II. It may be disconcerting to the reader at first to be unsure of what exactly is implied by "data path" or "bus" but by the end of this chapter the reader should feel confident that he understands our usage of these terms in this study.

In Chapter III the mathematical model which has been developed will be presented in its entirety. This model views the central processor as a set of transformational units with interconnections between the ports of the units. This model incorporates a number of unique features. One of these is the definition of a generalized hardware unit which performs data transformations as defined by sets of statements from the same language used to describe the architecture. A unit can be quite complex and may contain internal registers or storage units. In general any data path which can be represented as a set of units can itself be defined as a single unit.

Another unique feature of this model is the use of a partial ordering of the statements in the language rather than the normal full ordering implied by the list structure of most languages. This allows flexibility in implementing functions on a data path since the time relationship between various language statements is not defined in more detail than necessary to accomplish the intended operation.

In Chapter IV, we can use the model to state a number of useful problems. First, the most general question of optimal data path design can be formulated and a solution to this expressed. Because of the complexity of this solution, any computer implementation faces some serious limitations. So it is desirable in Chapter IV to define a set of more restricted versions of this general problem and look at the related solutions. One of the more interesting of these was selected for implementation by a system of computer programs. These programs are also described in Chapter IV.

Finally in Chapter V we look at some examples that illustrate this research and the particular system of computer programs that was developed.

CHAPTER II

OVERVIEW OF THE RESEARCH

2.1 GENERALITIES OF DATA PATH DESIGN

The overall objective of this research is to investigate the relationships between the architecture of a computing system and various hardware designs which implement that architecture. Many different hardware designs will appear identical to the programmer if they execute the identical architecture. These designs, however, may differ in their performance and cost. By developing a model of a central processor and a language for describing the architecture we can study the manner in which the architecture influences the data path design and then develop algorithms that will yield the most appropriate data path for a specified performance or cost. Currently this process is solvable only with thoroughly heuristic techniques.

We will state the design automation problem with which this research is concerned in its simplest form and then elaborate on each of the elements of the problem statement.

Given:

(1) A system architecture.

(2) Performance requirements and maximum cost.

(3) An algorithm library.

(4) A hardware library.

Utilize:

(1) Algorithms from the algorithm library to compile the architecture definition.

(2) Hardware units taken from the hardware library to implement the compiled architecture.

Determine:

(1) A unique, detailed data path.

(2) The microprogram to control the data path for the instruction set.

2.1.1 System Architecture

The system architecture specifies precisely how the computer is to appear to the programmer. It is composed of a definition of facilities and the list of instructions the computer is to perform. The facilities consist of input and output busses, with which the computer interacts with its environment, and a set of registers. The architecture specifies a width in bits for each of these.

The most important part of the architecture is the instruction set which is a set of language statement groups that define the data transformations accomplished by each instruction. There is one group for each instruction. These definitions refer to the facilities possessed by the computer and specify the desired contents of each register and output port after execution of the instruction as a function of the input ports and the initial contents of the registers. It is important that the set of statements that define an instruction be complete and unambiguous but it should not specify a particular algorithm. This can be accomplished by using a rich set of operators for defining architecture instructions and by providing a means to distinguish those portions of an instruction definition which must have a particular execution sequence from those which can be more freely reordered. That is, we recognize that the user will want to specify a particular ordering for some statements in the instruction definitions, but we want to be sure we do not require him to order statements which he knows need not have a particular ordering.

Examples of this would be the use of one symbol to designate multiply rather than defining the operation as a sequence of adds, shifts, and subtracts since the latter definition would greatly limit the choice of algorithms and data path configurations. Similarly, for the loading of two registers, the user should have freedom to say that it is immaterial which of these is done first.

2.1.2 Performance Requirements and Maximum Cost

There are two parts to the specification of performance requirements:

(1) A list of weighting factors for the instructions in the architecture.

(2) A maximum permitted execution time for each instruction.

The fundamental means of evaluating the performance of a data path will be to compute a weighted average instruction execution time for the instruction set. A single value of performance will be found by:

(1) Multiplying the minimum number of cycles for each instruction (on the particular data path) by the weighting factor for that instruction.

(2) Summing these values over the instruction set.

(3) Multiplying this sum by the cycle time of the data path.

The weighting factors used in this calculation are approximations to the frequency of occurrence of the instructions in the expected program environment of the computer. However, an instruction mix is only an estimate of processor usage and is generally derived from a statistical analysis of representative programs. For instructions of very low usage these figures are of questionable validity due to the limited data available and uncertainty over the representative nature of the programs chosen for analysis. Rather than take a theoretical tack and ignore this difficulty, it was decided to include a limit on the execution time of the instructions, as part of the performance requirements, which can be used to insure that a reasonable implementation is chosen for instructions of low usage.

The maximum system cost is simply a limit to the cost of the total data path as obtained by summing the costs of the individual data path units.

2.1.3 Algorithm Library

A unique feature of the approach to design automation taken in this study is the incorporation of an algorithm library and a "compilation" or "translation" phase of the solution. The language used to define the architecture instructions uses one set of operators while the definition of the hardware functions uses a different set of operators. The architecture is then compiled by selecting entries from the algorithm library which allow the architecture instructions to be expanded in terms of hardware operations.

The principal advantage of this approach is that by including many algorithms in the library to interpret each architecture operator, a wide range of specific implementations can be investigated. Thus, an architecture operation such as multiply can be implemented in hardware by a number of specific adding, subtracting, and shifting algorithms. But in addition we find there is no requirement to associate some understandable meaning with each operator in order to accomplish the optimization. The meaning is thoroughly and completely defined by the algorithm entries for the operators. Thus, to say some architecture operator o_1 is implemented by doing an o'_1 or two sequential o'_2 hardware operations is to define completely the o_1 architecture operator. There is no need for the optimization strategy to be concerned with what the o_1 architecture operator actually does to data. The development of the optimizing strategies can therefore be carried out in a very general manner and kept independent of any fixed set of operators.

Now the algorithm library itself is treated as being open-ended in two senses. First, new operators may be added to the set of architecture operators simply by adding new entries to the library to translate them. Second, some particular architecture operator can have a new algorithm added by making a simple addition of the algorithm in the appropriate location of the library. Furthermore, the addition of a new hardware operator implies only the modification or addition of algorithms to take advantage of the new hardware operator. In all of these instances the optimization strategy is not effected by these changes to the library. The effect is only to allow more flexibility in the definition of new architectures or the discovery of better data paths by exploring the new algorithms.

2.1.4 Hardware Library

As the algorithm library is a catalogue of different translations of the operators of architecture definitions, so the hardware library is a catalogue of different hardware units which implement the various operators of the translated or compiled architecture. In principle then, when we have a translated architecture, we can go to the hardware library and select a set of units which covers the set of operators required by the instruction set of the compiled architecture.

The hardware library is defined as a set of unit specifications where the entry for each unit includes:

(1) A description of the unit facilities.

(2) The unit cost.

(3) The function set of the unit.

The facilities of the unit can be input ports, output ports, and internal registers and each of these has a specified width measured in bits.

The cost simply is the total cost for the unit.

The most important part of a unit entry in the library is the set of language statement groups, called functions, which specify precisely the data transformation that can be performed by the unit. Each of these functions gives a definition of the contents of the output ports and internal registers as a function of the initial register contents and input ports using only operators from the set of hardware definition operators. Since we place no limit on the size of the function set or the complexity of a function, a unit may perform many different and intricate transformations using any of the hardware description operators. There is virtually no limit to the complexity of a unit described in the hardware library. Each function also has time delay information associated with it which gives the total time for that function to be performed.

Again, as in the algorithm library, the hardware library is doubly open-ended. First, a new unit can be added to the library which has some functions to implement operators not previously covered by library entries. Second, a new unit may be added which performs operators already covered by existing units but perhaps has different cost and delay figures. These changes to the library will perhaps lead to an improved solution to some particular architecture by expanding the set of possible solutions. However, they cause no change in the optimizing strategies and would require no change in a computer implementation of this model.

2.1.5 Data Paths and Microprograms

The result of the design and optimizing routines consists of a description of a data path and a set of sequencing charts or microprograms which specify the manner of implementing each architecture instruction on the data path.

The description of the data path is given as:

(1) A list of the hardware units selected from the hardware library.

(2) A description of the connections between the input and output ports of the hardware units.

The implementation of the instruction set is a cycle by cycle description of the operations of the data path for each instruction. This is essentially a microprogram for the architecture on the data path.

For each cycle this specifies:

(1) The connecting busses between unit ports which are being used and the bits of these busses which are being gated.

(2) The function that each unit is performing.

(3) The names of operands present in the data path at the end of each cycle and the registers in which they are stored.

In order to evaluate a particular data path and architecture implementation, we need to know:

(1) The cost of the data path.

(2) The number of cycles required for each instruction.

(3) The duration of a data path cycle.

The cost of the data path is taken to be the total cost of the individual units. Each unit cost is part of the hardware library entry for the unit. This computation is straightforward and the requirement for a valid data path is that this figure be less than the value included in the input data as the maximum permitted cost. We should observe, however, that in this model no allowance is made for the cost of the interconnecting busses nor for the cost of whatever hardware is required to control the data path. This latter cost would normally cover status and sequencing triggers, gate control triggers, clocking circuitry, and a control memory if the data path employed microprogram control. Generally speaking then, we see from this definition of cost that our attention is directed primarily at obtaining an optimal hardware unit set for an architecture. However, we will see that the set of busses does undergo a minimization process as part of the optimization algorithms.

2.1.6 Variables in the Problem Solution

Now that we have looked at the form of the initial inputs to the optimizing programs and the form that the solution will take, we must discuss the parameters that can be varied to explore the solution space of an architecture. We have chosen a very difficult problem that has a large number of independent parameters, and we should be prepared for difficulty in implementing a computer solution. The parameters considered in deriving a data path are:

(1) Selection of algorithms.

(2) Selection of hardware units.

(3) Selection of a data path bus configuration.

(4) Determination of the data path cycle time.

(5) Determination of the microprogram.

The selection of algorithms is basically a simple covering problem. We want to explore all the different possible covers but the number of these for a reasonable architecture and algorithm library tends to be very large and it is difficult to establish any convenient objective function which can be used to choose among them or search through them.

The selection of the hardware unit is also a simple covering problem. Here, however, it is complicated by a lack of structure among the various library entries. For instance, we have chosen not to require any monotonic relationships between the costs or delays of units which execute the same operator. Hence, we have little basis for predicting the characteristics of different covering sets based on an analysis of the operators or the evaluation of some similar covering sets. Each set requires an involved evaluation. Another complicating factor is the ability for each unit to perform many functions in serial or parallel order. We would nevertheless like to examine all sets of units with particular attention to the comparison between simple but quick units requiring a number of passes to achieve the desired transformation and complex but slow units capable of performing involved functions in a single pass.

For a given set of data path units an enormous number of different interconnecting bus patterns is usually possible. We will see that the search for an optimal busing configuration to maximize the performance of a unit set can be organized for an efficient search of the possibilities, but unfortunately we are generally handicapped by the exceptional number to be examined. As with the unit set selection, an important characteristic of different busing configurations that we want to explore is a comparison of short, simple data path cycles to long, complex cycles.

The evaluation of the data path cycle time from the individual unit delays can be formulated as a "longest path" problem, and a great deal is known about the solution of these problems. But what we see is that a reasonable model for a computer does not lead to a longest path problem having an efficient solution.

Finally the computation of a microprogram to implement the compiled architecture is itself a major undertaking. In the current world of computer design this task is usually divided among a group of engineers who work for weeks to find an optimal solution. For this model we do have a good and efficient solution to this problem.

The two most important facets of this are the assignment of operands to registers and the sequential ordering of the statements in each instruction. It will be recalled that the ordering of these statements was deliberately left vague to provide flexibility and it is in the computation of the microprogram that use is made of this freedom. We must look at many different assignments or groupings of statements into instruction cycles to find the one which yields the smallest number of cycles.

2.2 A SIMPLE EXAMPLE

For a simple example of the processes just described, consider an architecture with one instruction which is defined by the single architecture operator, o_1. We might write this instruction as:

$$R2 \leftarrow o_1(R1)$$

where R1 and R2 are the two architecture registers. Figure 2.1 is a graphic representation of this instruction. Let us assume the algorithm library has an entry that defines o_1 in terms of hardware operators o'_2 and o'_3 as shown in Figure 2.2. Then we compile the instruction by applying this algorithm to obtain the result depicted in Figure 2.3.

Going to the hardware library we find a unit, u_1, shown in Figure 2.4 which has functions for both o'_2 and o'_3. That is,

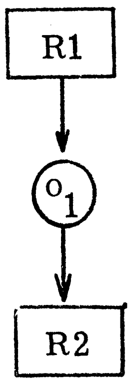


Figure 2.1. The Instruction Graph.

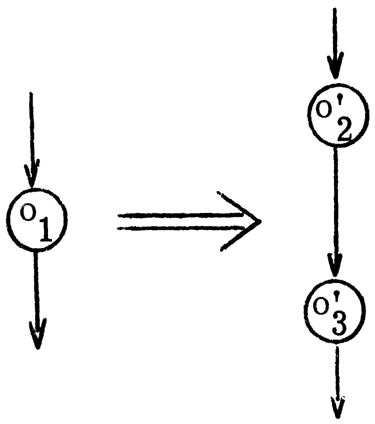


Figure 2.2. An Algorithm for o_1.

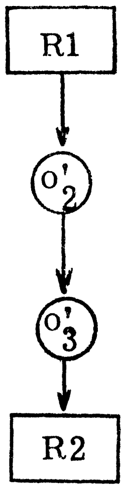


Figure 2.3. The Compiled Instruction.

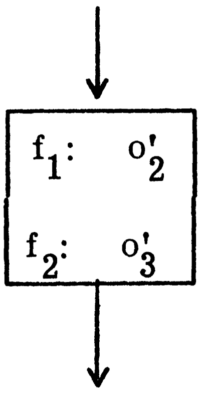


Figure 2.4. A Hardware Unit, u_1.

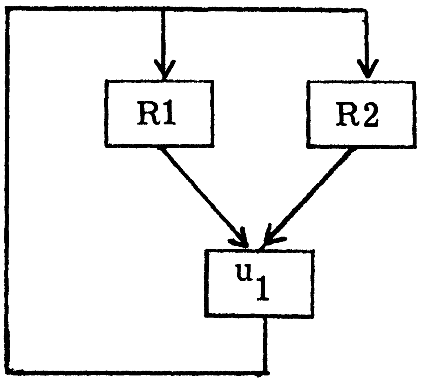


Figure 2.5. The Data Path.

cycle 1	$R1 \rightarrow u_1 \rightarrow R2$	u_1 does f_1
cycle 2	$R2 \rightarrow u_1 \rightarrow R2$	u_1 does f_2

Figure 2.6. The Microprogram.

u_1 performs two functions one of which is o'_2 and the other o'_3. In Figure 2.5 we have a possible data path constructed with two register units and u_1. An optimal microprogramming of the instruction is quite obvious. There are two possibilities but both require two data path cycles. We have arbitrarily chosen one of these and listed the data path cycles in Figure 2.6. This represents an implementation of the instruction on the data path.

This example illustrates the basics of compilation and microprogramming. With just one algorithm and one hardware unit we do not see the process of algorithm and hardware unit selection, and more importantly we have not mentioned the techniques that enable the various units to be assembled and connected into a data path. But these are complex problems and are the principal topics of Chapter IV and Chapter V.

CHAPTER III

MATHEMATICAL MODEL

In this chapter we present the model used to express and manipulate the problems posed in this research. The model provides the ability to express a desired computer architecture in terms of a set of architecture operators. Then the architecture is "compiled" by selecting algorithms from an algorithm library which translates the architecture operators into hardware operators. Hardware units, whose functions are expressed in terms of the hardware operators may then be chosen to match the operators required by the architecture. These units are formed into a data path by specifying the inter-unit connections. For the data path we get a cycle time dependent on the time delays in the units. Each instruction is then analyzed to determine the number of these cycles required to perform it, and this then allows an evaluation of the architecture performance on the data path.

The methods of performing this design and evaluation are the subject of Chapter IV. In this chapter individual components of the model are discussed in the following order: a common language, architecture, algorithms and the library, hardware units and the library, the data path, data path cycles, and hardware instructions and evaluation. Throughout this chapter a simple example will be developed to illustrate the model. This same example will also be carried into the next chapter where the solution techniques are presented.

3.1 A COMMON LANGUAGE

In this section we will define a language which is used throughout the model to describe operations on data. This language is used for defining:

(1) architecture instruction transformations

(2) algorithms

(3) hardware unit functions

(4) data path cycle transformations

(5) data path instruction transformations.

Actually we will be describing not a particular, specific language but rather the characteristics required of whatever language is defined for a particular implementation of this system. A concrete language definition would need far more attention to the details of legal variable names, statement formats, etc., than we present here, but this would only obscure the important points we are trying to present. For the examples contained in this chapter, a specific, simple language is used, but it is not the intention here to define that language in any greater detail than is necessary to allow the example to be understood.

The most important characteristics of this language are that:

(1) a transformation is defined by a set of statements

(2) the set of statements is given a partial ordering rather than a complete ordering in order to avoid any unnecessary specification of the execution sequence of statements.

(3) the language rules are the same for all uses of the language in the model although operands and operators may be restricted to certain subsets for use in various sections of the model.

This language is general and quite simple. The largest unit in the language is a statement group. A statement group is a set of statements with a partial ordering. Each statement contains one operator, one output variable name, and a set of input variable names. The partial ordering for the statement allows an explicit definition of the execution sequence of all the statements in a group and constitutes one of the unique features of the language. The ordering of the statements need not reflect structure that arises from the list format in which the statements are most conveniently written.

3.1.1 Definition of Terms in the Language

An operand is a binary data item. It has associated with it an attribute of width, w, which is the number of binary bits in the data item. For purpose of identification, the bits of an operand are labeled 1 through w with bit 1 being the most significant bit.

A _variable name_ is used to identify or refer to an operand. Variable names may be:

(1) the name of an architecture or hardware register

(2) the name of an input or output port

(3) a literal value

(4) data label, i.e., an operand not necessarily associated with a register or port.

Bit modifiers, which denote a subset of contiguous bits of the operand, may be appended to a variable name. We will denote this by a starting bit and a bit count enclosed in brackets after the variable name.

example: the variable name

REG1 [a,b]

refers to b bits starting with bit a of the operand named REG1.

An _operator_ is a function which maps from a set of input operands to an output operand. These operands are called the parameters of the operator. Each operator has a fixed number of input parameters and not more than one output parameter. Note, however, that an operator may have no inputs or no output.

We will consider the set of operators in the language to be divided into two disjoint sets. These are the set of operators used to describe architecture transformations, denoted by the symbol O, and the set of operators used to describe hardware transformations, denoted by the symbol O'. Individual operators are $o_i \in O$ and $o'_i \in O'$. Thus data path units never perform an o_i operation and architecture instructions are never described by an o'_i operator. The translation between these will be accomplished by the algorithm library and constitutes another of the unique features of this model. Both sets of operators are treated throughout this study without reference to any particular conventional transformations. That is, we do not assume that the characteristics of the operators are known or base the optimization procedure on a recognition of operator properties other than those properties presented in a particular algorithm library.

The single exception to this policy is the definition of two unique operators, $o_0 \in O$ and $o'_0 \in O'$, which we use to designate the operation of "transfer without alteration." These operators will have a single input and a single output parameter. The definition of these two, unique operators is a concession to problems which arise in a practical implementation of this model rather than being necessary to the model itself. It proves to be extremely helpful if the analysis procedure for determining a microprogram be capable of recognizing and manipulating these o_0 and o'_0 operators without the need to interact with the algorithm library.

An expression is an operator with an ordered set of variable names denoting the operator input parameters. An expression produces one new operand which is the result of the operator operating on these parameters. We denote an expression with the generalized notation of

$$\bar{e}[X, y]$$

where y is the operator and $X = (x_1, \ldots, x_n)$ are the variable names. An expression can have bit modifiers appended to signify a subset of contiguous bits of the operand produced by the expression.

As a convenience we broaden this notation so that we can write

$$e[X, Y]$$

to denote any single expression $\bar{e}[X', y]$ such that:

$$X' \subset X, \text{ a set of variable names}$$

$$y \in Y, \text{ a set of operator names.}$$

A statement is a pairing of an expression with a variable name and means that the operand produced by the expression becomes the new operand associated with that variable name. We denote a statement, s, as:

$$s = z \leftarrow \bar{e}[X, y]$$

where z is some variable name. As with the broadened notation used for expressions, we write

$$s = Z \leftarrow e\,[X, Y]$$

to mean any statement $s = z \leftarrow \bar{e}\,[X', y]$ where

$$z \in Z,\ X' \subset X \text{ and } y \in Y.$$

As an example of this notation, let R1, R2, and R3 be names of architecture registers; ADD be the operator of binary addition; and OR be the logical OR operator. Then the statement

$$R1 \leftarrow \bar{e}[\{R2, R3\}, ADD]$$

means the contents of register R1 is replaced by the sum of the contents of registers R2 and R3. Similarly the statement

$$\{R1, R2\} \leftarrow e[\{R1, R2\}, \{ADD, OR\}]$$

means any one of the following statements:

$$R1 \leftarrow \bar{e}[\{R1, R2\}, ADD]$$

$$R1 \leftarrow \bar{e}[\{R1, R2\}, OR]$$

$$R2 \leftarrow \bar{e}[\{R1, R2\}, ADD]$$

$$R2 \leftarrow \bar{e}[\{R1, R2\}, OR].$$

In many areas of this study it is necessary to refer to operands which are not associated with registers or ports. We will reserve the symbol Z or Z_1, Z_2, etc., to denote a set $\{z_i\}$ of data labels used to refer to operands which can not be labeled with register or port variable names.

3.1.2 Statement Groups

The transformations performed in a hardware function, architecture instruction, etc. will be described by an entity called a statement group. This is composed of a set of language statements and a partial ordering over that set. We define a statement group as a two-tuple

$$S = <\{s_i\}, \rho>$$

(subject to certain restrictions given below) where $\{s_i\}$ is used to denote the set of statements

$$\{s_i \mid s_i = Z \leftarrow e\,[X, Y]\}$$

in accordance with the definition of a statement given in Section 3.1.1.

By the partial ordering, ρ, we mean the relationship "a precedes b." Throughout this report we will write

$$a \rho b$$

to indicate a is ordered before b by ρ, and

$$a \not\rho b$$

to indicate a is not ordered before b by ρ. We will occasionally apply a subscript to ρ to designate a particular ordering when more than one is applicable to a set. This partial ordering has the characteristics that

$$\forall s_j, s_k, s_l \in \{s_i\}$$

(1) $s_j \not\rho\, s_j$

(2) $s_j \,\rho\, s_k \Longrightarrow s_k \not\rho\, s_j$

(3) $s_j \,\rho\, s_k$ and $s_k \,\rho\, s_l \Longrightarrow s_j \,\rho\, s_l$.

For convenience in referring to the two components of S we will write

$$\{s_i\}_S \quad \text{and} \quad \rho_S$$

to mean the statement group and partial ordering of the statement group respectively of

$$S = < \{s_i\}, \rho>.$$

Then we say $S = <\{s_i\}, \rho >$ is a <u>statement group</u> if and only if

$$\forall s_j,\ s_k \in \{s_i\}_S$$

where

$$s_j = z_j \leftarrow \bar{e}\,[X, y]$$
$$s_k = z_k \leftarrow \bar{e}\,[X', y']$$

if

$$z_j \epsilon X' \text{ then } s_j \rho s_k$$

or if

$$z_k \epsilon X \text{ then } s_k \rho s_j .$$

Finally we want to define the terms predecessor and descendant with respect to statements in a statement group, S. For $s_j, s_k \epsilon \{s_i\}_{S}$,

$$s_j \text{ is a predecessor of } s_k \text{ if } s_j \rho s_k$$

and similarly,

$$s_j \text{ is a descendant of } s_k \text{ if } s_k \rho s_j .$$

3.1.3 An Example of a Language

Now let us present one particular language which exhibits the characteristics we have been discussing. It will be useful throughout this report to have this available for use in various examples. The language has been kept very simple in order to avoid the need for a lengthy description. Also it is not the intention here to describe the language in sufficient detail to let the reader himself write in the language, but only to provide enough additional commentary so that the examples can be understood.

For variable names we have

r1, r2, . . .	architecture registers
r1', r2', . . .	hardware registers
p1, p2, . . . , q1, q2, . . .	architecture ports
p1', p2', . . . , q1', q2', . . .	hardware ports
A, B, C, . . .	data labels

We will indicate the bit modifiers, when needed, by a starting bit and a width enclosed in brackets. So

A[4,3]

means bits 4, 5, 6, of operand A where bit 4 is the most significant bit.

For the sets of operators O and O' we have

<u>O</u>

MOVE	transfer without alteration (o_0)
SL1	shift left 1 bit position
SL2	shift left 2 bit positions
ADD	binary addition
SUB	binary subtraction

$\underline{O}'$

move	transfer without alteration (o_0)
sl1	shift left 1 bit position
comp	2's complement
add	binary addition
sub	binary subtraction

Input operands for the operators are written in parenthesis following the operator. For ADD, add, SUB, sub which require two operands, these are separated by a comma and the second operand is taken as the minuend for the two subtraction operators. The operand name of the result in a statement is placed to the left of a left pointing arrow before the expression. In this language then we can have statements of the form:

$$r1 \leftarrow ADD(r1, r2)$$

The precedence relation between statements in a statement group is defined by associating a binary vector with each statement. The number of elements in the vector is the same as the number of statements in the statement group, and the ith element of each vector is to be associated with the ith statement in the statement group. The value of each element is interpreted as follows:

> If the ith element of the vector associated with statement j is a one, then statement i must precede statement j. If it is a zero, then statement i is not required to precede statement j.

Now if we use the same register name to indicate the original contents of a register as well as the new contents, we could have the following ambiguous pair of statements:

Statement Number	Statement	Binary Precedence Vector
1	$r2 \leftarrow$ move (r1)	oo
2	$r1 \leftarrow$ move (r2)	oo

Here the ordering vector declares that neither statement is required to be executed first. But then the transformation will depend on the specific ordering selected for the two statements. In addition, these two statements violate the definition for a statement group given in Section 3.1.2. which says that statement 1 must precede statement 2 if the output variable name of statement 1 (r2) occurs as an input parameter in statement 2 (and similarly statement 2 should precede statement 1).

The cause of this difficulty lies in the dual use of a name to designate both the register and the contents of the register. There is convenience in allowing this multiple use and in general it appears to be a natural and intuitive notational system. However, we need to resolve the confusion which arises between the new and old contents of a register when we refer to them both by simply the name of register.

The solution of this problem which was chosen for this language is to modify the register names with a bar when referring to the new contents. These two statements would then be written as:

$\overline{r2} \leftarrow$ move (r1)   00

$\overline{r1} \leftarrow$ move (r2)   00

and the transformation is understood to be an exchange of the contents of the two registers. A correct implementation of this statement group must, of course, take care to avoid destroying one of the original operands in the process of moving the other operand.

As an example of a statement group we can have the following:

Statement Number	Statement	Binary Precedence Vector
1	A ← ADD (r1, r2)	0000
2	$\overline{r3}$ ← SL1 (r1)	0000
3	$\overline{r1}$ ← SL2 (A)	1000
4	$\overline{r2}$ ← SL1 (r2)	0100

which says that the only required time relationships are that statement 1 must precede 3 and statement 2 must precede statement 4. The first of these relationships is derived from the use of an intermediate result in a later operation. The second relationship is a thoroughly arbitrary timing restriction imposed on the statement group.

3.2 ARCHITECTURE

There are essentially three components of the architecture: the registers and ports provide the data and receive the results; the instructions define the data transformations; and the cost specifies the maximum system cost.

We describe the architecture of the computer, $\mathcal{A}$, formally as a five-tuple:

$$\mathcal{A} = \langle R, P, Q, \mathcal{J}, C \rangle$$

where $R = \{r_1, r_2, \ldots, r_n\}$ is a set of registers

$P = \{p_1, p_2, \ldots, p_m\}$ is a set of input ports which bring data into the computer

$Q = \{q_1, q_2, \ldots, q_\ell\}$ is a set of output ports which transmit data from the computer to its environment

$\mathcal{J} = \{I_1, I_2, \ldots, I_k\}$ is the set of instructions which the computer executes

C = the maximum permitted system cost

Each of the registers and ports has an attribute of width, w, which is the number of binary bits in the register or port.

We describe each instruction in the architecture as a three-tuple as follows:

$$I = < S, \emptyset, \tau > \text{for } I \in \mathcal{J}$$

where S is a statement group in the language such that

$$\forall s_j \in \{s_i\}_S, s_j = R \cup Q \cup Z \leftarrow e\ [R \cup P \cup Z, O]$$

with Z being a set of data labels and O being the set of architecture definition operators.

$\emptyset$ is the weighting factor for the instruction (frequency of occurrence in an instruction mix)

τ is the maximum permitted instruction execution time

As an example architecture, employing the language described in the previous section, let

$\mathcal{A} = <R, P, Q, \mathcal{J}, C>$

$R = \{r_1, r_2\}$

P = empty

Q = empty

w = 16 bits (for all registers)

$\mathcal{J} = \{I1, I2, I3\}$ where $I = < S, \emptyset, \tau>$

For I1

$S_1 = \overline{r2} \leftarrow \text{MOVE (r1)} \qquad \emptyset_1 = 0.30 \quad \tau_1 = 20$

For I2

$S_2 = A \leftarrow SL2\ (r2)$	00	$\phi_2 = 0.40$	$\tau_2 = 30$
$\overline{r1} \leftarrow ADD(r1, A)$	10		

For I3

$S_3 = B \leftarrow SL1(r1)$	00	$\phi = 0.30$	$\tau_3 = 40$
$\overline{r1} \leftarrow SUB\ (B, r2)$	10		

$C = 100$

3.3 ALGORITHMS AND THE ALGORITHM LIBRARY

The language employs two disjoint sets of operators, O and O'. The algorithm library provides a translation or mapping from the set O of architecture operators into the set O' of hardware operators. This is accomplished by letting each algorithm be a two-tuple consisting of a single statement using a particular $o_i \in O$ and a statement group where the statement group is a transformational equivalent of the first statement. It is not necessary, in general, for the statement group to use only operators from O'. In cases where particular statements in this group use O operators, then the application of the algorithm does not yield a completed translation of the initial statement. In this event we would require the application of other algorithms, applied to the new O operators, in order to complete the translation.

We define an algorithm, g_{ij}, as

$$g_{ij} = <s, S>$$

$$g_{ij} = <z_1 \leftarrow \bar{e}\,[\,Z_1, o_i]\,, S> =$$

$$<z_1 \leftarrow \bar{e}[\,Z_1, o_i]\,, <\{s_1\}_{S_{g_{ij}}}\,, \rho_{S_{g_{ij}}}>>$$

where, for

$$s_a \in \{s_1\}_{S_{g_{ij}}}$$

$$s_a = Z_2 \leftarrow e\ [\,Z_3, O \cup O']$$

(Z_2, Z_3 sets of data labels)

We leave the definition of an algorithm broad at this time and require only that

$$z_1 \in Z_2 \text{ and } Z_1 \subset Z_3$$

although other restrictions may be imposed on g_{ij} later.

A particular algorithm is similar in spirit to a programming macro. The single statement serves as a model for the use of the operator, and the statement group gives a functionally equivalent set of transformations.

Now let $G_i = \{\, g_{ij} \,\}$ be the set of all algorithms for $o_i \in O$ and define the algorithm library, G, as the union of all algorithm sets,

$$G = G_1 \cup G_2 \cup \ldots \cup G_n$$

where $n = |O|$, and g_{ij} is the jth algorithm for the operator o_i in G.

In all our later dealings with the algorithm library, we will assume the ability to select entries independently and mix them together in a single statement group. For this reason we must make the restriction that all entries in a particular library, G, use the same encoding system for operands. For instance, we can not permit algorithms of 2's complement notation to occur simultaneously with algorithms assuming a signed integer encoding since we do not have the facility to recognize the incompatability of these two encoding systems.

We have assumed the existence of two distinguished operators, $o_0 \epsilon O$ and $o'_0 \epsilon O'$, which are known to be the "transfer without alteration" of the operand. In manipulating statement groups we will make use of these known properties to replace, insert, or delete statements which use these operators. The form of these three manipulations is:

(1) replacement

$z_1 \leftarrow \bar{e}\,[X, o_0]$ becomes $z_1 \leftarrow \bar{e}\,[X, o'_0]$

(2) insertion

$z_1 \leftarrow \bar{e}\,[X, \rho_i]$ becomes $z_2 \leftarrow \bar{e}\,[X, o_i]$ 00

$z_1 \leftarrow \bar{e}\,[z_2, o_0]$ 10

or

$z_1 \leftarrow \bar{e}\,[X, o_i']$ becomes $z_2 \leftarrow \bar{e}\,[X, o_i']$ 00

$z_1 \leftarrow \bar{e}\,[z_2, o_0']$ 10

(3) deletion

$z_2 \leftarrow \bar{e}\,[X, o_i]$ 00 becomes $z_1 \leftarrow \bar{e}\,[X, o_i]$

$z_1 \leftarrow \bar{e}\,[z_2, o_0]$ 10

or

$z_2 \leftarrow \bar{e}\,[X, o_i']$ 00 becomes $z_1 \leftarrow \bar{e}\,[X, o_i']$

$z_1 \leftarrow \bar{e}\,[z_2, o_0']$ 10

In the example language introduced in Section 3.1.3, $|O| = 5$. Let us define a small set of algorithms for these as follows (with variable names A, B, C, D):

$g_{1,1}$	$B \leftarrow SL1(A)$:	$B \leftarrow sl1(A)$	
$g_{2,1}$	$B \leftarrow SL2(A)$:	$C \leftarrow SL1(A)$	00
		$B \leftarrow SL1(C)$	10
$g_{3,1}$	$C \leftarrow ADD(A, B)$:	$C \leftarrow add(A, B)$	
$g_{3,2}$	$C \leftarrow ADD(A, B)$:	$C \leftarrow add(B, A)$	
$g_{4,1}$	$C \leftarrow SUB(A, B)$:	$C \leftarrow sub(A, B)$	
$g_{4,2}$	$C \leftarrow SUB(A, B)$:	$D \leftarrow comp(B)$	00
		$C \leftarrow ADD(A, D)$	10

3.4 HARDWARE UNITS AND THE LIBRARY

A data path used in this model is composed of a set of units with interconnecting busses. A unit description has three principal components: the register and ports which are the sources and sinks of data, a set of functions which describe the capability of the unit for transforming the data, and a cost figure. So hardware unit, u_i, is defined as

$$u_i = < R'_i, P'_i, Q'_i, F_i, C'_i >$$

where

$R'_i = \{r'_{i1}, r'_{i2}, \ldots r'_{in}\}$ is a set of registers

$P'_i = \{p'_{i1}, p'_{i2}, \ldots, p'_{im}\}$ is a set of output ports

$Q'_i = \{q'_{i1}, q'_{i2}, \ldots, q'_{il}\}$ is a set of input ports

$F_i = \{f_{i1}, f_{i2}, \ldots, f_{ik}\}$ is a set of functions

$C'_i =$ the unit cost.

We define a <u>function</u>, $f_{ij} \in F_i$, of the unit, u_i, as:

$$f_{ij} = <S_{ij}, \{\delta_k\}_{ij}>$$

where S_{ij} is a statement group and $\{\delta_k\}_{ij}$ is the time delay values for the function.

The statement group S_{ij} has the property that

$$\forall s_l \in \{s_k\}_{S_{ij}}, s_l = R'_i \cup P'_i \cup Z \leftarrow e[R'_i \cup Q'_i \cup Z, O']$$

where Z is a set of variable names used to refer to operands which are intermediate results in the unit for this function. These may not necessarily be associated with a unit port or register.

The set of delays, $\{\delta_k\}_{ij}$, has the property that

$$\forall\ \delta_l \in \{\delta_k\}_{ij}$$

δ_l is the time required to execute statement

$$s_l \in \{s_k\}_{S_{ij}}$$

and, of course,

$$|\{\delta_k\}_{ij}| = |\{s_k\}_{S_{ij}}|.$$

The data path, which we will later construct, will consist of hardware units chosen from a hardware library. This library is an open ended set of hardware units and should include a representation of any kind of hardware unit which might be useful in a data path.

Define this hardware unit library, H, as

$$H = \{u_1, \ldots, u_n\}.$$

For our example we will have a hardware library containing four units: a register, and adder/shifter, a complement/shifter, and a subtraction unit. We can write this as follows:

$$H = \{u_1, u_2, u_3, u_4\}.$$

For unit u_1

$R'_1 = \{r1'\}$

$P'_1 = \{p1'\}$

$Q'_1 = \{q1'\}$

$w = 16$ (for $r1', p1', q1'$)

$f_{1,1}$: $p1' \leftarrow$ move ($r1'$)

$f_{1,2}$: $\overline{r1'} \leftarrow$ move ($q1'$)

$f_{1,3}$: $p1' \leftarrow$ move ($r1'$) 00

$\overline{r1'} \leftarrow$ move ($q1'$) 00

all delays, δ, for unit u_1, $\delta = 0$

$C'_1 = 10$

For unit u_2

R'_2 = empty

$P'_2 = \{p1'\}$

$Q'_2 = \{q1', q2'\}$

$w = 16$ (for $p1'$, $q1'$, $q2'$)

$f_{2,1}$: $p1' \leftarrow$ add ($q1', q2'$)

$f_{2,2}$: $p1' \leftarrow$ add ($q2', q1'$)

$f_{2,3}$: $p1' \leftarrow$ sl1 ($q2'$)

$f_{2,4}$: $A \leftarrow$ sl1 ($q2'$) 00

$p1' \leftarrow$ add ($A, q1'$) 10

$f_{2,5}$: A ← sl1 (q2') 00

pl' ← add (q1', A) 10

$\{\delta_1\}_{2,1} = \{\delta_1\}_{2,2} = \{\delta_1\}_{2,3} = \{10\}$

$\{\delta_1, \delta_2\}_{2,4} = \{\delta_1, \delta_2\}_{2,5} = \{5, 5\}$

$C'_2 = 40$

For unit u_3

R'_3 = empty

P'_3 = {p1'}

Q'_3 = {q1'}

w = 16 (for p1', q1')

$f_{3,1}$: p1' ← comp (q1')

$f_{3,2}$: p1' ← sl1 (q1')

$\{\delta_1\}_{3,1} = \{\delta_1\}_{3,2} = \{5\}$

$C'_3 = 25$

For unit u_4

R'_4 = empty

P'_4 = {p1'}

Q'_4 = {q1', q2'}

w = 16 (for p1', q1', q2')

$f_{4,1}$: p1' ← sub (q1', q2')

$\{\delta_1\}_{4,1} = \{10\}$

$C'_4 = 35$

3.5 THE DATA PATH

We define a data path, D, to be a four-tuple

$$D = < P'_0, Q'_0, U, M >$$

where

$P'_0 = \{p'_{0,1}, p'_{0,2}, \ldots, p'_{0,n}\}$ a set of input ports

$Q'_0 = \{q'_{0,1}, q'_{o,2}, \ldots, q'_{o,l}\}$ a set of output ports

$U = \{u_1, u_2, \ldots, u_m\}$ a set of units

M = a static connection matrix

As with ports in the architecture and hardware unit definitions, we associate an attribute of width, w, with each of the ports in P'_0 and Q'_0.

The units of D must be indexed since it is frequently necessary to identify particular units of the data path, but we must be careful to avoid confusion between the index of the unit in H, which is essentially a unit type number and the index of the unit in D, which is the individual identity of the unit in the data path. Through the remainder of this report we will use a superscript, when necessary, to indicate the unit index in H and a subscript to indicate the unit index in D.

Hence u_i^j refers to a unit having the index j in H and i in D. Most often, however, the H index is unimportant and will not be shown.

3.5.1 Data Path Registers and Port Sets

For data path D, let $m = |U|$ and define the set of registers, input ports, and output ports.

$$R' = R_1' \cup R_2' \cup \ldots \cup R_m'$$

$$P' = P_0' \cup P_1' \cup \ldots \cup P_m'$$

$$Q' = Q_0' \cup Q_1' \cup \ldots \cup Q_m'$$

where

R_i' = the register set of u_i for $i = 1, \ldots, m$

P_i' = the output port set of u_i for $i = 1, \ldots, m$

Q_i' = the input port set of u_i for $i = 1, \ldots, m$

The individual ports are then indexed as follows:

$p_{ij}' \in P'$ is the jth output port of u_i for $i \geq 1$ or the jth port of P_0' for $i = 0$.

Similarly

$q_{ij}' \in Q'$ is the jth input port of u_i for $i \geq 1$ or the jth port of Q_0' for $i = 0$.

It is helpful here to point out that the input port set, P', and the output port set, Q', receive their names from the fact that they are the source and sink of data respectively for the busses which interconnect the units. So P' consists of the input ports to the data path and the output ports of the units, while Q' consists of the output ports of the data path and the input ports of the units. We must keep in mind that the unit input ports, which take data into a unit, are sinks or output ports for the set of interconnecting busses. Likewise the unit output ports take data from a unit but are then sources or input ports for the interconnecting busses.

3.5.2 Data Path Connections

In describing the connections of a data path, we want first to identify the paths on which data can flow between units and then to be able to specify a particular subset of these paths in order to define the dynamic operation of the data path.

First, we define a <u>gate</u> as a three-tuple which designates a set of contiguous bits connecting two ports. Let a gate

$$M_n\,(i,j,k,l) = \langle x,y,w \rangle$$

denote the connection of w bits from $p'_{ij} \in P'$ to $q'_{kl} \in Q'$ as follows:

$$\begin{array}{lll} \text{bit } x & \text{to} & \text{bit } y \\ x+1 & \text{to} & y+1 \\ \cdot & & \cdot \\ \cdot & & \cdot \\ \cdot & & \cdot \\ x+w & \text{to} & y+w \end{array}$$

We collect all the gates for a particular source-sink port pair into a <u>bus</u>. So a bus

$$M(i,j,k,l) = \{M_n(i,j,k,l)\}$$

is the set of all gates from p'_{ij} to q'_{kl} for a data path D. If there are no bits connected between some p'_{ij} and q'_{kl}, then

$$M(i,j,k,l) = \text{empty}.$$

To describe the complete collection of interconnections for a data path, we define the <u>static connection matrix</u>

$$M = \bigcup_{K} M(i,j,k,l)$$

for $K = \{(i,j,k,l) \mid p'_{ij} \epsilon\ P' \text{ and } q_{kl}\ \epsilon\ Q'\}$

which is the set of all gates for a data path.

To specify the dynamic operation of the data path, we want to designate some gate for each bus where a transfer is desired, and no gate on the busses on which no data is to flow. At most one gate may be designated for each bus. A particular selection of gates is specified in the <u>dynamic connection matrix</u>, $\vec{M}$, which is defined as

$$\vec{M} = \bigcup_{K} \vec{M}\,(i,j,k,l)$$

for $K = \{(i,j,k,l) \mid p'_{ij} \in P' \text{ and } q'_{kl} \in Q'\}$

where

$$\vec{M}(i,j,k,l) \subset M(i,j,k,l)$$

and

$$|\vec{M}(i,j,k,l)| \leq 1.$$

3.5.3 Data Path Function Sets

Recall that a unit

$$u_i = < R'_i,\ P'_i,\ Q'_i,\ F_i, C'_i >$$

where

$$F_i = \{f_{i1}, f_{i2}, \ldots, f_{im}\}.$$

Now for a data path D having

$$U = \{u_1, u_2, \ldots, u_n\}$$

we can define the <u>static function set</u>

$$F = F_1 \cup F_2 \cup \ldots \cup F_n.$$

This is the set of all functions of the units of the data path.

As with the data path connections, we wish to specify a single function for each unit in order to define the dynamic operation of the units of the data path. This is done with the dynamic function set, $\vec{F}$ which is defined as

$$\vec{F} = \vec{F}_1 \cup \vec{F}_2 \cup \ldots \cup \vec{F}_n$$

where

$$\vec{F}_i \subset F_i \text{ and } |\vec{F}_i| \leq 1.$$

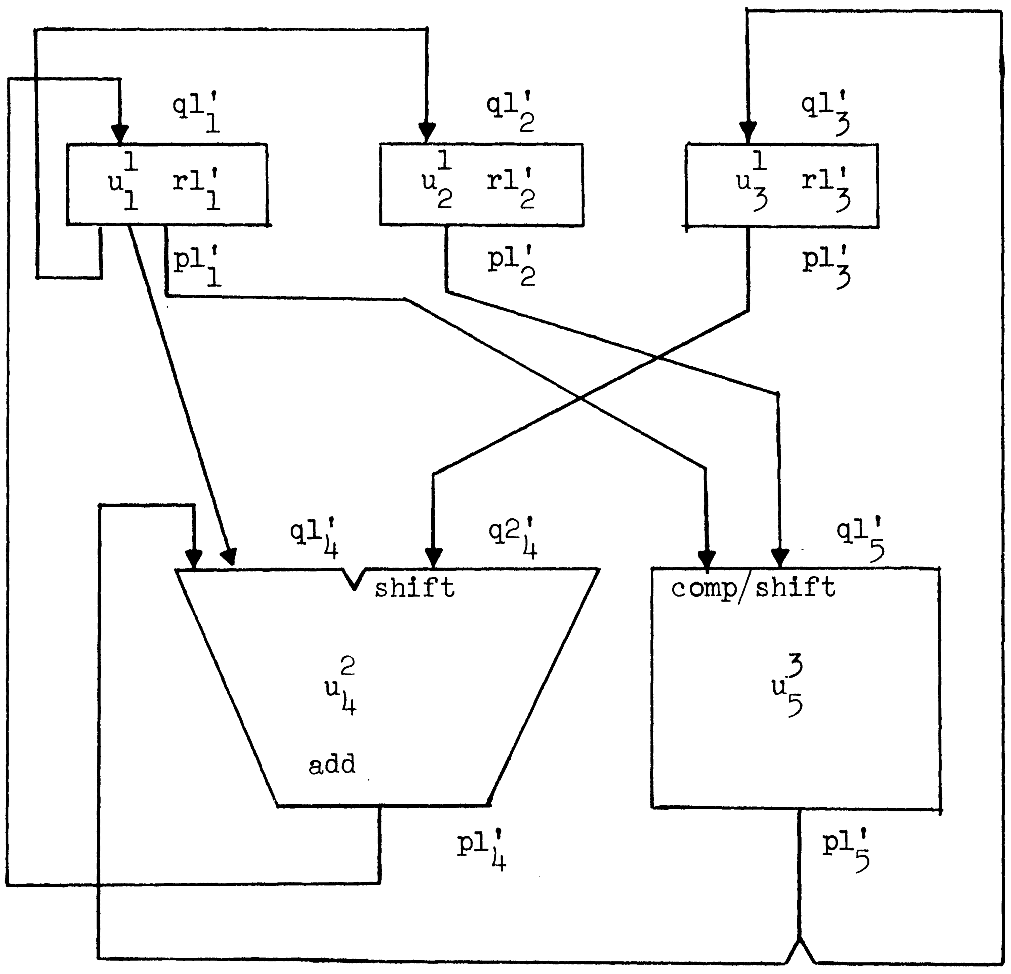
We indicate that some unit j is not performing any function by setting

$$\vec{F}_j = \text{empty}.$$

3.5.4 Data Path Example

To continue our example we have constructed one possible data path using units selected from the hardware library, H, defined in Section 3.4. This contains three registers, an adder/shifter, and a complement/shifter. That is, three copies of u_1, one of u_2, and one of u_3 from H. Figure 3.1 is a diagram of these units with the directed lines indicating the busses between the units. We will see that this data path is adequate for implementing the architecture described in Section 3.2, but it is not intended to be an optimal data path for that architecture in any way.

We can describe this data path as



Summary of unit specifications

Unit	Unit Type	Cost	Delay
1	1	10	0
2	1	10	0
3	1	10	0
4	2	40	10
5	3	25	5

Figure 3.1 Example data path

$$D = < P'_0, Q'_0, U, M>$$

P'_0 = empty

Q'_0 = empty

$$U = \{u^1_1, u^1_2, u^1_3, u^2_4, u^3_5\}.$$

Now we form the set of registers and ports

$$R' = \{r1'_1, r1'_2, r1'_3\}$$

$$P' = \{p1'_1, p1'_2, p1'_3, p1'_4, p1'_5\}$$

$$Q' = \{q1'_1, q1'_2, q1'_3, q1'_4, q2'_4, q1'_5\}.$$

All registers and ports in this data path have width 16, and we will have only one gate per bus which will be the entire 16 bits flowing between port pairs. So for all i, j, k, and l for which a bus exists we can say simply that the gates are

$$M_1(i,j,k,l) = <1,1,16>.$$

Then each bus contains only one gate

$$M(i,j,k,l) = \{M_1(i,j,k,l)\} = \{<1,1,16>\}.$$

We can list the busses as follows:

$$M(1,1,2,1) = \{<1,1,16>\}$$

$$M(1,1,4,1) = \{<1,1,16>\}$$

$$\vdots \qquad\qquad \vdots$$

but a more understandable format is to present the information as a matrix as in Figure 3.2.

Input Ports \ Output Ports	pl'_1	pl'_2	pl'_3	pl'_4	pl'_5
ql'_1	0	0	0	x	0
ql'_2	x	0	0	0	0
ql'_3	0	0	0	0	x
ql'_4	x	0	0	0	x
$q2'_4$	0	0	x	0	0
ql'_5	x	x	0	0	0

where

$$x = \{< 1,1,16 >\}$$

Figure 3.2 Example Static Connection Matrix

The static function set is the collection of all functions which the units can perform. For this data path we have

$$F = F_1 \cup F_2 \cup F_3 \cup F_4 \cup F_5$$

where

$$\begin{array}{llllcll}
F_1 & = & f^1_{1,1} & : & p1' & \leftarrow & \text{move } (r1') & \\
 & & f^1_{1,2} & : & \overline{r1'} & \leftarrow & \text{move } (q1') & \\
 & & f^1_{1,3} & : & p1' & \leftarrow & \text{move } (r1') & 00 \\
 & & & & \overline{r1'} & \leftarrow & \text{move } (q1') & 00 \\
F_2 & = & \text{same as } F_1 & & & & & \\
F_3 & = & \text{same as } F_1 & & & & & \\
F_4 & = & f^2_{4,1} & : & p1' & \leftarrow & \text{add } (q1', q2') & \\
 & & f^2_{4,2} & : & p1' & \leftarrow & \text{add } (q2', q1') & \\
 & & f^2_{4,3} & : & p1' & \leftarrow & \text{sl1 } (q2') & \\
 & & f^2_{4,4} & : & A & \leftarrow & \text{sl1 } (q2') & 00 \\
 & & & & p1' & \leftarrow & \text{add } (A, q1') & 10 \\
 & & f^2_{4,5} & : & A & \leftarrow & \text{sl1 } (q2') & 00 \\
 & & & & p1' & \leftarrow & \text{add } (q1', A) & 10 \\
F_5 & = & f^3_{5,1} & : & p1' & \leftarrow & \text{comp } (q1') & \\
 & & f^3_{5,2} & : & p1' & \leftarrow & \text{sl1 } (q1') &
\end{array}$$

3.6 DATA PATH CYCLES

For a data path, $D = < P'_0, Q'_0, U, M>$, we have seen how the interconnection of the units is described by the static connection matrix, M, and the set of all functions for the units is described by the static function set, F. We have also defined the dynamic connection matrix, $\vec{M}$, and the dynamic function set, $\vec{F}$, which are subsets of M and F. We intend to use $\vec{M}$ and $\vec{F}$ now to describe the dynamic operation of the data path as it performs some transformation on data. We will think of the dynamic operation as the flow of data out of registers, through transformational units, and back into registers. This register to register flow will constitute one cycle of the data path.

For a data path, D, we define a <u>cycle</u>, ψ, to be a two-tuple

$$\psi = < \vec{F}, \vec{M} >$$

where $\vec{F}$ is a dynamic function set of D

$\vec{M}$ is a dynamic connection matrix of D

This definition of ψ is very general and includes many combinations of $\vec{F}$ and $\vec{M}$ which may have no meaningful interpretation in terms of data transformations.

For a simple example using the data path defined in Section 3.5.4, we will add the contents of registers $r1'_1$ and $r1'_3$ and return the sum to $r1'_1$. With

$$\psi = \langle \vec{F}, \vec{M} \rangle$$

we would have

$$\vec{F} = \{f_{1,3}, f_{3,1}, f_{4,1}\}$$

or

$\vec{F} = p1'_1$	$\leftarrow$	move $(r1'_1)$	00	for unit 1
$\overline{r1'_1}$	$\leftarrow$	move $(q1'_1)$	00	
0				unit 2
$p1'_3$	$\leftarrow$	move $(r1'_3)$		unit 3
$p1'_4$	$\leftarrow$	add $(q1'_4, q2'_4)$		unit 4
0				unit 5

$$\vec{M} = \{M_1(1,1,4,1),\ M_1(3,1,4,2),\ M_1(4,1,1,1)\}$$

3.6.1 Cycle Statement Group

We need to be able to form a statement group which expresses the transformation defined by a particular cycle of the data path. We develop the cycle statement group in this section with a two-step approach: first, a constructional step to form a set of statements and a partial ordering; and second, a step to determine whether these statements and the ordering can be combined to form a valid statement group. The statement group is obtained by collecting the statement groups defined by the unit functions and adding a set of statements to express the gating between units.

$$\text{For } \psi = < \vec{F}, \vec{M} >$$

we have

$$\vec{F} = \bigcup_{i=1}^{|U|} \vec{F}_i = \{f_{ij}\}$$

$$\vec{M} = \bigcup_{K} \vec{M}\ (i,j,k,l) = \{M_n(i,j,k,l)\}$$

for $K = \{(i,j,k,l) \mid p'_{ij} \in P' \text{ and } q'_{kl} \in Q'\}$.

Recall that

$$f_{ij} = < S_{ij}, \{\delta_k\}_{ij} >$$

where S_{ij} is the statement group defining the transformations of f_{ij}, and where $M_n(i,j,k,l) = <x,y,w>$ defines the contiguous bits of the gate.

So we form the set of statements which correspond to the gates in use for this cycle as:

$$\{s_b\}_m = \{s_b \mid s_b = q'_{kl}[y,w] \leftarrow \bar{e}[p'_{ij}[x,w], o'_0]$$

$$\Longleftrightarrow M_n(i,j,k,l) = <x,y,w> \text{ and}$$

$$M_n(i,j,k,l) \in \vec{M}\}.$$

Then we combine these statements with those from the unit functions to get the set of statements $\{s_i\}_\psi$ for the cycle ψ.

$$\{s_i\}_\psi = \{s_b\}_m \cup (\bigcup_K \{s_k\}_{S_{ij}})$$

where we define

$$K = \{S_{ij} \mid S_{ij} \text{ is the first element of the two-tuple}$$

$$f_{ij} = < S_{ij}, \{\delta_k\}_{ij} > \text{ and } f_{ij} \in \vec{F} \}.$$

Now, for this set of statements we define a partial ordering, ρ_ψ, as follows:

$$\forall s_j, s_k \in \{s_i\}_\psi$$

where

$$s_j = \{z_j\} \leftarrow e[Z_1, O']$$

$$s_k = Z_2 \leftarrow e[Z_3, O']$$

if $z_j \in R' \cup Q_0'$, then $s_j \not\rho s_k$

if $z_j \notin R' \cup Q_0'$, then $s_j \rho s_k \Longleftrightarrow$

either $z_j \in Z_3$ or

there exists $s_l = Z_4 \leftarrow e[Z_5, O']$ such that

$z_j \in Z_5$ and $s_l \rho s_k$.

We recognize that not all cycles, $\psi = <\vec{F}, \vec{M}>$, represent meaningful operations. Although we treat any cycle, ψ, as valid on the data path, we wish to limit the use of the cycle statement group S_ψ to cycles which can be meaningful in terms of data transformations. It is clear that for some combinations of functions and gates, it will not be possible to define a partial ordering on the resulting statements that is consistent with the definition of the ordering of statements in a statement group given in Section 3.1.2.. Even if the statements can be ordered satisfactorily, we wish to exclude cycles where some operand does not have a legitimate origin or destination. We make these concepts precise by saying:

$$S_\psi = <\{s_i\}_\psi \rho_\psi> \text{ is a } \underline{\text{cycle statement group}}$$

$$\Longleftrightarrow$$

(1) ρ_ψ exists

and

(2) $\forall s_j \in \{s_i\}_\psi$ where $s_j = \{z_j\} \leftarrow e\ [Z_1, O']$

either

$$z_j \in R' \cup Q'_0$$

or

$$z_j \notin R' \cup Q'_0 \quad \text{and}$$

there exists $s_k \epsilon \{s_i\}_\psi$ where $s_k = \{z_{j'}\} \leftarrow e[Z_2, O']$ such that

$$z_j \epsilon Z_2 \text{ and } s_j \rho s_k$$

and (3) $\forall s_j \epsilon \{s_i\}_\psi$ where $s_j = \{z_j\} \leftarrow e[Z_1, O']$

$\forall z_k \epsilon Z_1$

either $z_k \epsilon R' \cup P'_0$

or $z_k \notin R' \cup P'_0$ and

there exists $s_l \epsilon \{s_i\}_\psi$ where $s_l = \{z_k\} \leftarrow e[Z_2, O']$ such that

$$s_l \rho s_j.$$

Now recall the example of a cycle given in Section 3.6 which put the sum of registers rl'_1 and rl'_3 into rl'_1. We had

$$\psi = < \vec{F}, \vec{M} >$$

with

$$\vec{F} = \{f_{1,3}, f_{3,1}, f_{4,1}\}$$

or

$$\begin{array}{lll} \vec{F} = pl'_1 & \leftarrow \text{ move } (rl'_1) & 00 \\ \overline{rl'_1} & \leftarrow \text{ move } (ql'_1) & 00 \\ pl'_3 & \leftarrow \text{ move } (rl'_3) & \\ pl'_4 & \leftarrow \text{ add } (ql'_4, q2'_4) & \end{array}$$

$$\vec{M} = \{M_1(1,1,4,1),\ M_1(3,1,4,2),\ M_1(4,1,1,1)\}$$

The set of statements which correspond to the selected gating is,

$$\begin{aligned}\{s_b\}_m = q1'_4 &\leftarrow \text{move}\ (p1'_1)\\ q2'_4 &\leftarrow \text{move}\ (p1'_3)\\ q1'_1 &\leftarrow \text{move}\ (p1'_4)\end{aligned}$$

Forming the statement set $\{s_i\}_\psi$ and computing the precedence relationship ρ_ψ we have the cycle statement group S_ψ for this cycle. Table 3.1 represents this resulting statement group. Note that the bit modifiers have been omitted since in all statements we are referring to the entire sixteen bit operands.

Statement Number	Statement	Binary Precedence Vector
1	$p1'_1 \leftarrow$ move $(r1'_1)$	000 0000
2	$q1'_4 \leftarrow$ move $(p1'_1)$	100 0000
3	$p1'_3 \leftarrow$ move $(r1'_3)$	000 0000
4	$q2'_4 \leftarrow$ move $(p1'_3)$	001 0000
5	$p1'_4 \leftarrow$ add $(q1'_4, q2'_4)$	111 1000
6	$q1'_1 \leftarrow$ move $(p1'_4)$	111 1100
7	$\overline{r1'} \leftarrow$ move $(q1'_1)$	111 1110

Table 3.1 The Cycle Statement Group for ψ.

3.6.2 Data Path Cycle Time

Recall that each function j of unit i is a statement group with a set of delay values. We have

$$f_{ij} = < S_{ij}, \{\delta_k\}_{ij}> = <<\{s_k\}_{ij}, \rho_{ij}>, \{\delta_k\}_{ij}>$$

where δ_k is the time required to execute s_k in f_{ij}. In Section 3.6.1 we defined a statement group for $\psi = <\vec{F}, \vec{M}>$. Now we would like to calculate the time required to execute this statement group using the delay values for the functions.

For cycle $\psi = <\vec{F}, \vec{M}>$ we have

$$\{s_i\}_\psi = \{s_b\}_m \cup (\bigcup_K \{s_k\}_{S_{ij}})$$

where we define

$$K = \{S_{ij} | S_{ij} \text{ is the first element of the two-tuple}$$
$$f_{ij} = <S_{ij}, \{\delta_k\}_{ij} > \text{and } f_{ij} \in \vec{F} \}.$$

The statements $\{s_b\}_m$ correspond to the selected gating, $\vec{M}$, for the cycle. In this model we do not attempt to represent the delays associated with these busses and do not provide for a description of the physical separation of units. Therefore we will define the delays associated with the statements $\{s_b\}_m$ to be zero, i.e.,

$$\forall s_k \in \{s_b\}_m, \delta_k = 0.$$

Now to find the execution time of cycle ψ we define the accumulated execution time, $\vec{\delta}_k$, of each statement, $s_k \in \{s_i\}_{S_\psi}$ as follows:

$$\forall s_k \in \{s_i\}_{S_\psi},\ \vec{\delta}_k = \delta_k + \max_J\ [\ \vec{\delta}_j]$$

with $J = \{j\,|\,j$ an integer and $s_j\ \rho\ s_k\}$.

Then we let the execution time of cycle ψ be t_ψ where

$$t_\psi = \max_K\ [\ \vec{\delta}_k]$$

with $K = \{k\,|\,k$ an integer and $s_k \in \{s_i\}_{S_\psi}\}$.

Then the <u>data path cycle time</u>, T, is defined for a data path, D, as the longest cycle, ψ, that can be executed on D:

$$T = \max_\Psi\ [\ t_\psi]$$

with $\Psi = \{\psi\,|\,\psi$ is a cycle of D and ρ_ψ exists$\}$.

Since $\vec{\delta}_k$ requires the existence of ρ_ψ, and t_ψ is derived from $\vec{\delta}_k$, it follows that $\vec{\delta}_k$ and t_ψ, are defined only for a cycle ψ which possesses an S_ψ.

This definition of cycle time is for a synchronous computer having a fixed cycle length based on the longest possible path through the data path. To model a synchronous computer having a variable length cycle we can simply redefine T to be a variable

$$T = t_{\psi}$$

having a different value for each cycle ψ. We might consider the modeling of an asynchronous computer, but then the definition of cycle time is not meaningful. Rather the evaluation of execution time would be derived from an analysis of delay values assigned in some manner to the statements of instructions.

For an example of cycle execution time, refer to the cycle statement group in the example at the end of Section 3.6.1. This cycle is quite simple and by referring to the specifications for the unit functions in H we see that only one statement has a non-zero delay. That is the actual addition and has a delay of 10,

$$t_{\psi} = 10.$$

It is possible to have a cycle in this data path which requires the serial connection of units 4 and 5 and by observation we can see that for this cycle we have $t_{\psi} = 15$. This happens to be the longest possible cycle and hence becomes the data path cycle time, $T = 15$.

3.7 HARDWARE INSTRUCTIONS AND EVALUATION

3.7.1 Instruction Statement Group

Define a hardware instruction, I', as a sequence of cycles

$$I' = (\psi_1, \psi_2, \ldots, \psi_m).$$

If there is a S_ψ for each $\psi \epsilon$ I', then we can easily construct a statement group, $S_{I'}$, to express the transformations of I'. Let

$$\{s_{ij}\}_{S_{I'}} = \bigcup_{i=1}^{|I'|} \{s_j\}_{\psi_i}.$$

Here we retain a double index on the statements in this new set so that we can use the first subscript to indicate the cycle from which the statement came. Then

$$S_{I'} = <\{s_{ij}\}_{S_{I'}}, \rho_{S_{I'}}>$$

where we define $\rho_{S_{I'}}$, as follows:

$$\forall s_{kl} \text{ and } s_{mn} \epsilon \{s_{ij}\}_{S_{I'}}$$

(1) if $k < m$, then $s_{kl}\, \rho_{S_{I'}}\, s_{mn}$

(2) if $k > m$, then $s_{mn}\, \rho_{S_{I'}}\, s_{kl}$

(3) if $k = m$, then $s_{kl}\, \rho_{S_{I'}}\, s_{mn} \Longleftrightarrow s_l\, \rho_{\psi_k}\, s_n$

and $s_{mn}\, \rho_{S_{I'}}\, s_{kl} \Longleftrightarrow s_n\, \rho_{\psi_k}\, s_l$.

We call $S_{I'}$ the <u>instruction statement group</u> of I' and note that it will always exist if S_{ψ_i} exists for $i = 1, \ldots, |I'|$.

As an example of a hardware instruction, take the following two cycles on our example data path:

ψ_1 :	$pl'_2 \leftarrow$ move (rl'_2)	00000
	$ql'_5 \leftarrow$ move (pl'_2)	10000
	$pl'_5 \leftarrow$ sl1 (ql'_5)	11000
	$ql'_3 \leftarrow$ move (pl'_5)	11100
	$\overline{rl'_3} \leftarrow$ move (ql'_3)	11110
ψ_2 :	$pl'_1 \leftarrow$ move (rl'_1)	0000 0000
	$ql'_4 \leftarrow$ move (pl'_1)	1000 0000
	$pl'_3 \leftarrow$ move $(r3')$	0000 0000
	$q2'_4 \leftarrow$ move (pl'_3)	0010 0000
	$A \leftarrow$ sl1 $(q2'_4)$	0011 0000
	$pl'_4 \leftarrow$ add (A, ql'_4)	1111 1000
	$ql'_1 \leftarrow$ move (pl'_4)	1111 1100
	$\overline{rl'_1} \leftarrow$ move (ql'_1)	1111 1110

We are showing here only the statement groups for these cycles but the reader could easily construct the corresponding $\vec{F}$ and $\vec{M}$ for each if he so desired. For an instruction

$$I' = (\psi_1, \psi_2)$$

the statement group would be formed from the union of the two sets with all statements from ψ_1 being predecessors of all statements in ψ_2.

The complete statement group for I' has thirteen statements and ten of these are simple "move" statements which tend to obscure the basic transformation. We will rewrite this and delete these "move" statements with the proper adjustment of variable names and orderings:

$$\begin{array}{ccll} \overline{rl'_3} & \leftarrow & \mathrm{sll}(rl'_2) & 000 \\ A & \leftarrow & \mathrm{sll}(rl'_3) & 100 \\ \overline{rl'_1} & \leftarrow & \mathrm{add}(A, rl'_1) & 110 \end{array}$$

3.7.2 Performance Evaluation

For a given data path, D, and a given set of hardware instructions for that data path, we wish to compute a performance figure. This is done simply by computing a weighted average of the execution time for each instruction over the hardware instruction set. Let

$$\mathcal{I}' = \{I'_1, I'_2, \ldots, I'_n\} \text{ with } n = |\mathcal{I}'|$$

be the hardware instruction set with each instruction being a sequence of cycles,

$$I'_i = (\psi_{i1}, \psi_{i2}, \ldots, \psi_{im_i}) \text{ with } m_i = |I'_i|$$

The weighting factors for the instruction set are given by the set

$$\{\phi_1, \phi_2, \ldots, \phi_n\}$$

where ϕ_i is the weighting factor for I'_i. Now if the data path cycle time is T, we have the <u>weighted average instruction execution time</u>, WAIET, as

$$\text{WAIET} = T \cdot \sum_{i=1}^{n} m_i \cdot \phi_i .$$

This is simply the execution time for each instruction averaged over the instruction set using the specified weighting factors for each instruction. We can also define <u>performance</u>, Φ, as the reciprocal of the execution time,

$$\Phi = \frac{1}{\text{WAIET}}$$

Thus, a computer with increasing computing power is represented by increasing performance, Φ, or decreasing execution time, WAIET.

Further, if each instruction I'_i has a maximum permitted execution time, τ_i, we can say:

$\mathcal{J}'$ is a valid hardware instruction set on D if and only if for $i = 1, \ldots, |\mathcal{J}'|$

$$\tau_i > m_i \cdot T$$

For the example data path we have found

$$T = 15 .$$

Let us contine that example by assuming we had hardware instructions having:

Instruction	Number of Cycles	Maximum Execution Time	Weighting Factor
I1'	1	20	0.30
I2'	2	30	0.40
I3'	2	40	0.30

This obviously relates to the architecture example in Section 3.2. Also we can see the particular hardware instruction example of the previous section represents I2 as we will see later in this chapter.

Of course, in evaluating a data path you need not have any particular architecture in mind but in practice the hardware instructions will be derived from some architecture, and so the evaluation of the data path is effectively an evaluation of some architecture implemented on the data path.

In any event we now compute WAIET for these values as

$$\text{WAIET} = 15[(1)(0.30) + (2)(0.40) + (2)(0.30)]$$

$$\text{WAIET} = 25.50$$

$$\Phi = \frac{1}{\text{WAIET}} = 0.0392$$

As shown below the individual instruction execution times are less than the specific maximum execution times, making this a valid hardware instruction set.

$$I1' = (15)(1) = 15 \leq 20$$

$$I2' = (15)(2) = 30 \leq 30$$

$$I3' = (15)(2) = 30 \leq 40$$

3.8 TRANSFORMATIONAL EQUIVALENTS AND REALIZATIONS

We conclude the treatment of the mathematical model by defining the desired relationship between an architecture and a data path. The basis of these relationships lies in the compilation of a statement group for an architecture instruction, using entries from the algorithm library, into a statement group using hardware operators. Statements from the latter group are then partitioned into sets of statements which can be executed serially in data path cycles in a manner consistent with the original partial ordering of statements in the architecture instruction.

Of course for any architecture there will be a great number of data paths which satisfy the relationships specified in this model and it will be the object of the next chapter to discuss criteria for finding and evaluating some "best" data path from the set of satisfactory data paths.

3.8.1 Transformational Equivalents

The process of "compiling" an architecture involves taking each instruction in the architecture and replacing the architectural operators with hardware operators according to entries in the algorithm library. To understand this process we look first at the replacement of a single statement in a statement group.

Let $S_1 = <\{s_\ell\}, \rho>$ be a statement group and some $s_k \epsilon \{s_\ell\}_{S_1}$ be a particular statement where

$$s_k = z_k \leftarrow \bar{e}\ [\ Z_k, o_i] .$$

If for some $j \geq 1$ there is an algorithm, g_{ij}, in the algorithm library, then we can compile statement s_k. Algorithm library entries are of the form

$$g_{ij} = <z_i \leftarrow \bar{e}[\ Z_1, o_i]\ , S>$$

where g_{ij} is the jth library entry for operator o_i. Rewriting, we have

$$g_{ij} = <z_1 \leftarrow \bar{e}[\ Z_1, o_i]\ , <\{s_{\ell'}\}_{S_{g_{ij}}}\ , \rho_{S_{g_{ij}}}>>.$$

Now we form a new statement group, S_2, by replacing s_k by $\{s_{\ell'}\}_{S_{g_{ij}}}$ in the following manner:

Form the set of statements

$$\{s_{\ell''}\}_{S_2} = \{s_\ell\}_{S_1} \cup \{s_{\ell'}\}_{S_{g_{ij}}} - s_k .$$

Define a partial ordering ρ_{S_2} for the new statement set as follows:

$$\forall s_a,\ s_b \in \{s_{\ell''}\}_{S_2}$$

(1) if $s_a \in \{s_\ell\}_{S_1}$ and $s_b \in \{s_\ell\}_{S_1}$

then $s_a \rho_{S_2} s_b \Longleftrightarrow s_a \rho_{S_1} s_b$

or

(2) if $s_a \in \{s_{\ell'}\}_{S_{g_{ij}}}$ and $s_b \in \{s_{\ell'}\}_{S_{g_{ij}}}$

then $s_a \rho_{S_2} s_b \Longleftrightarrow s_a \rho_{S_{g_{ij}}} s_b$

or

(3) if $s_a \in \{s_\ell\}_{S_1}$ and $s_b \in \{s_{\ell'}\}_{S_{g_{ij}}}$

then $s_a \rho_{S_2} s_b \Longleftrightarrow s_a \rho_{S_1} s_k$.

We are not ready to define a new statement group using $\{s_{\ell''}\}_{S_2}$ and ρ_{S_2} since the variable names in $\{s_{\ell'}\}_{S_{g_{ij}}}$ may not be consistent with those in $\{s_\ell\}_{S_1}$. We must first construct $\{s_{\ell''}\}_{S_2'}$ from $\{s_{\ell''}\}_{S_2}$ by performing the following substitutions on the variable names:

We had:

$$s_k = z_k \leftarrow \bar{e}[Z_k, o_i] \quad \text{for} \quad s_k \in \{s_\ell\}_{S_1}$$

and

$$g_{ij} = \langle z_1 \leftarrow \bar{e}[Z_1, o_i], \langle \{s_{\ell'}\}_{S_{g_{ij}}}, \rho_{S_{g_{ij}}} \rangle\rangle$$

where

$$\forall s_a \in \{s_{\ell'}\}_{S_{g_{ij}}}, \; s_a = Z_2 \leftarrow e[Z_3, O \cup O'].$$

Let the ordered sets

$$Z_k = (z^k_1, \ldots, z^k_n)$$

and

$$Z_1 = (z^1_1, \ldots, z^1_n).$$

We know

$$n = |Z_k| = |Z_1|$$

because Z_1 and Z_k are the set of input parameters to the operator o_i and the cardinality of such sets is always fixed for a particular operator.

Also we know that

$$z_1 \in Z_2 \quad \text{and} \quad Z_1 \subset Z_3.$$

Now we can define the set of names,

$$Z_4 = \{z^4_1, \ldots, z^4_m\}$$

$$Z_4 = Z_2 - \{z_1\} = Z_3 - Z_1 .$$

This is the set of variable names which occur only locally in the algorithm and do not correspond to names in s_k. Now construct a set of unique names, Z^{*4}, (names not occurring in $\{s_{\ell''}\}$):

$$Z^{*4} = (z^{*4}_1, \ldots, z^{*4}_m).$$

We are now ready to form the new set of statements $\{s_{\ell''}\}_{S'_2}$ from $\{s_{\ell''}\}_{S_2}$ by making the following substitution of names in $\{s_{\ell''}\}S_2$:

$$\forall s_a \in \{s_{\ell''}\}_{S_2} \text{ such that } s_a \in \{s_{\ell'}\}_{S_{g_{ij}}} \text{ substitute:}$$

z_k for z_1

z^k_1 for z^1_1

$\vdots \quad\quad \vdots$

z^k_n for z^1_n

z^{*4}_1 for z^4_1

$\vdots \quad\quad \vdots$

z^{*4}_m for z^4_m

With this new set of statements, called $\{s_{\ell''}\}_{S_2'}$, we can define

$$S_2 = <\{s_{\ell''}\}_{S_2'}, \rho_{S_2}>.$$

We say S_2 is the <u>transformational equivalent</u> of S_1 under g_{ij} and write it

$$S_2 = S_1 * g_{ij}$$

when S_2 is formed from S_1 in accordance with the above procedure. Intuitively, S_2 performs the same processing of data as S_1 inasmuch as the algorithm g_{ij} specifies a statement group, S_{ij}, which performs the same processing of data as o_i. Since we have not specified which statement in S_1 is replaced, S_2 is not unique if S_1 contains more than one statement using o_i.

The substitution of an algorithm into a statement group is really quite simple in spite of the apparent complexity of the formal description. The problem of generating names is exactly the same as that of calling macros in a computer program. First the parameters of the calling statement are directly substituted for the dummy variable names in the macro definition. Second, if the macro definition uses internal symbolic names, these must be modified whenever the macro is inserted into the program so that multiple calls of the macro will not produce multiple copies of the same symbolic name in the program.

Let us extend this notation as follows:

For a sequence of algorithms,

$$g = (g_{i_1 j_1}, \ldots, g_{i_n j_n})$$

and a statement group, S_1, define

$$S_1 * g$$

to be the successive substitution of algorithms:

$$\begin{aligned} S_2 &= S_1 * g_{i_1 j_1} \\ S_3 &= S_2 * g_{i_2 j_2} \\ &\vdots \\ S_{n+1} &= S_n * g_{i_n j_n} \end{aligned}$$

We again call S_{n+1} the transformational equivalent of S_1 under g and write it

$$S_{n+1} = S_1 * g \; .$$

Since each individual transformation is not unique, it is clear that $S_1 * g$ is not unique.

To see this applied to our example, consider I2 of the architecture of Section 3.2. This had the following statement group:

$A \leftarrow SL2\ (r2)$	00
$\overline{r1} \leftarrow ADD\ (r1, A)$	10

The algorithm library in this example has an entry we would like to substitute which is:

$g_{3,1}$	$B \leftarrow SL2\ (A)$:	$C \leftarrow SL1\ (A)$	00
		$B \leftarrow SL1\ (C)$	10

We form the set of statements called $\{s_{\ell''}\}_{S_2}$ and get

$\{s_{\ell''}\}_{S_2}$	ρ_{S_2}
$C \leftarrow SL1(A)$	000
$B \leftarrow SL1(C)$	100
$\overline{r1} \leftarrow ADD(r1, A)$	110

After making the proper substitution of names we get

$\{s_{\ell''}\}_{S_2'}$	ρ_{S_2}
$C \leftarrow SL1(r2)$	000
$A \leftarrow SL1\ (C)$	100
$\overline{r1} \leftarrow ADD(r1, A)$	110

Finally we have $S_2 = \langle\{s_{\ell''}\}_{S_2'}, \rho_{S_2}\rangle$.

Returning to the algorithm library we find

$g_{2,1}$ $B \leftarrow SL1\ (A)$: $B \leftarrow sl1\ (A)$

Substituting this twice in the same manner as the previous algorithm we have the statement group

$C \leftarrow$ sl1 (r2)	000
$A \leftarrow$ sl1 (C)	100
$\overline{r1} \leftarrow$ ADD(r1, A)	110

Finally we go to the algorithm library for

$$g_{4,2} \quad C \leftarrow \mathrm{ADD}(A, B) : \; C \leftarrow \mathrm{add}\ (B, A)$$

and substituting this into our statement group, we have

$C \leftarrow$ sl1 (r2)	000
$A \leftarrow$ sl1 (C)	100
$\overline{r1} \leftarrow$ add (r1, A)	110

Now we could use the implicit algorithm described in Section 3.3 for inserting "move" operations into this statement group in order to arrive at a statement group similar in form to the one for the hardware instruction in Section 3.7.1 (before the "move" operations were deleted from that statement group). However, we will omit this step since it does not add any insight to the translation process.

3.8.2 Realizations

We now define

$$\mathcal{A}^c = \langle R, P, Q, \mathcal{J}^c, C \rangle$$

to be the <u>compilation</u> of architecture

$$\mathcal{A} = \langle R, P, Q, \mathcal{I}, C \rangle$$

where

$$\mathcal{I}^c = \{I_i^c\} \text{ with } I_i^c = \langle S_{I_i^c}, \phi_{I_i^c}, \tau_{I_i^c} \rangle$$

$$\mathcal{I} = \{I_i\} \text{ with } I_i = \langle S_{I_i}, \phi_{I_i}, \tau_{I_i} \rangle$$

if there exists a set $\{g^k\}$ of algorithm sequences with

$$g^k = (g^k_{i_1 j_1}, \ldots, g^k_{i_{m_k} j_{m_k}})$$

and

$$n = |\{g^k\}| = |\mathcal{I}^c| = |\mathcal{I}|$$

such that

$$S_{I^c_a} = S_{I_a} * g^a \text{ for } a = 1, \ldots, n$$

and also that for a = $1, \ldots, n$

$$\forall s_b \in \{s_\ell\}_{S_{I^c_a}}, s_b = R \cup Q \cup Z \leftarrow e[R \cup P \cup Z, O'].$$

The compiled architecture differs, then, from the initial architecture by having the architecture operators replaced by hardware operators. In this process the number of statements in the statement groups may increase, and there may be some additional data labels added to the set of variable names. Now we will treat the problem of relating the architecture to hardware instructions and a data path. This basically involves mapping the various variable names and partitioning the statement groups into data path cycles.

Given

an architecture $\mathcal{A} = \langle R, P, Q, \mathcal{I}, C \rangle$

a data path $D = \langle P'_0, Q'_0, U, M \rangle$

and a set of hardware instructions $\mathcal{I}' = \{I'_i\}$

where $I'_i = \langle S_{I'_i}, \phi_{I'_i}, \tau_{I'_i} \rangle$

We say

$$R = \langle D, \mathcal{I}' \rangle$$

is a <u>realization</u> of $\mathcal{A}$, for some compilation of $\mathcal{A}$,

$$\mathcal{A}^c = \langle R, P, Q, \mathcal{I}^c, C \rangle$$

if the following three conditions are met:

(1) there exists the following maps

$h_1 : R \rightarrow R'$ (not onto)

$h_2 : P \rightarrow P'_0$

$h_3 : Q \rightarrow Q'_0$

$h_4 : \mathcal{I}^c \rightarrow \mathcal{I}'$

$h_5 : Z \rightarrow R' \cup (P - P_0) \cup (Q - Q_0) \cup Z$ (not onto) where Z and Z' are a set of data labels in the statement groups that are part of $\mathcal{J}^c$ and $\mathcal{J}'$ respectively. That is, Z is defined

$$\text{for } a = 1, \ldots, |\mathcal{J}^c|$$

$$\forall s_k \in \{s_i\}_{S_{I_a}^c} \quad \text{where} \quad s_k = A \leftarrow e\,[A, O']$$

$$Z = A - R \cup P \cup Q$$

and Z' is defined

$$\text{for } a = 1, \ldots, |\mathcal{J}'|$$

$$\forall s_k \in \{s_i\}_{S_{I'_a}} \quad \text{where} \quad s_k = A' \leftarrow e\,[A', O']$$

$$Z' = A' - R' \cup P' \cup Q'.$$

In addition we require the set of maps

$$\{h_6^1, h_6^2, \ldots, h_6^n\} \quad \text{for} \quad n = |\mathcal{J}^c|$$

where

$$h_6^a : \{s_\ell\}_{S_{I_a^c}} \rightarrow \{s_{\ell'}\}_{S_{I'_b}} \qquad a = 1, \cdots, n$$

and (2) that these maps satisfy the following conditions:

$$\text{for } a = 1, \ldots, |\mathcal{J}^c|$$

$$\text{and } \forall s_k \in \{s_\ell\}_{S_{I_a^c}}$$

where

$$s_k = x \leftarrow \bar{e}\,[\,Y,\, o_i'\,]\,, \quad Y = (y_1, \ldots, y_m)$$

that

$$h_6^a(s_k) = s_k'$$

where

$$s_k' \in \{s_{\ell'}\}_{S_{h_4(I_a^c)}}$$

and

$$s_{k'} = x' \leftarrow \bar{e}\,[\,Y', o_i'\,]\,, \quad Y' = (y_1', \ldots, y_m')$$

with

$$\text{if } x \in R, \text{ then } h_1(x) = x' \text{ and } x' \in R'$$
$$\text{if } x \in Q, \text{ then } h_3(x) = x' \text{ and } x' \in Q_0'$$
$$\text{if } x \in Z, \text{ then } h_5(x) = x' \text{ and } x' \in R' \cup Z'$$

for $i = 1, \ldots, m$

$$\text{if } y_i \in R, \text{ then } h_1(y_i) = y_i' \text{ and } y_i' \in R'$$
$$\text{if } y_i \in P, \text{ then } h_2(y_i) = y_i' \text{ and } y_i' \in P_0'$$
$$\text{if } y_i \in Z, \text{ then } h_5(y_i) = y_i' \text{ and } y_i' \in R' \cup Z'$$

and (3) that these maps also satisfy the conditions that

$$\text{for } a = 1, \ldots, |\mathcal{J}^c|$$

$$\forall s_i, s_j \in \{s_\ell\}_{S_{I_a^c}}$$

$$h_6^a(s_i)\ \rho_\beta\ h_6^a(s_j) \Longleftarrow s_i\ \rho_\alpha\ s_j$$

and

$$h_6^a(s_j)\ \rho_\beta\ h_6^a(s_i) \Longleftarrow s_j\ \rho_\alpha\ s_i$$

where $\alpha = I_a^c$ and $\beta = h_4(I_a^c)$.

We should observe that the partial ordering for a hardware instruction, I_i', may be more complete than the ordering of the corresponding compiled instruction, I_i^c, since the statements in I_i' have been partitioned into sets which are in serially ordered cycles of the data path. In the ordering $\rho_{I_i'}$, all statements in cycle ψ_j precede statements in ψ_k for $j < k$. So two statements in I_i^c which are unordered by $\rho_{I_i^c}$ may be ordered by $\rho_{I_i'}$, if they are partitioned into different cycles. This is an important feature of the model in that it allows portions of an instruction to be defined in the architecture without a temporal relationship although in any specific implementation some further structure may be imposed. This structure may then be adjusted in accordance with some optimizing criteria.

If we repeat here the statement group for the hardware instruction of Section 3.7.1, called I2', and the translated architecture instruction developed in Section 3.8.1, which we will call $I2^c$, we can see the obvious similarity

I2':

$\overline{rl'_3}$	$\leftarrow$ sll (rl'_2)	000
A	$\leftarrow$ sll (rl'_3)	100
$\overline{rl'_1}$	$\leftarrow$ add (A, rl'_1)	110

$I2^c$:

C $\leftarrow$ sll (r2)	000
A $\leftarrow$ sll (C)	100
$\overline{rl}\leftarrow$ add (A, rl)	110

A realization of $\mathcal{A}$ on D would require translations of I1 and I3 which we will not bother to develop. But for this one instruction we can see the mappings involved in a realization will include:

$$h_1(rl) = rl'_1$$

$$h_1(r2) = rl'_2$$

$$\left.\begin{matrix} h_2 \\ h_3 \end{matrix}\right\} \quad \text{P and Q empty for } \mathcal{A}$$

$$h_4(I2^c) = I2'$$

$$h_5(A) = A$$

$$h_5(C) = rl'_3$$

We can interpret the second h_5 map as indicating that the operand referred to by the data label C in the compiled architecture is put into register $r1'_3$ during the executicn of I2'. In other instances the h_5 map could relate data labels from a compiled architecture to ports of hardware units or to data labels for operands not associated with ports.

The correspondence between statements for the two instructions (the h_6 map for $I2^c$) is obvious:

$$h_6(s_1) = s_1$$
$$h_6(s_2) = s_2$$
$$h_6(s_3) = s_3$$

A final check on the realization is to compute the cost of the data path, C', as the sum of the five unit costs,

$$C' = C'_1 + C'_2 + C'_3 + C'_4 + C'_5$$
$$C' = 10 + 10 + 10 + 40 + 25 = 95$$

and observe that this is less than the specified architecture cost, C,

$$C' < C = 100.$$

CHAPTER IV
PROBLEMS AND SOLUTIONS

4.1 THE GENERAL PROBLEM

With the mathematical model of a central processor and a computer architecture that we have developed, it is simple to state the general problem to be considered here. We wish to be able to specify a desired computer architecture and then have a set of optimization algorithms programmed on a computer that will compute the most suitable set of hardware units and interconnecting busses for that architecture. This will be done by making reference to a library of algorithms for compiling the architecture and a library of hardware units for implementing the compiled architecture. The criteria for optimization will be to maximize performance while keeping the total data path cost under some predefined limit.

In terms of the model, for some given:

architecture	$\mathcal{A} = \langle R, P, Q, \mathcal{J}, C \rangle$
algorithm library	$G = \{g_{ij}\}$
hardware library	$H = \{u_i\}$

we want to find:

realization	$\mathcal{R} = \langle D, \mathcal{J}' \rangle$

where D is a data path and $\mathcal{J}'$ is a hardware instruction set, such that

(1) $\mathcal{R}$ is a realization of $\mathcal{A}$

(2) Cost of D $\leq$ C

(3) $\Phi_{\mathcal{R}} \geq \Phi_{\mathcal{R}'}$ for any realization $\mathcal{R}'$ of $\mathcal{A}$

which utilizes G and H. ($\Phi_{\mathcal{R}'}$ is the performance of $\mathcal{R}'$.)

The statement of the problem in this form would have value in many situations. Principally, the problem as it is formulated would be applicable to actual data path design for a defined architecture. But additionally a system of programs to solve this general problem would allow evaluation of the implications of various modifications to a given architecture. Alternatively, evaluation of new hardware unit specifications or alternative algorithms could be accomplished by repeatedly applying these programs to a fixed architecture while substituting new entries in G and H.

4.2 A GENERAL SOLUTION

A present there does not seem to be a feasible approach to solving the general problem. In this section we will outline the direction in which a general solution must proceed and try to demonstrate the difficulties that are encountered. A later section will explore the results that can be attained for a subset of the general problem.

Four principal steps must be performed in order to arrive at one realization for an architecture:

(1) A sequence of algorithms must be selected from G for each architecture instruction statement group.

(2) A set of units must be chosen to cover the operators occurring in the compiled architecture.

(3) A set of busses must be selected to connect the units.

(4) The performance must be measured in order to evaluate this particular realization by computing the cycle time for the data path and the number of required cycles for each instruction.

Figure 4.1 is a flow chart illustrating how these four steps can be combined to produce a solution to the general problem. There are three sequential decision steps in this process which permit all possible solutions to be evaluated. Sections 4.2.1 through 4.2.4 will discuss the specific problems that occur in each of these.

4.2.1 Selecting Algorithms

For an architecture instruction, $I = \langle S, \emptyset, \tau \rangle$, we have a set of statements, $\{s_i\}_{S_I}$, which specify the data manipulations required for I. Each statement employs one operator, o_i, from the set of architecture operators. For o_i there is the set $\{G_i\}$ of algorithms in G where $g_{ij} \in G_i$ is the jth algorithm for o_i. The choice of some g_{ij} to translate o_i implies the substitution of the statement group $S_{g_{ij}}$ from g_{ij} into the statement group, S_I, of I at some statement s_b as defined in Section 3.8.

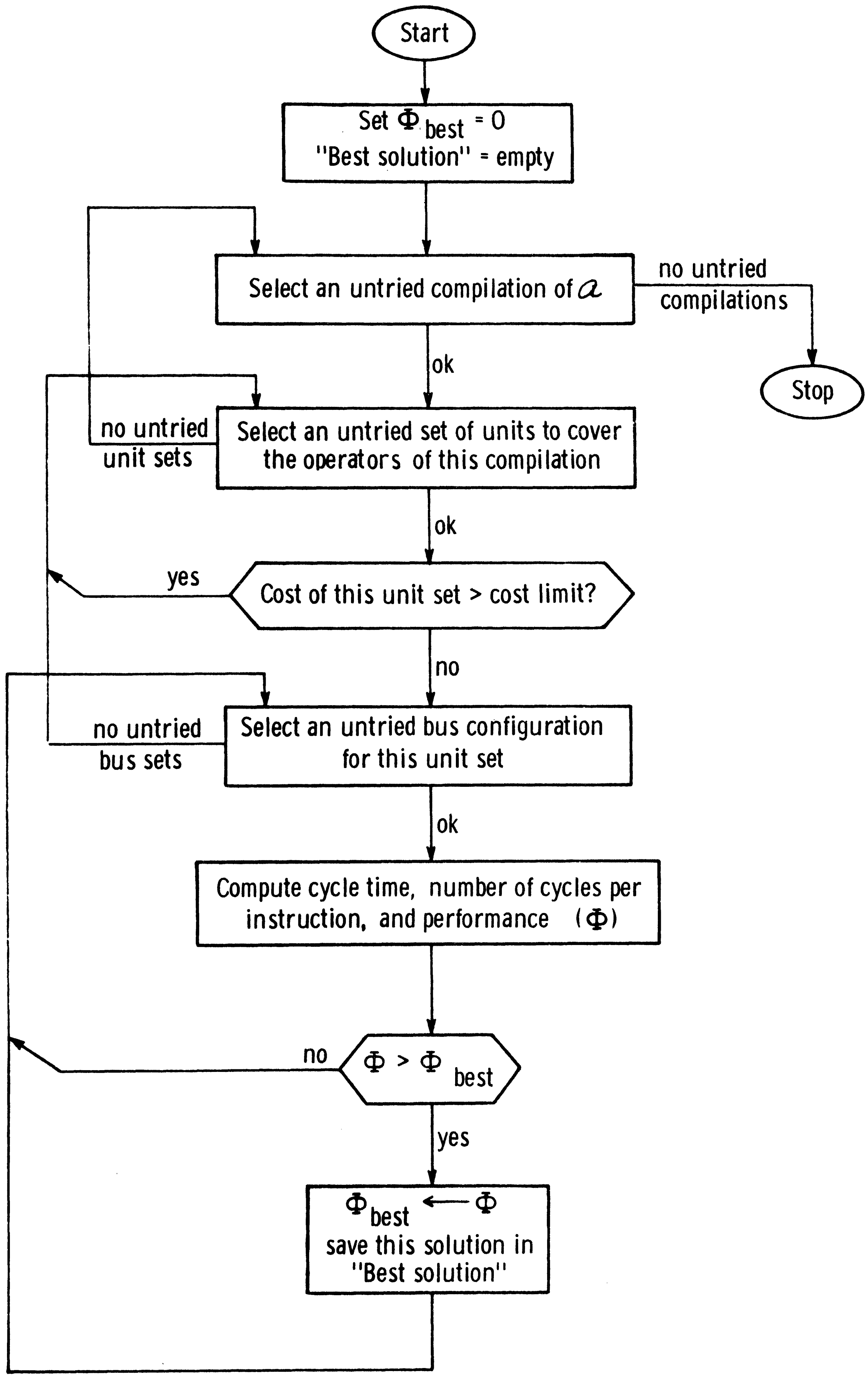

Start
Set Φ best = 0
"Best solution" = empty
Select an untried compilation of 𝒜
no untried compilations
Stop
ok
Select an untried set of units to cover the operators of this compilation
no untried unit sets
ok
Cost of this unit set > cost limit?
yes
no
Select an untried bus configuration for this unit set
no untried bus sets
ok
Compute cycle time, number of cycles per instruction, and performance (Φ)
Φ > Φ best
no
yes
Φ best ← Φ
save this solution in "Best solution"

Figure 4.1 Flow chart of the general solution

Now the substituted $S_{g_{ij}}$ may or may not contain statements using O operators. If not, then the compilation of statement s_b is completed. However, if $S_{g_{ij}}$ contains a statement using o_k, then the translation must proceed with a $g_{kn} \epsilon G_k$ for the o_k statement. This process must continue until all statements from I have been replaced by statements using only O' operators.

For some recursive structures in G it is possible that there are an infinite number of translations for a statement. As an example, for the statement s_1 and the algorithm g_{ij} where

$$s_1 = z_1 \leftarrow \bar{e}[\, Z_1, o_i]$$

$$g_{ij} = z_2 \leftarrow \bar{e}\,[\, Z_2, o_i\,] \;:\; z_3 \leftarrow \bar{e}[\, Z_2, o_i'] \qquad 00$$

$$z_2 \leftarrow \bar{e}[\, z_3, o_i] \qquad 10$$

we see that g_{ij} can be repeatedly applied without yielding a statement group containing only O' operators. This problem must be recognized and some provision made in any computer implementation to insure that the program will not remain in an infinite loop. An acceptable solution to this problem in practice might be to arbitrarily limit the number of times a given algorithm can be applied to a statement group. In any case, compilations employing unlimited, recursive application of an algorithm are of questionable value in any practical situation. So with careful programming the complete tree of algorithmic expansions of a statement group can be systematically ennumerated (with endless branches arbitrarily terminated).

To see how many possible expansions there might be for an architecture, let us assume

$$n = \frac{1}{|G|} \sum_{i=1}^{|G|} |G_i| = \text{average number of algorithms per architecture operator}$$

j = average number of sequential substitutions of algorithms needed to compile an algorithm into O' operators.

$$m = \sum_{i=1}^{|\mathcal{V}|} |\{s_k\}_{S_{I_i}}| = \text{number of statements or operators in an architecture}$$

Then there are

$$n^j$$

expansions of a single statement. And

$$(n^j)^m = n^{mj}$$

translations or compilations of the architecture assuming each statement is compiled independently. Note that this estimate takes into consideration the fact that different occurrences of an operator might best be compiled by different algorithms rather than assuming an operator should always be translated the same way in an architecture.

However, G does not contain information which would allow a direct evaluation of these compilations without implementing some hardware data path. All optimization criteria are in terms of cost and execution times which are properties of the hardware units rather than the algorithms. There is no way at this point to avoid a thorough search of the complete set of compilations. No evaluating function can be derived from the given constraints which would be useful in choosing a translation. We will see that it is precisely this problem which becomes the central issue in the practical solution taken up in later in this chapter. There we make the assumptions necessary to allow an evaluating function to be developed for algorithm selection.

We can realize from this estimate the advantage of restricting G so that each algorithm gives a translation entirely in terms of O' operators. In that case $j = 1$ and the total number of possible compilations for an architecture becomes

$$n^m$$

4.2.2 Selecting Data Path Units

After a compilation has been selected for all instructions, we have a compiled architecture,

$$\mathcal{A}^c = \langle R, P, Q, \mathcal{I}^c, C \rangle$$

where each instruction statement group has operators of O' only and operand names are from the set of architecture registers and ports or data labels $R \cup P \cup Q \cup Z$. We can collect the set of operators $\{o'_i\}$ which occur in this architecture so that units can be selected from the library, H, to implement these operators. Referring to Figure 4.1, we see this process is the second of three sequential decisions in the general solution.

In general there may be more than one unit in H for a given operator. We will iteratively select and evaluate one set of units to cover the required operators at a time and choose for our final set the one producing the highest performance. In order to consider the tradeoffs between cost and performance, we must include sets of units which contain varying numbers of units for each operator if there are multiple occurrences of the operator in an instruction. That is, if an operator o'_i occurs at most n times in a single instruction, then we should consider data paths with $1, 2, \ldots, n$ units which execute o'_i. And, just as different algorithms may be best for compiling different occurrences of the same operator in an architecture, so different units may prove best in implementing different occurrences of the same operator in a compiled architecture when multiple units are included for an operator.

To compute the total number of covering sets of units for a compiled architecture, let

a = number of different operators in a compiled architecture

b = average number of times each operator occurs in the compiled architecture

c = average number of units in H which execute each O' operation

Then there are c ways to cover each occurrence of an operator. We have

$$c^b$$

ways to cover b occurrences of an operator using b units. But we may choose to provide less than b units. Any number, 1, . . . , b, must be considered. We should then write the number of possible covers as

c	if one unit provided
c^2	if two units provided
.	
.	
.	
c^b	if b units provided

This gives

$$\sum_{i=1}^{b} c^i = \text{number of ways to cover all occurrences of one operator}$$

From this we have

$$\left[\sum_{i=1}^{b} c^{i}\right]^{a}$$

sets of units with which a data path can be designed for the compiled architecture.

It is not necessary to enumerate and evaluate this set completely. We can find criteria for eliminating some sets of units from consideration without a detailed performance evaluation. For instance, we need never consider data paths where some particular unit is duplicated more times than the corresponding operator occurs in any one instruction. That is, if operator o'_i occurs once in each of two instructions, we can consider using one copy of two different units (assuming H has two units for o'_i), but we should not consider using two copies of either unit. This puts an upper bound on the multiple occurrence of units. We have an upper limit on the total cost. By carefully ordering the enumeration of unit sets we can use this upper cost bound to skip groups of unit sets which we know would exceed this bound. Another observation is that additional copies of a unit can never cause a loss in performance. So we need not evaluate a data path unit set if it is contained in another unit set which is cost feasible and will itself be evaluated.

4.2.3 Selecting Interconnecting Busses

Given a set of units, U, and a compiled architecture, $\mathcal{A}^c = \langle R, P, Q, \mathcal{S}^c, C \rangle$, there are many possible interconnections of busses between the unit ports which will allow the architecture to be realized. The selection of a configuration of busses is the third sequential decision in the general solution of Figure 4.1. If there is an average of e inputs and f output ports per unit, then

$$(|U|\, e + |Q|) \cdot (|U|\, f + |P|)$$

is the total number of possible busses between ports and

$$2^{(|U|\, e + |Q|) \cdot (|U|\, f + |P|)}$$

is the total number of data paths which can be constructed from the set of units U and the port sets P and Q.

Since we assume a zero cost for busses, our selection process will not be economically motivated to remove extra busses. The basis for deleting busses from a data path which might contain all possible busses will be the improvement in performance that may be obtainable. The principal consideration, which arises when the bus configuration for a set of units is varied, is the tradeoff between a longer data path cycle having considerable serial computational power and a shorter cycle time requiring multiple cycles to accomplish a series of operations.

An example will illustrate the complexity of this phenomenon. Given the statement group shown in Figure 4. 2, consider the two data paths shown in Figure 4.3 and 4.4. which can implement it. Both DP1 and DP2 have the same set of units and differ only in the bussing configuration . But we see in Figure 4.3 that DP1 requires three cycles to implement the statement group while in Figure 4.4 DP2 requires four cycles.

The performance , Φ, for the two paths can be written

$$\Phi_{DP1} = \frac{1}{3T_{DP1}}$$

$$\Phi_{DP2} = \frac{1}{4T_{DP2}}$$

from which we conclude that

$$\Phi_{DP1} \leq \Phi_{DP2} \Longleftrightarrow 4T_{DP2} \leq 3T_{DP1}$$

$$\Phi_{DP1} \leq \Phi_{DP2} \Longleftrightarrow \max\,[t_1, t_2, t_3] \leq \frac{3}{4} \max\,[t_1, t_2 + t_3].$$

E ven in this simple example we see the relationship between the bussing and performance is not simple. The data path yielding minimum execution time depends on the relationship between the individual time delays of all three units.

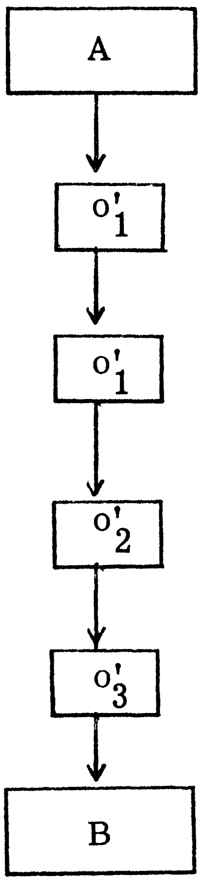
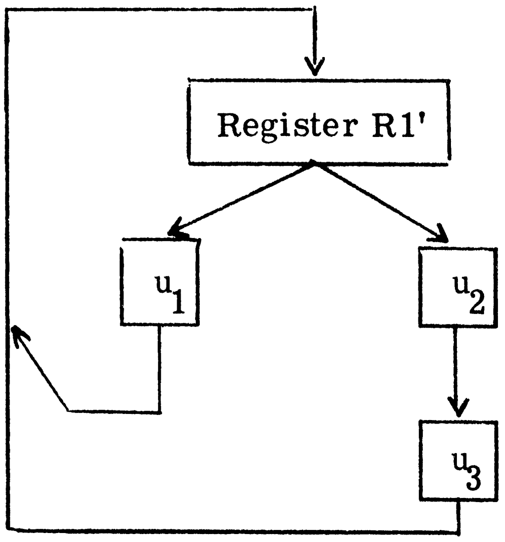


Figure 4. 2 An instruction graph to illustrate the variation of performance with variations in the bussing configuration

Data Path:



unit	operator	delay
1	o'_1	t_1
2	o'_2	t_2
3	o'_3	t_3

instruction implementation:

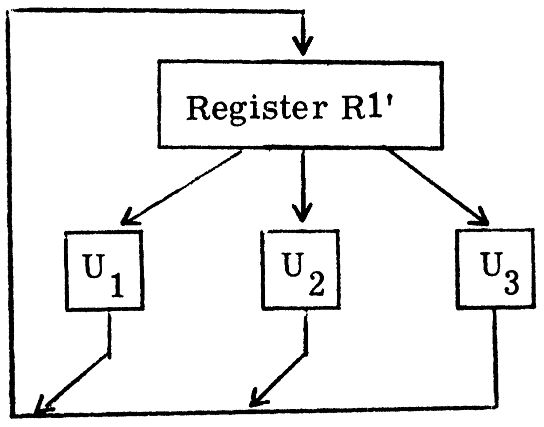
ψ_1:	$\overline{R1'} \leftarrow o'_1[R1']$	
ψ_2:	$\overline{R1'} \leftarrow o'_1[R1']$	
ψ_3:	$Z1 \leftarrow o'_2[R1']$	00
	$\overline{R1'} \leftarrow o'_3[Z1]$	10

data path cycle time

$$T_{DP1} = \max[t_1, (t_2 + t_3)]$$

Figure 4.3 Data path DP1 and the corresponding instruction implementation

Data Path:



unit	operator	delay
1	o'_1	t_1
2	o'_2	t_2
3	o'_3	t_3

instruction implementation:

$$\psi_1: \overline{R1'} \leftarrow o'_1 [R'1]$$

$$\psi_2: \overline{R1'} \leftarrow o'_1 [R1']$$

$$\psi_3: \overline{R1'} \leftarrow o'_2 [R1']$$

$$\psi_4: \overline{R1'} \leftarrow o'_3 [R1']$$

data path cycle time:

$$T_{DP2} = \max[t_1, t_2, t_3]$$

Figure 4.4 Data path DP2 and the corresponding instruction implmentation

It does not seem possible to develop any well behaved optimization function for the selection of the set of busses. Removal of any particular bus may either lower performance, if the chief result is an increase in the number of required cycles, or it may raise performance if the chief effect is to shorten the cycle time.

A few observations can be used however to significantly reduce the search required tofind the optimum configuration. For instance, the removal of any bus not in the longest or critical path can not decrease WAIET. Since no decrease in cycle time can occur, the execution time will be either the same, if no additional cycles must be added, or it will increase if the deletion of the bus implies additional cycles to implement the instruction. Another observation is that busses should not be considered which do not correspond to some transfer in the translated architecture. That is, if there is no transfer of operands in an instruction graph from operator o_i' to o_j', then in the data path we need not examine configurations which have a bus from the unit performing o_i' to the unit performing o_j'.

4.2.4 Evaluating the Data Path

The calculation of the performance of an architecture requires the knowledge of the data path cycle time, T, and the number of cycles, $|I_i'|$, needed to implement each architecture instruction. With these two values and the weighting factor for each instruction, ϕ_i, the weighted average instruction execution time, WAIET, can be easily computed. As we saw in Section 3.7.2, this is simply

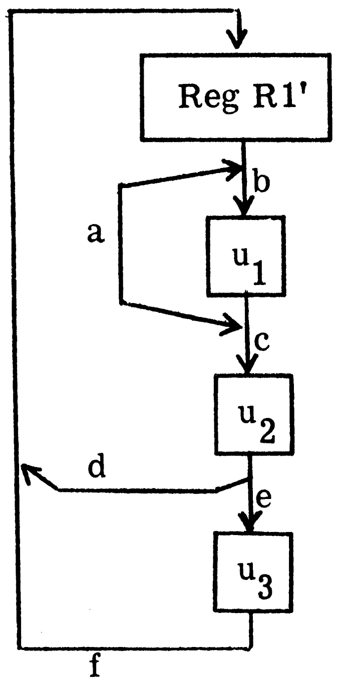
$$\text{WAIET} = T \cdot \sum_{i=1}^{|\mathcal{L}'|} |I_i'| \cdot \phi_i$$

We will now treat the problems of determining the two quantities T and |I'| needed for this computation. At this point our discussion will deal with the overall aspects of these problems. Later we will return to them again to present details of the algorithms actually used in the computer implementation.

4.2.4.1 Cycle Time Calculation

We being a discussion of the computation of the data path cycle time by noting that T is defined as the longest path in the data path and not the longest path actually used by the architecture. These two need not be the same. Presumably no busses would be left in a data path which are not used in some data transfer, but it does not follow that the longest path is necessarily used as such. The example shown in Figure 4.5 illustrates this distinction. This definition of cycle time was chosen rather than one based on the longest path actually used because it was felt that the cycle time should be solely a property of the data path and not a function of the use made of the data path. This allows changes or additions in the microprogramming of a computer to be made without the cycle time and performance of unchanged

Data Path:



unit	operator	delay
1	o'_1	t_1
2	o'_2	t_2
3	o'_3	t_3

Instruction implementation:

ψ_1:	$Z1 \leftarrow o'_1[R1']$	00
	$\overline{R1'} \leftarrow o'_2[Z1]$	10
ψ_2:	$Z2 \leftarrow o'_2[R1']$	00
	$\overline{R1'} \leftarrow o'_3[Z2]$	10

Data path cycle time:

$$T = t_1 + t_2 + t_3$$

Longest path used:

$$\max[t_1 + t_2,\ t_2 + t_3]$$

Figure 4.5 A simple data path showing the distinction between the longest path and the longest path actually used.

portions being affected. However, it also has the advantage of greatly simplifying the computational problem by allowing the instruction sequencing computation to be carried out independent of the cycle time computation.

To compute the data path cycle time we can represent the data path as a directed graph where the nodes correspond to units and the edges correspond to busses. We wish to compute the cycle time as the longest path starting and ending in a register and having no intervening registers or loops where the path length is the sum of the unit delays along the path. Since it is possible to have loops in the data path, the longest path can be infinite if these loops are included. Therefore we prohibit paths with repeated nodes from consideration in the cycle time calculation. This gives us a well defined problem which always has a finite solution and is consistent with the physical problem being modeled. That is, on one cycle of the data path we do not want more than one operand flowing through the same path.

As an example of a data path with a loop, consider Figure 4.6. There are busses a and b which form a loop or closed path from u_1 to u_2 and back to u_1. This would be useful if there were two instructions to be implemented such as:

(A and B are data labels)

I1':

$A \leftarrow o'_i\ [R1']$	00
$\overline{R1'} \leftarrow o'_2\ [A]$	10

I2':

$B \leftarrow o'_2\ [R1']$	00
$\overline{R1'} \leftarrow o'_3\ [B]$	10

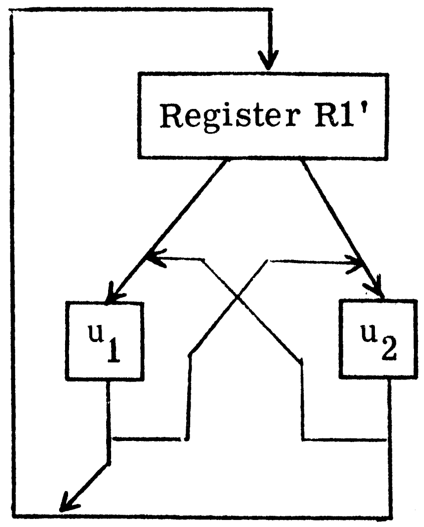
We see that all busses in the data path are used but the loop consisting of a and b is not used.

Now if no loops occurred in the graph, we could find the longest path of an n node graph with computation of the order of n^2. This is a well known result [9, 21] . However, for the longest path problem which we have here, where loops can occur in the graph, we must be prepared for n factorial computations. This computation is straightforward and can be performed easily enough, but it is also a well known result that no reduction in computational complexity can be expected.

4.2.4.2 Instruction Sequencing Computation

After a data path is completely fixed by specifying all units and defining all busses, the set of compiled instructions must be implemented

Data path:



unit	operation	delay
1	o'_1	t_1
2	o'_2	t_2

data path cycle time:

$$T = \max[\,(t_1 + t_2), (t_2 + t_1)] = t_1 + t_2$$

Figure 4.6 A data path with a loop.

on it. This problem is essentially the problem of "microprogramming" the instructions for the data path. This is indeed a formidable problem. The principal result we expect from this computation, when performed on a set of compiled instructions, $\mathcal{J}^c$, is a set of integers, $\{n_i\}$, such that n_i is the minimum number of data path cycles required to implement $I_i^c \in \mathcal{J}^c$. There are many ways this computation can be made. One specific method will be outlined later in discussing a particular computer implementation of this research. Now we will just list the variables involved in this computation. These can be grouped into three categories:

(1) Maps h_1, h_2, and h_3, where

$$h_1 : R \rightarrow R' \qquad \text{(not onto)}$$
$$h_2 : P \rightarrow P_0'$$
$$h_3 : Q \rightarrow Q_0'$$

must be defined. Generally this should not be a difficult process.

(2) For each instruction, $I_i^c \in \mathcal{J}^c$, the set of statements in the instruction statement group, $\{s_j\}_{S_{I_i^c}}$, must be partitioned into indexed sets $\{s_k\}_d$, $d = 1, \ldots, m_i$. The partitions, of course, will correspond to data path cycles. These partitions must satisfy the conditions:

(i) $\bigcup_{d=1}^{m_i} \{s_k\}_d = \{s_j\}_{S_{I_i^c}}$

and

(ii) $\forall s_a, s_b \in \{s_j\}_{S_{I_i^c}}$

if $s_a \in \{s_k\}_d$ and $s_b \in \{s_{k'}\}_{d'}$ and $d \neq d'$

then $s_a \, \rho_{S_{I_i^c}} \, s_b \Longrightarrow d < d'$

and $s_b \, \rho_{S_{I_i^c}} \, s_a \Longrightarrow d > d'$

(3) Each of the partitions from (2), i.e., subsets of statements from the compiled instructions, must be implemented as a cycle of the data path. This requires:

(i) assigning operands to registers

(ii) assigning unit functions to perform each statement in the partition

(iii) selecting gating to route the operands through the various units.

In category (2) there are many partitions that will satisfy the requirements. Similarly, in category (3) there may be many possible implementations for a partition or there may be none at all. The objective of this computation is to find the smallest number of partitions in category (2) such that each partition has at least one implementation in category (3).

The number of possible implementations for a given instruction is difficult to compute. However, when we realize that for b statements, there may be 2^b possible subsets for the first cycle, and in fact, 2^b possible subsets for each cycle since statements may be repeated, it is obvious the number of possible partitionings of statements grows rapidly with the cardinality of the instruction statement group. Also, if there are c registers in the data path, then there may be as many as c! possible assignments of operands to registers at the end of each cycle. From these two observations we suspect the solution space will be very large.

4.3 POSSIBLE SPECIAL CASE SOLUTIONS

In constructing and discussing the general solution, flow charted in Figure 4.1, we realized that an iterative structure with three sequential decision points resulted from the inability to formulate valid objective functions which are both independent of some problem variables and easily computed. For instance, in selecting a particular algorithm translation of an architecture, the only "evaluating function" we could find to compute a performance for this compilation is to examine all the possible data paths which can implement it. So it is necessary to iteratively enumerate the set of all possible solutions in order to be sure of discovering the optimal one. Basically, the performance is a sensitive and badly behaved function of the solution variables. A small permutation of one variable about its value in some known solution can result generally in a large and unpredictable change in performance for the new data path.

At this point many different tacks can be taken in attempting to achieve a workable computer solution to this problem:

First, a program to implement the general solution could be written. But the estimates of the size of the solution space we saw in the previous sections indicate that program execution time would be unreasonable for all but extremely simple problems with small values of $|G|$, $|H|$, and $|\mathcal{V}|$.

A number of ways can be found to significantly reduce the magnitude of the problem by making certain assumptions about the problem specifications. For instance, we could have one of three cases:

(1) require $|H| = |O'|$ and for $j = 1, \ldots, |f_i|$, where $f_{ij} \in f_i$ for unit $u_i \in H$, that the statement group of f_{ij} be simply

$$s = p'_{ik} \leftarrow \bar{e}[\, Q'_i, o'_i]$$

or

(2) require $|G| = |O|$ and for all j, $|G_j| = 1$, and

$$g_{j1} = <z_2 \leftarrow \bar{e}[\, Z_2, o_j] \,, <\{Z_3 \leftarrow \bar{e}[\, Z_4, O'] \,, \rho>>$$

or

(3) require $|\mathcal{J}| = 1$ and $I = <S, \emptyset, \tau>$

where

$$|\{s_i\}_S| \quad \text{is small} .$$

Each of these restrictions allows a reduction in the computations required to find the optimal solution. Case (1) essentially eliminates the need to search the hardware library. There is only one operator associated with each unit, so the selection process is trivial. Similarly case (2) eliminates the need to search the algorithm library since one and only one algorithm is applicable for each operator, and no sequential expansion of algorithms is needed to produce a compiled architecture. Finally, in case (3) where only one instruction occurs, it is possible to develop analysis procedures which greatly simplify the construction of the data path. These possibilities, although interesting and offering the possibility of allowing a workable computer implementation, will not be pursued in this study.

Another possibility for treating the general problem would be to compute an optimal solution subject to restrictions on the solution space such as:

(1) considering only data paths that do not duplicate units

or

(2) considering only data paths with nonserial or nonparallel interconnections of units.

Still another approach would be to compute a solution to the general problem using heuristic algorithms for the three decision points. These, however, would not necessarily produce an optimal solution. Considerable effort was expended in exploring this last possibility, but finally it was abandoned due to doubt in the ability of this approach to lead to any sort of consistently good solution.

4.4 ONE PARTICULAR IMPLEMENTATION

As part of this investigation it was decided to develop a prototype programming system to implement the ideas of this research and demonstrate the feasibility and usefulness of this model. Many possible subsets of the general problem were considered such as those discussed in Section 4.3 It was finally decided that the most successful approach would be to develop a programming system based on defining a limited solution space for a given problem. That is, we restrict the search for a solution by making certain assumptions about the nature of the problem.

Fundamentally the assumption which is made is that the cost and performance for each algorithm can be computed without consideration of a particular architecture. By manipulating the algorithm and hardware libraries we can construct a table which associates a large set of compilations and hardware realizations with each algorithm and, for each of these, we can compute a delay value. For a specified G and H, the table, called a GH Table, is constructed which gives the complete set of algorithms and possible data path unit sets for implementing each architecture operator and the execution time of the operator on that unit set. For a particular operator the table can have entries for every expansion of the algorithms and every possible hardware implementation of the expanded algorithm.

When we are given a particular architecture we can use the delay values for each operator in this table to select both a set of algorithms to compile the architecture and a set of hardware library units for the data path. That is, for each required architecture operator we pick one entry from among the set of entries for that operator based on the frequency of usage of the operators in the particular architecture. We then need only maximize the performance for this compilation and unit set without iteratively changing either of these.

There are two basic assumptions implied in this approach which limit the solution space that the program will explore. First, it assumes that the algorithms can be evaluated independently. During the selection process for a particular architecture we do not take into account the possibility of parallel execution of various operators in an instruction. In a truly optimal data path the implementation of each operator will be dependent on the other operators required by the instruction and the execution sequence imposed by the partial ordering of the statements in the instruction statement group. But we do not consider this at the time we make a selection from the GH Table. The second assumption on which this method is based is that we need choose only one entry for each architecture operator. We will compile every occurrence of an operator in an architecture in the same manner. We make a selection based on the overall importance of each operator (a weighted average computation is used) in the architecture as a whole and hence arrive at a single choice for compiling each operator.

A third limitation of the GH Table method, which is purely a practical programming limitation, arises because of the need to limit the size of the GH Table. If we use arbitrary bounds to terminate this process, there will be some realizations of an architecture that we will no longer be able to consider. For instance, if we limit the number of sequentially applied algorithms to n, then we will never explore a compilation of the architecture requiring n + 1 successive algorithm substitutions to translate one operator.

Now at first glance it might seem that the generation of the GH Table has all the difficulties we described for the general problem. In fact the flow chart of the GH Table generation, which will be discussed in the next section, is similar to Figure 4.1. However, the computational complexity is considerably reduced. The significant advantages that arise from this method are that: the computation of each algorithm expansion need be carried out only once for a given G and H; each algorithm can be expanded independently; and the performance evaluation of each one is enormously simiplified. We will see more about this when we discuss the individual components of the GH Table generation program.

The programming system required to implement the model in this manner consists of one set of programs used once for a given G and H to generate the GH Table, and another set used to compute a data path solution for each architecture. The remaining sections of this chapter provide a discussion of these programs.

4.4.1 GH Table Generation

Figure 4.7 is a flow chart of the program which takes descriptions of G and H and forms the GH Table. It has two sequential decisions for each algorithm library entry: expanding or compiling the algorithm, and assigning units to the compiled algorithm. For each architecture operator there is a group of GH Table entries. Each entry consists of:

(1) a delay value

(2) a sequence of algorithms

(3) a set of units from the hardware library

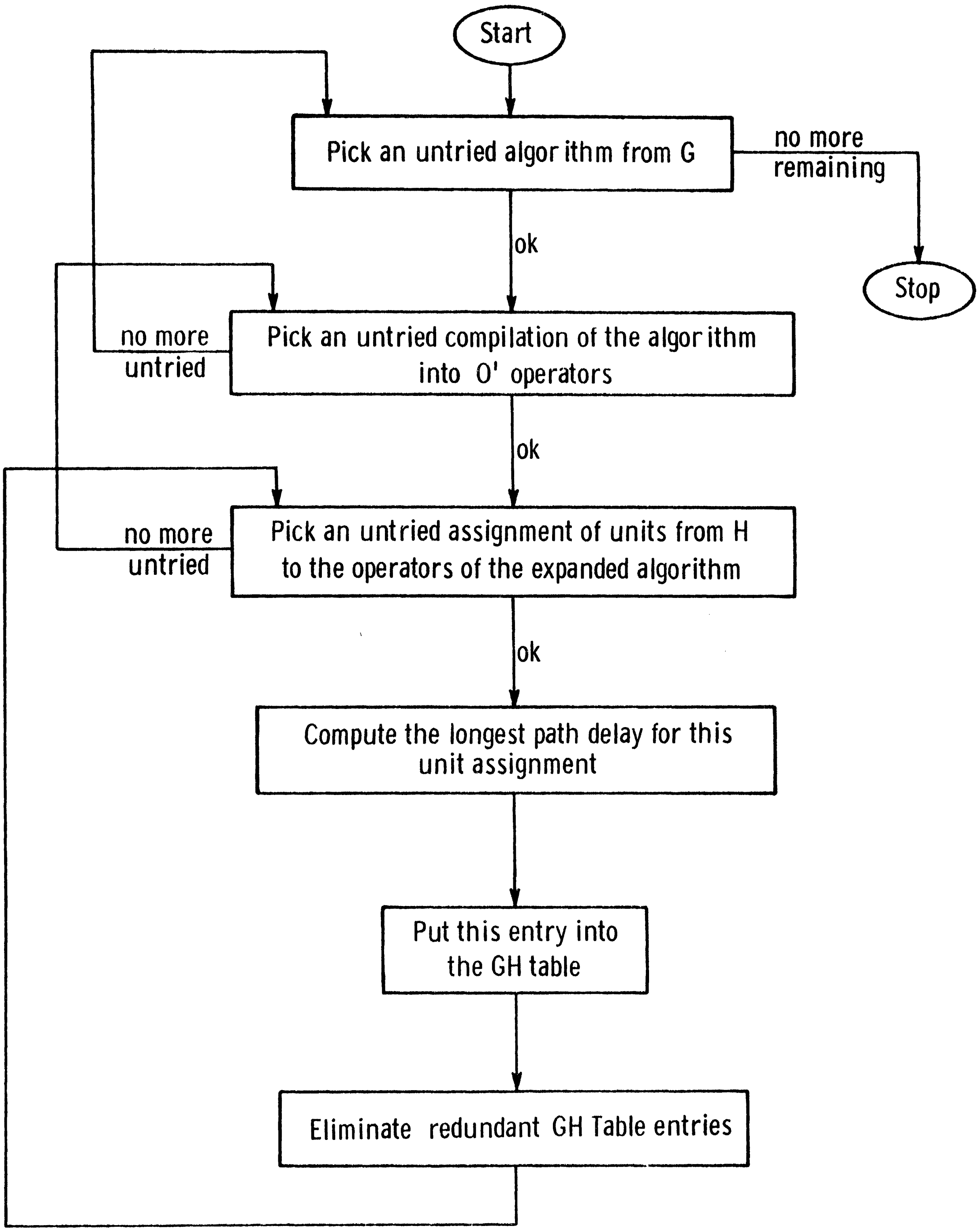

Start
Pick an untried algorithm from G
no more remaining
Stop
ok
Pick an untried compilation of the algorithm into O' operators
no more untried
ok
Pick an untried assignment of units from H to the operators of the expanded algorithm
no more untried
ok
Compute the longest path delay for this unit assignment
Put this entry into the GH table
Eliminate redundant GH Table entries

Figure 4.7 Preparation of the GH Table

The delay values and the unit set are used to select entries from the GH Table for a particular architecture. The algorithm sequence describes the compilation process used in generating the GH Table entry from the initial architecture operator. After entries have been selected for a particular architecture, the algorithm sequences for each selected entry allow the architecture instructions to be compiled in a manner compatible with the unit set that has been selected.

The GH Table must be essentially a complete enumeration of the possibilities for each operator because it is not possible to choose here between two different implementations of an operator on the basis of their individual cost and performance. This can be done only by considering the whole set of operators required by some architecture. (The single exception to this statement is considered in Section 4.4.1.4.) To illustrate this point consider an architecture operator that can be implemented by both units b and c, where c is the more costly. Now, if they have identical execution times, we might be tempted to omit the unit c implementation from the GH Table. But some specific architecture may require both this operator and a second operator which can only be performed by unit c. Had we omitted the entry for unit c for the first operator, we would now be forced to add unit b to the data path to implement the first operator rather than take advantage of unit c which must be included to cover the second operator.

Now, in Section 4.4.1.1 through 4.4.1.4, we will examine the four basic components of the GH Table generation program and then present an example of the process in Section 4.4.1.5.

4.4.1.1 Compilation of an Algorithm

This portion of the program generates the set of all possible compilations or expansions of a single algorithm into a statement group having only O' operators. If we have

n = average number of algorithms per architecture operator

j = average number of sequential substitutions of algorithms needed to compile an algorithm into O' operators

then there are

$$n^j$$

expansions for each architecture operator. This is, of course, the same as we computed in Section 4. 2. 1. Now if

$$b = |O| = \text{number of architecture operators}$$

we have

$$b \cdot n^j$$

algorithm expansions to consider for the GH Table. Comparing this to the quantity derived in Section 4. 2. 1, where we had

$$n^{mj}$$

with m being the number of operators in an architecture, we see a substantial reduction in the number of possibilities that have to be treated. The principal difference in the form of these two expressions is due to the ability to treat each expansion independently here rather than having to consider the expansion of groups of operators simultaneously. We must also note the difference between b and m:

$$b = |O|$$

$$m = \sum_{i=1}^{|\mathcal{O}|} \left| \{s_k\}_{S_{I_i}} \right| .$$

4.4.1.2 Assignment of Units

This portion of the program generates all the possible unique covering sets of units from the hardware library for a particular compiled algorithm. Every unit that can perform a particular hardware operator will be tried in turn for each occurrence of the operator in the compiled algorithm. So if there are multiple occurrences of some operator, these will be covered not only by multiple copies of the same unit but also by assigning different unit types for each occurrence. However, in one unit set covering we do not consider assigning the same unit to two nodes; each node will have a distinct unit assigned to it.

To compute the number of unit set assignments for each algorithm, let:

- a = number of different operators in a compiled algorithm
- b = average number of times each operator occurs in an algorithm
- c = average number of units in H which execute each O' operator

Then there are

$$a \cdot b$$

operators in the algorithm and

$$c^{(a \cdot b)}$$

possible assignments of units to each algorithm. The number of units for each operator, c, will vary with the completeness of H, but we can expect (a · b) to be small.

The expression derived in Section 4. 2. 2 was

$$\left[\sum_{i=1}^{b} c^{i} \right]^{a}$$

for the number of covering sets for an architecture, but it is not very meaningful to compare these two expressions. In Section 4. 2. 2, the values a and b are defined for a compiled architecture whereas here they are defined for a compiled algorithm. If we assume the compiled architecture has only one instruction, then the two expressions can have a similar interpretation: the number of unit covering sets for a single statement group. In that case the difference in the two expressions reflects the fact that in Section 4. 2. 2 we could use a unit to cover more than one statement, while in this section there are always as many units as statements. The expression

$$c^{a \cdot b} = (c^b)^a$$

is then seen to be simply the last term of the expression from Section 4. 2. 2.

4.4.1.3 Compute Longest Path Delay

The result of the compilation of an algorithm is a statement group having only O' operators. This can be represented as a noncyclic, directed graph Each node corresponds to a statement with one operator and the edges correspond to operands. When the delays to perform each operation are associated with each node according to the covering of the operations by the selected unit set, then the computation of execution time is simply the problem of finding the longest path through the graph. This is a much simpler computation than that required for a general data path where the corresponding graph may have cycles or closed loops. The computation required here has only n^2 steps for a graph having n nodes. (That is, n^2 steps for an algorithm compiled into a statement group having n statements.)

The measure of performance associated with a GH Table entry is precisely this longest path delay. The set of units used to estimate performance has no registers. Therefore, there is no need to examine alternative assignments of operands to registers, no need to modify the bus configuration to seek performance improvements, and no micro-program to compute.

4.4.1.4 Eliminate Redundant Entries

There is one criteria which should be used to eliminate some entries from the GH Table. We can state the following rule:

> For every pair entries for an operator, if the unit set of the first is contained in the unit set of the second and the delay value of the second is not smaller, then the second entry should be omitted from the table.

That is, we expect an improvement in performance if the number of copies of any unit type is increased. This rule should be compared to the statement and example given in Section 4.4.1. The distinction between these two is that we can not make a value judgment between two entries when their unit sets contain different unit types. But when both entries have all the same unit types then we do not reduce the solution space by applying the above rule. Any solution that could use the entry with the larger unit set will be explored using the entry with the smaller unit set.

In this particular program implementation we apply this rule immediately following the calculation for a new entry to the GH Table.

4.4.1.5 An Example of GH Table Generation

To clarify the concept of the GH Table, we will compute the table for the example algorithm and hardware libraries presented previously in Section 3.3 and 3.4, respectively. These are the libraries used for the example carried throughout Chapter III. Five of the seven algorithms are already in terms of hardware operators and hence need no compilation. The appropriate algorithm sequence for these will contain simply their individual algorithm number. The algorithm $g_{2,1}$ and $g_{4,2}$, however, have architecture operators in their statement group and therefore must be translated.

For $g_{2,1}$ we have

$g_{2,1}$	B ← SL2(A) :	C ← SL1(A)	00
		B ← SL1(C)	10

we apply

$g_{1,1}$, yielding

B ← SL2(A) :	C ← sl1(A)	00
	B ← SL1(C)	10

and apply $g_{1,1}$ again, yielding

B ← SL2(A) :	C ← sl1(A)	00
	B ← sl1(C)	10

which has only hardware operators in the statement group. The correct algorithm sequence is:

$$(g_{2,1}, g_{1,1}, g_{1,1}).$$

This is the only possible compilation of $g_{2,1}$.

We can translate $g_{4,2}$ similarly, but since there are two possible translations, we have two separate algorithm sequences,

$$(g_{4,2}, g_{3,1})$$

and

$$(g_{4,2}, g_{3,2}).$$

To construct the GH Table, now that we have every possible algorithmic compilation from G, we enumerate every possible assignment of hardware units to a compiled algorithm.

Table 4.1 gives a summary of the relevant data from the description of the hardware library since it is perhaps obscured there by other data. We see two units, u_2 and u_3, perform sll. So, for the statement group we derived above for $g_{2,1}$, which requires two sll operations, we will have four possible unit assignments: u_2, u_2; u_2, u_3; u_3, u_2; and u_3, u_3. The ordering of these two statements requires statement 2 to follow statement 1, thus making the delay equal to the sum of the delays for each step.

The complete GH Table for this algorithm and hardware library

Unit Number	Operations Performed	Delay
u_2	add, sll	10
u_3	comp, sll	5
u_4	sub	10

Table 4.1 Summary of example hardware unit specifications.

pair is given in Table 4.2. We see there the specific entries resulting for $g_{2,1}$. (Blanks in any column indicate the entry has not changed from the previous line.)

The Table of Figure 4.2 still contains the redundant entries (marked with an asterisk) which should be deleted according to the procedure of Section 4.4.1.4. They have been left in simply to clarify the process of GH Table generation.

4.4.2 Data Path Generation

In Section 4.4.1 we discussed how the algorithm library, G, and hardware library, H, can be manipulated to construct the GH Table. Now we will consider how the GH Table can be applied to the problem of computing an optimal data path when a specific architecture is defined. Figure 4.8 gives a flow chart of the program which performs this computation.

There are three principal steps in this process. First, the GH Table is used to select entries for the architecture operators used in the given architecture. This process yields both a unit set for the data path and an algorithm sequence for compiling each architecture operator. Next, the architecture is compiled using these algorithm sequences. Finally the compiled architecture is implemented and evaluated on the unit set. These three components are discussed separately in Section 4.4.2.1, 4.4.2.2, and 4.4.2.3, respectively.

Entry	Operator	Algorithm Sequence	Unit Set	Delay	
1	o_1	$g_{1,1}$	u_2	10	
2			u_3	5	
3	o_2	$g_{2,1}, g_{1,1}, g_{1,1}$	u_2, u_2	20	
4			u_2, u_3	15	
5			u_3, u_2	15	*
6			u_3, u_3	10	
7	o_3	$g_{3,1}$	u_2	10	
8		$g_{3,2}$	u_2	10	*
9	o_4	$g_{4,1}$	u_4	10	
10		$g_{4,2}, g_{3,1}$	u_2, u_3	15	
11		$g_{4,2}, g_{3,2}$	u_2, u_3	15	*

* - these entries are redundant

Table 4.2 An example GH Table

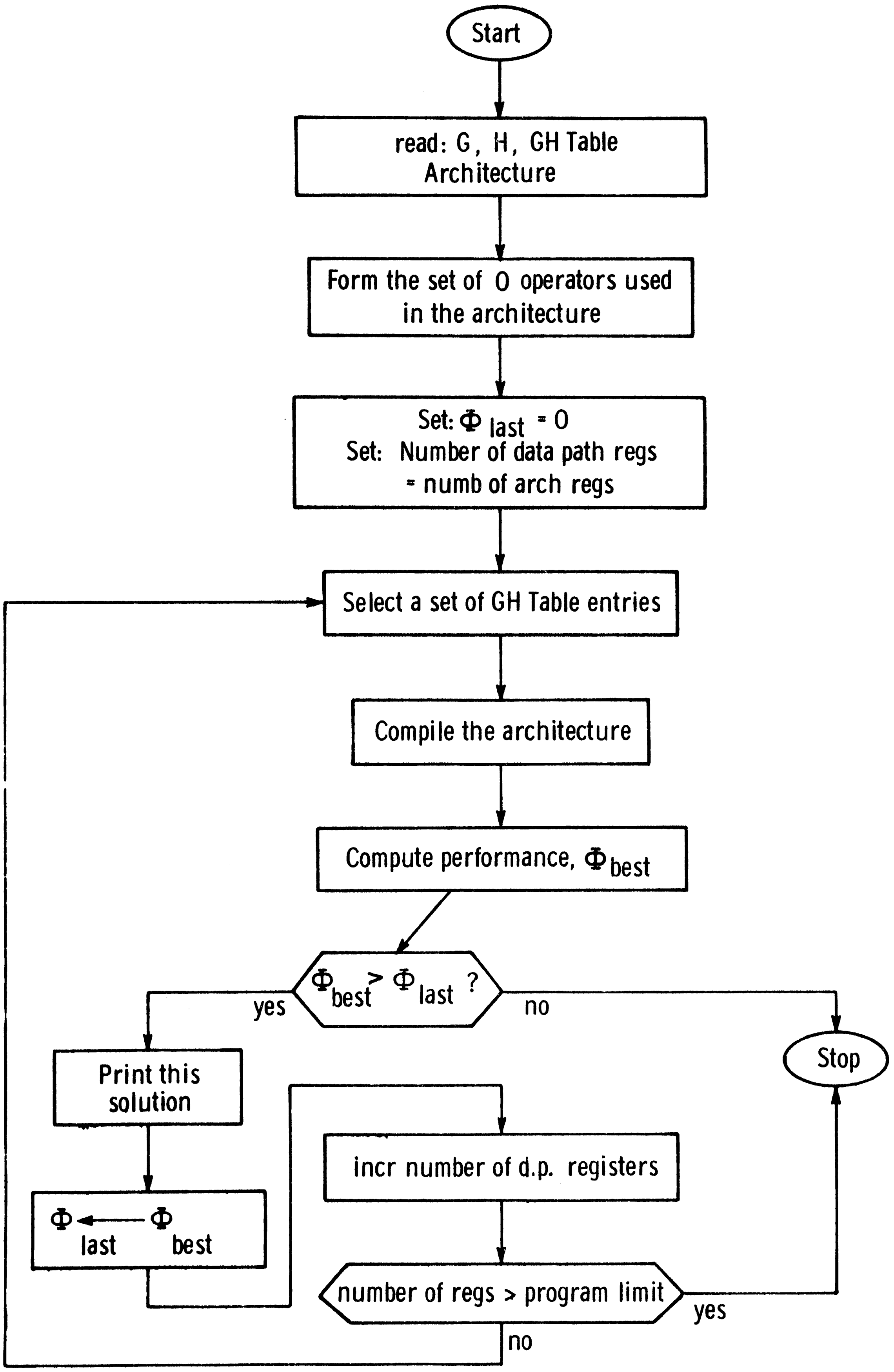

Start
read: G, H, GH Table Architecture
Form the set of O operators used in the architecture
Set: $\Phi_{last} = 0$
Set: Number of data path regs = numb of arch regs
Select a set of GH Table entries
Compile the architecture
Compute performance, Φ_{best}
$\Phi_{best} > \Phi_{last}$?
yes
no
Stop
Print this solution
$\Phi_{last} \leftarrow \Phi_{best}$
incr number of d.p. registers
number of regs > program limit
yes
no

Figure 4.8 Flow chart for data path generation

We will no longer attempt to carry through the example of Chapter III because the complexity of a hand calculation now becomes overwhelming and these sections are not sufficiently detailed to make a particular example understandable anyway.

The process just described can be considered one complete pass through the data path generation program. However, there is a complicating factor. We have not discovered any procedure to predict the optimal number of registers a data path should have. No registers were considered in the generation of GH Table entries, and the unit sets associated with the entries in the Table do not include registers. How do we determine the number of registers to be included in a data path?

We obviously need at least one hardware register for each architecture register, but there will usually be an advantage to including some temporary storage registers for intermediate results during an instruction execution. Without a cost limit we would simply put in a very large number of registers. Additional registers can not cause a loss in performance because our model does not consider fan-in and fan-out limitations, the cost of busses, or increased bus delays that would, in reality, result from increasing the number of registers. But since there is a total system cost limit, unnecessary registers may reduce the number of transformational units the data path can contain and lead to a sub-optimal data path. Starting from the other extreme and including too few registers, in order to be able to afford more transformational units within the cost limit, may result in some units being unusablė because there is no available temporary storage for their results. This again leads to a sub-optimal data path. The proper number will usually represent a balancing of these two opposing phenomenon.

Our research has not produced a solution to this problem. So in the data path generation program, we will try different numbers of registers and compare the data paths and performance for each. The process described in the beginning of this section becomes one independent computation for a fixed number of registers, and referring to Figure 4.8, we see that the entire process is simply repeated with an increasing number of registers. We start with the number of registers in the architecture and add one for each iteration until either we find that an additional register has not improved the performance or we have reached some arbitrary upper bound.

As a final point we should note that the data path generation portion of this programming system includes a number of steps which are described here as the enumeration of the values of some variable. In general these are implemented using various branch and bound techniques so as to be partially implicit rather than totally explicit enumerations. This is quite significant in reducing the execution time of the computer program but does not effect the solutions that the program finds. We feel that a complete description of any of these particular procedures would require the inclusion of an inappropriate amount of detail concerning the manner in which the program is coded, and therefore we will refer to all of these simply as enumerative searches without elaborating on their specific implementation.

4.4.2.1 Select the GH Table Entries

This component of the data path generation program selects a set of GH Table entries to translate the architecture operators required by a given architecture. The result of this selection process will be a definition of the unit set for the data path and the information required to translate the architecture.

The flow chart in Figure 4.9, with the key in Table 4.3, outlines the basic steps of this computation and we will consider this to be the primary documentation of the exact criteria used in making the GH Table entry selections. We will only summarize the overall objectives of this component of the programming system and draw attention to the most important facets of the program operation.

The goal of this program is to perform the following analysis:

ϕ_a	=	weighting factor of instruction a
OPSOCC(a, b)	=	number of occurrences of operator b in instruction a
OPSNED(b)	=	max[OPSOCC(a, b)], a = 1, . . . , $\lvert\mathscr{I}\rvert$
UNITSET	=	present selection of units from H
PUNITS(d)	=	number of copies of unit d in present unit set
BUNITS(d)	=	number of copies of unit d in best unit set
N(b)	=	number of GH Table entries for operator b
GHUNITS(b, c, d)	=	number of copies of unit d in the cth entry for operator b in the GH Table
GHX(b, c)	=	delay value for the cth entry for operator b of the GH Table
PSEL(b)	=	the entry number of the GH Table entry presently selected for operator b
BSEL(b)	=	the entry number of the GH Table entry of the best selection for operator b.

Table 4. 3 Key to terms in Figure 4. 9

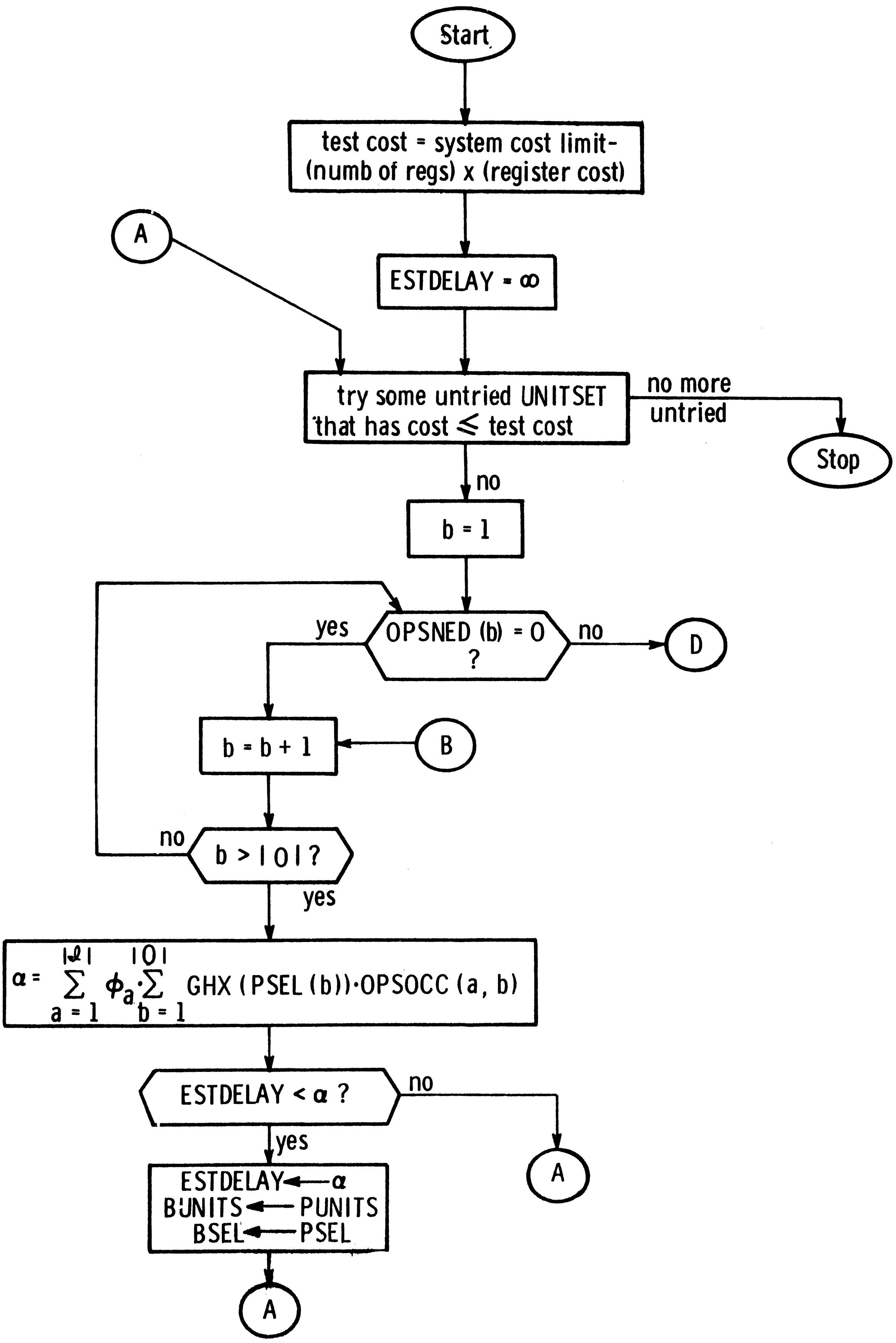


Figure 4.9 Selecting a set of GH Table entries

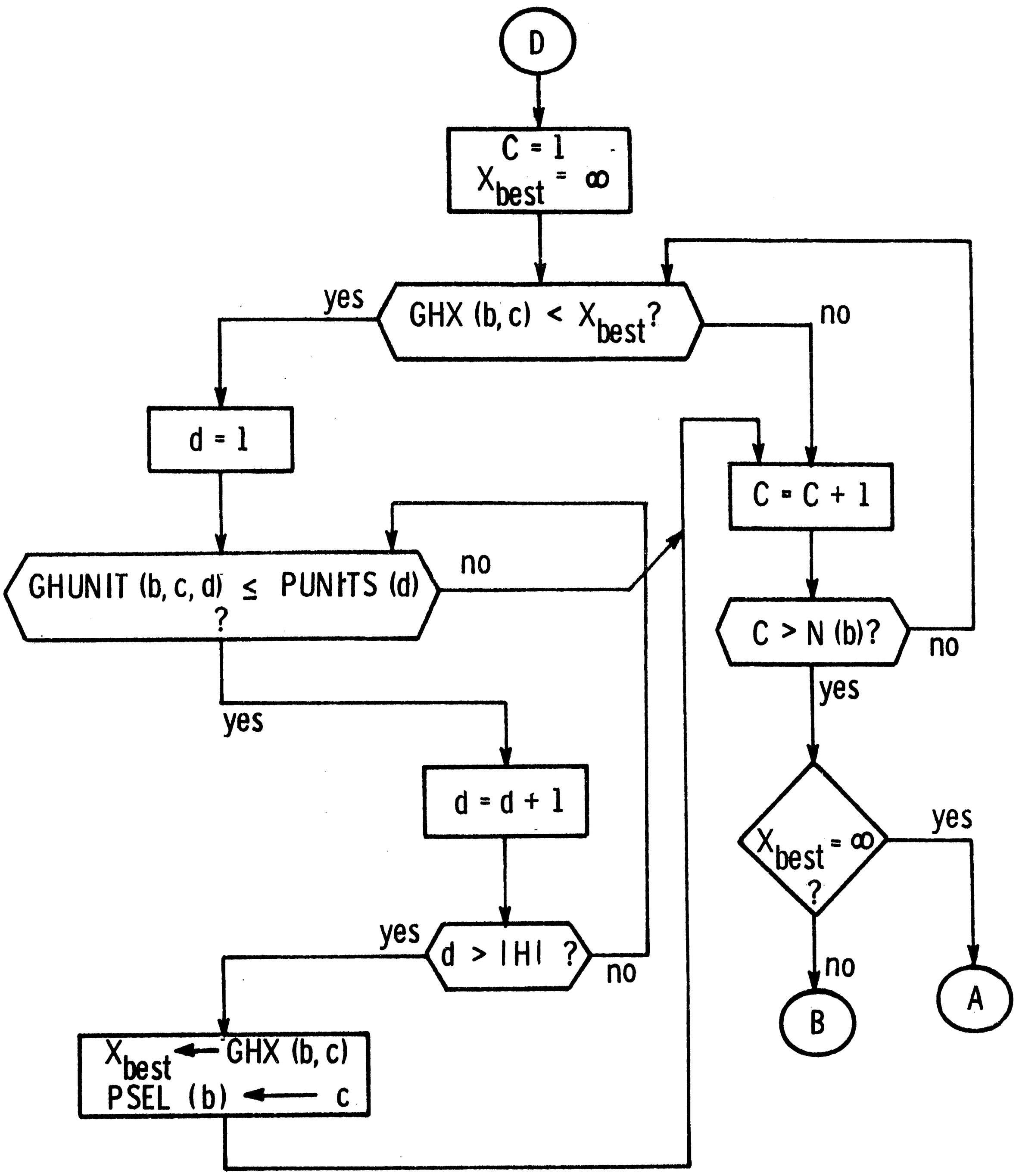


Figure 4.9 Selecting a set of GH Table entries (Continued)

Given:

(1) the system cost limit for the architecture

(2) the number of registers in the data path

(3) the frequency of occurrence of the architecture operators in each instruction

(4) the weighting factor of each instruction

Find:

(1) the number of copies of each unit that the data path should have

(2) an entry in the GH Table for each operator that occurs in the architecture

Subject to the constraints that:

(1) the cost of the selected unit set, including registers, does not exceed the systems cost limit.

(2) the estimated delay (from a weighted average computation) for the selected unit set and GH Table entries is minimized over all possible unit sets and Table entry selections.

The program proceeds by first noting the architecture operators unused by the given architecture and thereafter ignoring all portions of the GH Table associated with these operators. Next the program enumerates one of the possible data path unit sets (UNITSET) that does not exceed the system cost bound. A particular UNITSET partitions the entries of the GH Table into two groups: those whose associated unit set is contained in UNITSET and those whose unit set is not. For this UNITSET all those in the latter classification are ignored. Now for each operator, b, the GH Table entry (PSEL(b)) is selected which has the smallest delay value. These become the entries associated with the present UNITSET. Then an estimated architecture delay time is computed,

$$\text{ESTDELAY} = \sum_{a=1}^{|\mathcal{V}|} \phi_a \cdot \sum_{b=1}^{|O|} \text{GHX(PSEL}(b)) \cdot \text{OPSOCC}(a,b)$$

The program repeats this entire process with a different UNITSET. The desired result, after all UNITSET's have been tested, is the UNITSET and associated GH Table entries (the PSEL's) for which ESTDELAY was the smallest.

4.4.2.2 Compile the Architecture

This portion of the program uses the algorithms associated with the entries selected from the GH Table to compile the architecture. Recall that there is an algorithm sequence associated with each GH Table entry that describes how that operator had been compiled for that specific entry. Now this algorithm sequence is used to compile every occurrence of that operator in the architecture in exactly that same manner. This is an important step in the programming system; no other compilation procedure can be used. If the architecture were compiled by algorithm sequences other than those used in forming the selected entries, then the compiled architecture could require operators not coverable by the selected unit set.

The compilation phase of this programming system is very straight-forward. Each operator (that is, each statement) in the architecture is treated independently, and the program simply proceeds by compiling the first statement of the first instruction, then the second statement of the first statement, and so forth until the last statement of the last instruction has been done.

A point concerning compilation, which might be disconcerting, relates to the uniqueness of the compilation process. In Section 3.8.1, where the process was defined, we had

$$S_1 * g$$

as the compilation of statement group S_1 using algorithm sequence g, and noted that this will not be unique if there are multiple occurrences of an architecture operator. When we compile a single operator, we start with a statement group of one statement, but during compilation more statements may be added and an architecture operator may be duplicated. As we apply the algorithm sequences here to compile an architecture operator, however, we can insure the correct compilation by using a procedure which mimics the procedure used when the algorithm sequence was defined for the GH Table entry. That is, although there is insufficient information in g to allow $S_1 * g$ to be unique, knowing specifically how g was formulated during the GH Table generation we are able to apply the individual algorithms of g to the proper statements in S_1.

4.4.2.3 Compute the Data Path Performance

At this point in the generation of a data path we have determined both the compilation of the architecture and the unit set for the data path. We now need to determine the optimal interconnection of the units, the cycle time, and the number of cycles per instruction. Because we chose to make rather an accurate measure of performance, this component of the programming system is rather complex. An outline of it is given in the flow chart of Figure 4.10 with a key to the symbols in Table 4.4.

Solution	=	The microprogram, cycle time, longest path, the set of busses used by the microprogram, and the performance
Φ	=	$\frac{1}{\text{WAIET}}$ = performance of a solution
B	=	Best solution
P	=	Present solution
PB	=	Present Best solution
L.P.	=	Longest Path
M	=	Connection Matrix

Table 4.4 Key to terms and symbols of Figure 4.10

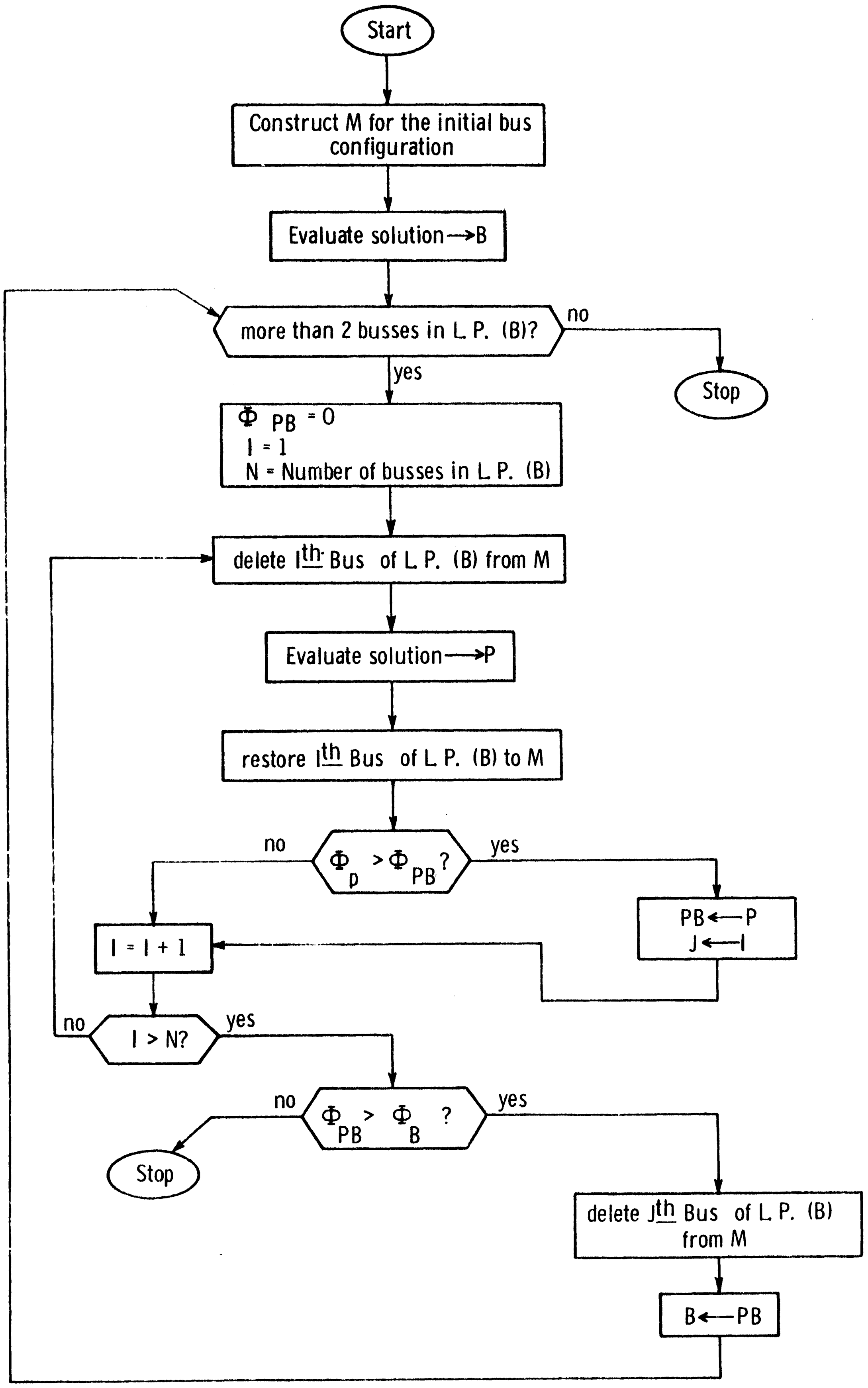

Start
Construct M for the initial bus configuration
Evaluate solution→B
more than 2 busses in L P. (B)?
no
Stop
yes
Φ PB = 0
I = 1
N = Number of busses in L P. (B)
delete Ith Bus of L P. (B) from M
Evaluate solution→P
restore Ith Bus of L P. (B) to M
no
Φp > ΦPB?
yes
PB←P
J←I
I = I + 1
no
I > N?
yes
no
ΦPB > ΦB ?
yes
Stop
delete Jth Bus of L P. (B) from M
B←PB

Figure 4.10 Flow chart to compute performance

It is important to realize that the program implementation that we have developed differs from the general solution only in the method of choosing a compilation and a unit set. The many problems of evaluation after this selection is completed are the same for both the general solution and this particular implementation. This section should therefore be treated as an extension or continuation of Section 4.2.4 rather than as a description of an alternative solution to the evaluation problem.

We have mentioned previously that since we have no cost associated with the busses, the only detrimental effect of having an extra bus is that it may contribute to an unnecessarily long cycle time. So the basic approach in optimizing the bus configuration is to start with essentially all possible busses and compute a performance; then see if an improvement can be obtained by deleting any bus that is part of the longest path. (Busses unused by the microprogram are not included.) If that is possible, we make the deletion of that bus permanent. For the new data path we can again compute a performance and a longest path and repeat the above procedure. In this program we choose to assume we have an optimal bus configuration when no improvement results from deleting any bus in the longest path. We recognize that will be only an approximation to the optimal data path since it is possible that in some cases deleting two or more busses will yield improved performance even though deleting any one bus will not. •

Once again, as in the description of the GH Table selection process, we will assume the reader, who is concerned with specific details of this part of the programming system, will find Figure 4.10 sufficient for his needs. The flow chart is considerably more accurate and complete than any reasonable verbal description could hope to be. Two steps in Figure 4.10, however, are unclear and worthy of some further elaboration. These are the formulation of the initial connection matrix and the step called "evaluating a solution." The "solution" in this context consists of: the data path cycle time, the microprogram, a list of the busses used by the microprogram, the number of cycles required by each instruction, and the performance. We end this chapter with a few notes on the functioning of these two components of the programming system.

Construct the Connection Matrix:

At this point in the programming system we have established the unit set that will be used to form the data path and compiled the architecture so that it is expressed in terms of operators that the selected unit set can perform. We need now to compute the static connection matrix for the data path so that we can perform the evaluation step of the program.

We will define the initial data path bus configuration to consist of:

(1) busses from all registers to all units.

(2) busses from all units to all registers.

(3) busses from all registers to all registers.

(4) busses which can correspond to an operand transfer in some compiled architecture instruction.

(5) but not including busses from a unit or register to itself.

This set of busses will contain any bus which could be useful in the data path.

The construction of the connection matrix can be programmmed easily from the above definition of the desired bus configuration. However, item (4) requires amplification. The busses defined by item (4) are derived by determining the set of units that can perform each statement of an instruction. Then, if the output of statement i is an input to statement j, we add a bus to the connection matrix from each unit in the covering set of i to each unit in the covering set of j.

Evaluating a Solution:

Finally there are no more variables left to adjust. We now have a compiled architecture and a well defined data path consisting of a unit set and a connection matrix. We must now compute the following:

(1) The microprogram of the compiled architecture having the minimum number of cycles on the data path.

(2) The set of busses used by the microprogram.

(3) The data path cycle time.

(4) The performance.

The microprogramming problem requires the the specification of the maps $h_1, \ldots, h_5$ and $\{h_6^a\}$ defined in Section 3.8.2, and the definition of the instruction cycles, $\psi = \langle \vec{F}, \vec{M} \rangle$, defined in Section 3.6. Since we have constructed the initial connection matrix with each register having identical connections to the remaining units, the mapping of architecture registers to hardware registers (h_1) is trivial. (That is, all possible mappings will give the same performance). Maps h_2 and h_3 for the input and output ports are similarly trivial. This allows each instruction to be microprogrammed independently.

Grossly oversimplifying, we compute the microprogram with the minimal number of cycles by constructing a list for each instruction of all possible data path states that can be reached in one cycle. We define a cycle here, just as in Chapter III, as a dynamic function set and a dynamic connection matrix, and define a state by the set of instruction statements remaining to be done and the particular assignment of operands to registers. Then we compute the set of states that can be reached after one additional cycle, and so forth, until we enumerate the final state. This is the state of no statements remaining to be done and the required operands in the appropriate registers. From this list of states we can construct the sequence of cycles from the initial state to the final state and having the fewest cycles. This gives us the desired

$$m_i = \text{minimum number of cycles for instruction } I_i'$$

and

$$I' = (\psi_1, \psi_2, \ldots \psi_{m_i}).$$

Since we use a constructional approach for finding I_a' for I_a^c, the map h_4 is trivial. The computation for the sequence of cycles described above is essentially a determination of the remaining maps h_5 and h_6^a for instruction I_a^c.

All of the busses in the connection matrix are available to the microprogramming computation, but a record is made of those that are actually used in some instruction. After all of the microprograms are calculated, we determine the cycle time of the data path. As explained in Section 4.2.4.1 this is a simple "longest path" computation utilizing the delay values in the units and the set of busses used by the microprogram (not the connection matrix). We must allow for possible loops in the data path and thus avoid counting any unit more than once in computing path delays. At the same time the cycle time is computed, a list of the busses that form the longest path through the data path is generated. (Or one of the longest paths if there are more than one with equal length.)

The final step is to verify that each instruction has been implemented within the specified maximum time limit and then compute the performance. Both of these steps are first mentioned in Section 3.7.2. For

τ_i = maximum permitted execution time for instruction i.

ϕ_i = weighting factor for instruction i.

we first determine that

$$\tau_i \geq m_i \cdot T \qquad \text{for } i = 1, \ldots, |\mathcal{A}'|$$

where we have just computed

m_i = number of cycles for instruction i.

T = data path cycle time.

If all instructions have been implemented within the allowable time, then we compute performance

$$\Phi = \frac{1}{\text{WAIET}}$$

with

$$\text{WAIET} = T \cdot \sum_{i=1}^{|\mathcal{A}|} m_i \cdot \phi_i$$

CHAPTER V

EXAMPLE PROBLEMS

In this chapter we present five examples which will serve to emphasize the major points we have covered in the previous chapters. We hope to clarify concepts that may have been cloudy or obscure when discussed previously in abstract terms. Also we want to emphasize the principal steps in the design process and illustrate the various optimization decisions that this procedure considers. Finally these examples allow insight into the nature of the programming system that has been developed since all the results in this chapter are based on the computer solution.

The first four examples each treat the issue of making a choice for a particular problem variable. We demonstrate this process by presenting two or three parts in each example which differ only in the particular variable of interest so that we can easily identify the effect of this change on the solution. In order to hold extraneous information at a minimum, these examples are kept as simple as possible. Variables which are not of interest in a particular example are usually constrained to a single value.

In the first example we examine the process of selecting units from the hardware library. The second example demonstrates the algorithm selection and compilation phase of the solution. The third example shows how the search for maximum performance will result in different data paths as the instruction weighting factors are varied. Finally in the fourth example we see how the performance can vary as the system cost limit is changed.

The fifth example relates this research to the real world by demonstrating how portions of a real computer might be described in this model. The first four examples use abstract operators and instructions that may not have direct counterparts in existing computer systems. Here we describe and then design a processor having more easily recognizable operations and instructions.

The programming system developed during this research is written in FORTRAN and the S/360 Assembler Language for execution on an IBM Model 67, and all the examples in this chapter were run on this program. The descriptions of these examples follow the formats used throughout the earlier chapters. The actual listings for the input and output for each example are reproduced in an appendix. These are included primarily for completeness and to illustrate the magnitude and form of the data required to completely describe the particular examples. The computer listings will probably appear quite cryptic when compared to the discussion of the examples in this chapter.

In Table 5.1 we list the execution time (CPU time) required to

Example	Required CPU Time (secs)
1, Case 1	0.967
1, Case 2	0.922
2, Case 1	0.747
2, Case 2	0.764
2, Case 3	1.468
3, Case 1	0.770
3, Case 2	0.857
4, Case 1	0.459
4, Case 2	0.744
4, Case 3	1.027
5	4.151

Table 5.1 Execution times of the five examples

perform the data path design in each of the cases of the five examples in this chapter. The first four examples are quite simple and require very little time. The last example, which is the most elaborate run by the program, still requires less than five seconds. This is very reassuring since it indicates that much larger problems can be attempted and refinements added to the program. The times listed in Table 5.1 do not include the time required to generate the GH Table, but this is insignificant compared to the Data Path Generation.

We will use the same names for the architecture and hardware operators throughout the first four examples and therefore will declare these once rather than in each of the individual examples. They are listed in Table 5.2 with the number of input parameters for each operator. There is no interpretation to associate with these operators yet. This will be done later by supplying an algorithm library in each example.

Architecture Operators

Name		Number of input parameters
AOP0	(o_0)	1
AOP1		1
AOP2		1
AOP3		1
AOP4		1
AOP5		2

Hardware Operators

Name		Number of input parameters
HOP0	(o'_0)	1
HOP1		1
HOP2		1
HOP3		1
HOP4		1
HOP5		2

Table 5.2 Operators for the examples of Section 5.1 through 5.4

5.1 AN EXAMPLE OF UNIT SELECTION

This example has two parts: Case 1 and Case 2. In both cases the algorithm library and architecture remain unchanged. The only difference in the hardware library is that the delay value for one unit is reduced in Case 2.

This example requires only the single architecture operator AOP1 and we have just one algorithm which relates it directly to HOP1 and HOP2. So we have,

$$G = \{g_{1,1}\}$$

with

$g_{1,1}$	$A \leftarrow \mathrm{AOP1}(B)$:	$C \leftarrow \mathrm{HOP1}(B)$	00
		$A \leftarrow \mathrm{HOP2}(C)$	10

For the hardware library there is one unit which is a register (it performs the move operator, HOP0), one unit which does the two hardware operators HOP1 and HOP2, and a third unit capable only of HOP3. Table 5.3 gives the specifications for the hardware library. We have indicated the delay value for unit u_3 as δ_{u_3} and for the two cases in this example we will let

Case 1	$\delta_{u_3} = 12$
Case 2	$\delta_{u_3} = 8$

The GH Tables for this algorithm and hardware library will have two entries for AOP1 and will differ only in the delay associated with the second entry. The GH Tables are given in Table 5.4 and Table 5.5 for Case 1 and Case 2, respectively.

The architecture, which is specified in Table 5.6, has one instruction and this instruction has one statement. We can quickly see the only possible compilation will yield a statement group having an HOP1 operator followed by an HOP2 operator.

It is interesting to compare the performance of three possible data paths which can implement this architecture using the given hardware library. We should point out however that the choice of an optimal data path has not been simply a one out of three choice. Even in this simple case it is nearly impossible to estimate the solution space from which these three solutions have been selected. To begin with, there are 132 cost feasible unit sets, many of which could yield performance as good as the selected solutions, but none which can have better performance. The three selected for discussion are in a sense minimal "best" solutions in that all other data paths of equal performance contain unused units or busses.

The solutions that the program finds are given in Figure 5.1 and Table 5.7 for Case 1 and Figure 5.2 and Table 5.8 for Case 2. We present there the data path, cycle time computation, instruction implementation, and the computation of WAIET.

In Figure 5.1 where the cycle time computation is given, we see that one times the register delay has been included in the cycle time. In all of the previous examples the delay for registers has been zero but in this example (and throughout the remainder of Chapter V) we will follow the convention of assigning a non-zero delay to the register unit and adding this into the cycle time. Part of this delay perhaps is associated with register out-gating and part with in-gating. Since all paths through the data path must start with register out-gating and end with register in-gating, it is correct to include the whole register delay once in the computation.

The third solution we consider would use only a single copy of unit u_2 and require two cycles for the instruction. This, though, has a larger WAIET than the two solutions found by the program and would be selected only if the system cost limit were lowered.

We can see from Figure 5.2 that the selected data path does incorporate unit u_3 after its delay was reduced. We can quickly verify that in both Cases 1 and 2 the selected solution is in fact optimal and that the other two data paths that were rejected by the program have greater WAIET. We summarize these results in Table 5.9.

$$H = \{u_1, u_2, u_3\}$$

For unit u_1:

Cost = 2 Delay = 2 Number of Functions = 2

$f_{1,1}$ $\overline{r1} \leftarrow$ HOP0(q1)

$f_{1,2}$ $p1 \leftarrow$ HOP0(r1)

For unit u_2:

Cost = 10 Delay = 10 Number of Functions = 2

$f_{2,1}$ $p1 \leftarrow$ HOP1(q1)

$f_{2,2}$ $p1 \leftarrow$ HOP2(q1)

For unit u_3:

Cost = 15 Delay = δ_{u_3} Number of Functions = 1

$f_{3,1}$ $p1 \leftarrow$ HOP2(q1)

Table 5.3 Hardware library for Example 5.1, Case 1

Entry	Operator	Algorithm Sequence	Unit Set	Delay
1	AOP1	$g_{1,1}$	u_2, u_2	20
2	AOP1	$g_{1,1}$	u_2, u_3	22

Table 5.4 The GH Table for Example 5.1, Case 1

Entry	Operator	Algorithm Sequence	Unit Set	Delay
1	AOP1	$g_{1,1}$	u_2, u_2	20
2	AOP1	$g_{1,1}$	u_2, u_3	18

Table 5.5 The GH Table for Example 5.1, Case 2

Number of Registers	=	1
Number of Instructions	=	1
System Cost Limit	=	50

Instruction 1 :

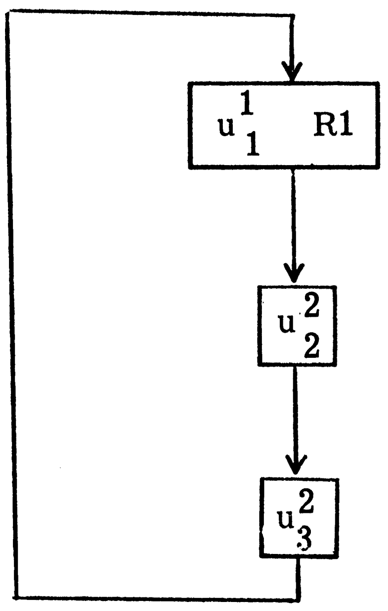
max execution time	=	90
weighting factor	=	1

statement group:

$$\overline{R1} \leftarrow AOP1(R1)$$

Table 5. 6 Architecture for Example 5. 1

Data Path:



Unit Number	Unit Type	Operator	Cost	Delay
1	1	HOP0	2	2
2	2	HOP1,HOP2	10	10
3	2	HOP1,HOP2	10	10

Data path cycle time:

$$T = 2 + 10 + 10 = 22$$

Data path cost:

$$C' = 2 + 10 + 10 = 22$$

Figure 5.1 Data path for Example 5.1, Case 1

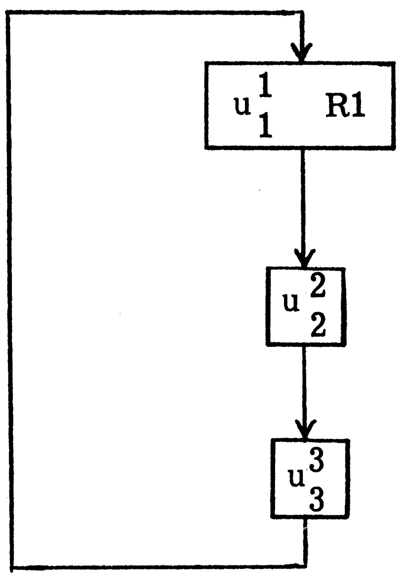
Instruction: $I' = (\psi_1)$

ψ_1:	$p1_2$	$\leftarrow$ HOP1(R1)	000
	$q1_3$	$\leftarrow$ HOP0($p1_2$)	100
	$\overline{R1}$	$\leftarrow$ HOP2($q1_3$)	110

$$\text{WAIET} = T \cdot |I'| = 22 \cdot 1 = 22$$

Table 5.7 Instruction implementation for Example 5.1, Case 1

Data Path:



Unit Number	Unit Type	Operator	Cost	Delay
1	1	HOP0	2	2
2	2	HOP1, HOP2	10	10
3	3	HOP2	15	8

Data path cycle time:

$$T = 2 + 10 + 8 = 20$$

Data path cost:

$$C' = 2 + 10 + 15 = 27$$

Figure 5.2 Data path for Example 5.1, Case 2

Instruction: $I' = (\psi_1)$

ψ_1 :	$p1_2 \leftarrow$	HOP1(R1)	000
	$q1_3 \leftarrow$	HOP0($p1_2$)	100
	$\overline{R1} \leftarrow$	HOP2($q1_3$)	110

$$\text{WAIET} = T \cdot |I'| = 20 \cdot 1 = 20$$

Table 5.8 Instruction implementation for Example 5.1, Case 2

Case 1

Solution Unit Set	Cycle Time	Number of Cycles	WAIET
u_2	12	2	24
u_2, u_2	22	1	22 *
u_2, u_3	24	1	24

Case 2

Solution Unit Set	Cycle Time	Number of Cycles	WAIET
u_2	12	2	24
u_2, u_2	22	1	22
u_2, u_3	20	1	20 *

* - the solution selected by the program

Table 5.9 Comparison of the three solutions of Example 5.1

5.2 AN EXAMPLE OF COMPILATION

In this example we illustrate the selection of algorithms and the compilation of an architecture. We see how increases in the available architecture operators can permit more convenient expression of the desired instruction and how enlargement of the algorithm library may lead to an improved data path solution. The hardware library is the same for all three cases of this example, but the architecture and algorithm library are changed.

We first define an algorithm library which has one entry for each of the three operators AOP1, AOP2, and AOP3. This is shown in Table 5.10.

For a hardware library there are just four units with each unit executing a single hardware operator. This is specified in Table 5.11.

Table 5.12 is the GH Table which results from this algorithm, hardware library pair. It is predictably short and simple with a single entry for each operator.

Now we define an architecture which has one instruction. We will let this instruction be described as the serial execution of the three architecture operators AOP1, AOP2, and AOP3. We present this in Table 5.13.

The solution found for this combination of algorithm, hardware library and architecture is illustrated in Figure 5.3 and Table 5.14. This is a rather obvious solution having the three transformational units in a serial connection so that only one cycle is required. There are four reasonable solutions for this problem. The other three, although using the same units, delete busses between the transformational units and inset busses to and from the register. This allows a shorter cycle time but requires additional cycles. We will return to this question in Section 5.3, but it is probably obvious in this example that these three other solutions all have reduced performance.

Now suppose we choose to introduce a new architecture operator, AOP4, and define it in the algorithm library as being the serial execution of the three architecture operators we used to describe the instruction of Case 1. We can do this by adding one algorithm to the library. The new algorithm library is given in Table 5.15 and the new corresponding GH Table is given in Table 5.16. The entry in the GH Table for AOP4 indicates by its algorithm sequence that three translations are required to compile AOP4 into a statement group using only hardware operators.

With the new architecture operator it is possible to express precisely the same architecture that we had in Case 1 in a more condensed form by recognizing that AOP4 does everything that we want the instruction to do. So we change the architecture of Case 1 by altering the instruction definition. The new architecture is given in Table 5.17.

When we re-run this problem, the program uses the new entry in the GH Table to compile the instruction. This produces a compiled architecture identical to that of Case 1 so that after compilation the processing is identical to Case 1. The solution is therefore identical to that of Case 1 which was given in Figure 5.3 and Table 5.14. The new operator introduced an alternative means of describing the desired architecture but the selected data path and instruction implementation is, as we would expect, unchanged.

Now building on Case 2, when a new method of implementing AOP4 is discovered, this will be reflected in this model by adding the appropriate new algorithm to the library. Suppose we now feel that AOP4 can be performed by two sequential AOP1's followed by an AOP2. We will make this algorithm the second entry for AOP4 and have the algorithm library displayed in Table 5.18.

The GH Table for this new library is given in Table 5.19. As we would expect there are now two entries for AOP4.

Now for Case 3 we can ask the program to find an optimal data path for the architecture of Case 2 using the new algorithm library and GH Table. The program still has access to the previous solution but now recognizes the advantage of using the newest GH Table entry. The delay estimate for the first entry is greater than that of the newer entry for AOP4 and therefore the latter entry is selected. The data path resulting from this selection is given in Figure 5.4 and Table 5.20.

Comparing this to the previous solution for Case 1 and 2 (Figure 5.3 and Table 5.14), we see there has been a performance improvement due to the decrease in cycle time which the new unit set permits. The data path for Case 3 still has the same serial organization as in Cases 1 and 2.

$$G = \{g_{1,1}, g_{2,1}, g_{3,1}\}$$

$g_{1,1}$	$A \leftarrow AOP1(B)$	:	$A \leftarrow HOP1(B)$
$g_{2,1}$	$A \leftarrow AOP2(B)$	:	$A \leftarrow HOP2(B)$
$g_{3,1}$	$A \leftarrow AOP3(B)$	:	$A \leftarrow HOP3(B)$

Table 5.10 Algorithm library for Example 5.2, Case 1

$$H = \{ u_1, u_2, u_3, u_4 \}$$

for unit u_1:

	Cost = 2	Delay = 2	Number of Functions = 2
$f_{1,1}$	$\overline{r1} \leftarrow$ HOP0(q1)		
$f_{1,2}$	p1 $\leftarrow$ HOP0(r1)		

for unit u_2:

	Cost = 10	Delay = 10	Number of Functions = 1
$f_{2,1}$	p1 $\leftarrow$ HOP1(q1)		

for unit u_3:

	Cost = 15	Delay = 15	Number of Functions = 1
$f_{3,1}$	p1 $\leftarrow$ HOP2(q1)		

for unit u_4:

	Cost = 20	Delay = 20	Number of Functions = 1
$f_{4,1}$	p1 $\leftarrow$ HOP3(q1)		

Table 5.11 Hardware library for Example 5.2

Entry	Operator	Algorithm Sequence	Unit Set	Delay
1	AOP1	$g_{1,1}$	u_2	10
2	AOP2	$g_{2,1}$	u_3	15
3	AOP3	$g_{3,1}$	u_4	20

Table 5.12 GH Table for Example 5. 2, Case 1

Number of Registers	=	1
Number of Instructions	=	1
System Cost Limit	=	50

Instruction 1:

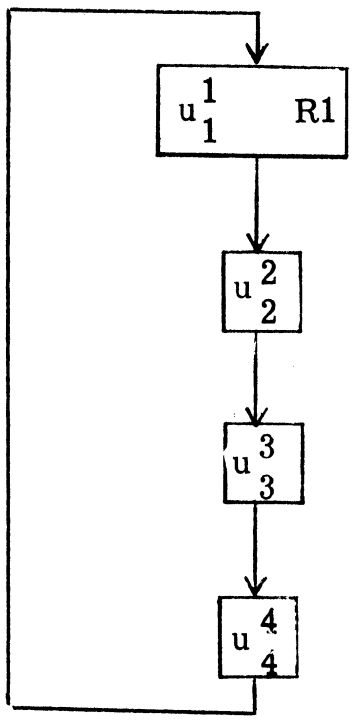
max execution time	=	90
weighting factor	=	1

statement group:

$A \leftarrow AOP1(R1)$	000
$B \leftarrow AOP2(A)$	100
$\overline{R1} \leftarrow AOP3(B)$	110

Table 5.13 Architecture for Example 5.2, Case 1

Data Path:



Unit Number	Unit Type	Operator	Cost	Delay
1	1	HOP0	2	2
2	2	HOP1	10	10
3	3	HOP2	15	15
4	4	HOP3	20	20

Data path cycle time:

$$T = 2 + 10 + 15 + 20 = 47$$

Data path cost:

$$C' = 2 + 10 + 15 + 20 = 47$$

Figure 5.3 Data path for Example 5.2, Cases 1 and 2

Instruction I' : $I' = (\psi_1)$

ψ_1:	$p1_2 \leftarrow HOP1(R1)$	00000
	$q1_3 \leftarrow HOP0(p1_2)$	10000
	$p1_3 \leftarrow HOP2(q1_3)$	11000
	$q1_4 \leftarrow HOP0(p1_3)$	11100
	$\overline{R1} \leftarrow HOP3(q1_4)$	11110

$$WAIET = T \cdot |I'| = 47 \cdot 1 = 47$$

Table 5.14 Instruction implementation for Example 5.2, Cases 1 and 2

$$G = \{g_{1,1}, g_{2,1}, g_{3,1}, g_{4,1}\}$$

$g_{1,1}$	A ← AOP1(B)	:	A ← HOP1(B)	
$g_{2,1}$	A ← AOP2(B)	:	A ← HOP2(B)	
$g_{3,1}$	A ← AOP3(B)	:	A ← HOP3(B)	
$g_{4,1}$	A ← AOP4(B)	:	C ← AOP1(B)	000
			D ← AOP2(C)	100
			A ← AOP3(D)	110

Table 5.15 Algorithm library for Example 5.2, Case 2

Entry	Operator	Algorithm Sequence	Unit Set	Delay
1	AOP1	$g_{1,1}$	u_2	10
2	AOP2	$g_{2,1}$	u_3	15
3	AOP3	$g_{3,1}$	u_4	20
4	AOP4	$g_{4,1}, g_{1,1}, g_{2,1}, g_{3,1}$	u_2, u_3, u_4	45

Table 5.16 GH Table for Example 5.2, Case 2

Number of Registers	=	1
Number of Instructions	=	1
System Cost Limit	=	50

Instruction 1:

max execution time	=	90
weighting factor	=	1

statement group:

$$\overline{R1} \leftarrow AOP4(R1)$$

Table 5.17 Architecture for Example 5.2, Case 2

$$G = \{g_{1,1}, g_{2,1}, g_{3,1}, g_{4,1}, g_{4,2}\}$$

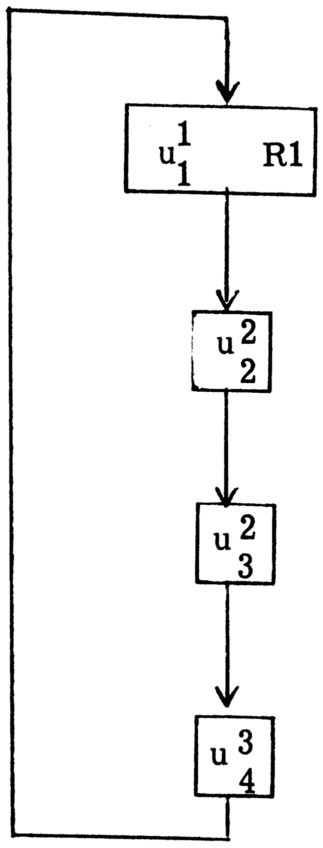
$g_{1,1}$	$A \leftarrow AOP1(B)$	:	$A \leftarrow HOP1(B)$	
$g_{2,1}$	$A \leftarrow AOP2(B)$	:	$A \leftarrow HOP2(B)$	
$g_{3,1}$	$A \leftarrow AOP3(B)$	:	$A \leftarrow HOP3(B)$	
$g_{4,1}$	$A \leftarrow AOP4(B)$	:	$C \leftarrow AOP1(B)$	000
			$D \leftarrow AOP2(C)$	100
			$A \leftarrow AOP3(D)$	110
$g_{4,2}$	$A \leftarrow AOP4(B)$	:	$C \leftarrow AOP1(B)$	000
			$D \leftarrow AOP1(C)$	100
			$A \leftarrow AOP2(D)$	110

Table 5.18 Algorithm library for Example 5.2, Case 3

Entry	Operator	Algorithm Sequence	Unit Set	Delay
1	AOP1	$g_{1,1}$	u_2	10
2	AOP2	$g_{2,1}$	u_3	15
3	AOP3	$g_{3,1}$	u_4	20
4	AOP4	$g_{4,1}, g_{1,1}, g_{2,1}, g_{3,1}$	u_2, u_3, u_4	45
5	AOP4	$g_{4,2}, g_{1,1}, g_{1,1}, g_{2,1}$	u_2, u_2, u_3	35

Table 5.19 GH Table for Example 5.2, Case 3

Data Path:



Unit Number	Unit Type	Operator	Cost	Delay
1	1	HOP0	2	2
2	2	HOP1	10	10
3	2	HOP1	10	10
4	3	HOP2	15	15

Data path cycle time:

$$T = 2 + 10 + 10 + 15 = 37$$

Data path cost:

$$C' = 2 + 10 + 10 + 15 = 37$$

Figure 5.4 Data path for Example 5.2, Case 3

Instruction I': $I' = (\psi_1)$

ψ_1:	$p1_2 \leftarrow \text{HOP1}(R1)$	00000
	$q1_3 \leftarrow \text{HOP0}(p1_2)$	10000
	$p1_3 \leftarrow \text{HOP1}(q1_3)$	11000
	$q1_4 \leftarrow \text{HOP0}(p1_3)$	11100
	$\overline{R1} \leftarrow \text{HOP2}(q1_4)$	11110

$$\text{WAIET} = T \cdot |I'| = 37 \cdot 1 = 37$$

Table 5.20 Instruction implementation for Example 5.2, Case 3

5.3 AN EXAMPLE WITH WEIGHTING FACTOR VARIATION

We come now to an example that is probably more interesting then those covered so far. In this example we see how the program will alter the selected solution so that performance is optimized around the particular architecture parameters. Specifically we will have our usual, simple algorithm and hardware libraries, but this time we have an architecture with two instructions and observe the effect on the solution as the weighting factors are varied. This example is an illustration of the procedure described in section 4.2.3.

The algorithm library is presented in Table 5.21 and gives translations for AOP1, AOP2, and AOP3. The hardware library appears in Table 5.22 and has four units to perform the four hardware operators HOP0, HOP1, HOP2, and HOP3. The two libraries are the same for both cases of this example and the corresponding GH Table is given in Table 5.23.

We specify the architecture in Table 5.24. This is almost the same for Case 1 and Case 2. We define it only once using the symbols x and y to designate the instruction weighting factors which are the only items that change in this example. We have included two registers in this example to introduce a bit of variety from the previous two examples. This change has no special significance.

For the first case, let

$$x = 8$$

$$y = 2$$

to indicate that instruction I1 is four times as important as instruction I2. The program solution for these weighting factors is illustrated in Figure 5.5 and Table 5.25. The data path cycle time is determined by the delay through the serial connection of units u_2 and u_3 since this is greater than the delay of unit u_5. Both instructions can be performed in one cycle and we see that

WAIET = 270 for Case 1.

At first glance it is surprising to note that no bus exists to take data out of register R2, but this is a natural outcome of the fact that there is no instruction in the architecture which requires the data of R2. Therefore the program did not include an output bus for this unit.

Now, for Case 2, we shift the emphasis to instruction I2 by reversing the weighting factors so that

$$x = 2$$

$$y = 8$$

The new solution appears in Figure 5.6 and Table 5.26. The unit set is the same as in Case 1. For this given architecture and pair of libraries, no other unit set could reasonably be considered. But the program has responded to the new weighting factors by altering the bus configuration . Units u_3 and u_4 are no longer connected so that instruction I1 must now be done in two cycles. However, the more important instruction, I2, still requires just one cycle, and the cycle time is now shorter. We have

$$\text{WAIET} = 264 \qquad \text{for Case 2} .$$

As an interesting comparison we can compute by hand the WAIET for some solutions which the program has rejected for these two cases. For this unit set there are more possible bus configurations than the two found for Case 1 and Case 2. Example 5. 2 demonstrated another possibility. However, all configurations besides the two already evaluated above include busses that do not correspond to data transfers for this particular architecture. For instance, the data path which was the optimal solution for Example 5. 2 would contain a bus from unit u_4 to u_5 but there is never an occasion in this architecture to use that bus. Therefore we know that all bus configurations besides the two described above can not offer an improvement in performance. We evaluate each of the two architectures on the data path that was rejected for that particular architecture as follows:

For Case 1:

$$x = 8$$

$$y = 2$$

$$T = 22 \quad \text{(for the Case 2 data path)}$$

$$|I1'| = 2$$

$$|I2'| = 1$$

$$\text{WAIET} = (8 \cdot 2 + 2 \cdot 1) \cdot 22 = 396$$

For Case 2:

$$x = 2$$

$$y = 8$$

$$T = 25 \quad \text{(for the Case 1 data path)}$$

$$|I1'| = 1$$

$$|I2'| = 1$$

$$\text{WAIET} = (2 \cdot 1 + 8 \cdot 1) \cdot 27 = 270$$

These results are summarized in Table 5.27 where we can see that for both architectures the selected solution is optimal.

$$G = \{g_{1,1}, g_{2,1}, g_{3,1}\}$$

$g_{1,1}$	A ← AOP1(B)	:	A ← HOP1(B)
$g_{2,1}$	A ← AOP2(B)	:	A ← HOP2(B)
$g_{3,1}$	A ← AOP3(B)	:	A ← HOP3(B)

Table 5.21 Algorithm library for Example 5.3

$$H = \{u_1, u_2, u_3, u_4\}$$

for unit u_1:

Cost = 2 Delay = 2 Number of Functions = 1

$f_{1,1}$	$\overline{r1} \leftarrow$ HOP0(q1)
$f_{1,2}$	p1 $\leftarrow$ HOP0(r1)

for unit u_2:

Cost = 10 Delay = 10 Number of Functions = 1

$f_{2,1}$	p1 $\leftarrow$ HOP1(q1)

for unit u_3:

Cost = 15 Delay = 15 Number of Functions = 1

$f_{3,1}$	p1 $\leftarrow$ HOP2(q1)

for unit u_4:

Cost = 20 Delay = 20 Number of Functions = 1

$f_{4,1}$	p1 $\leftarrow$ HOP3(q1)

Table 5.22 Hardware library for Example 5.3

Entry	Operator	Algorithm Sequence	Unit Set	Delay
1	AOP1	$g_{1,1}$	u_2	10
2	AOP2	$g_{2,1}$	u_3	15
3	AOP3	$g_{3,1}$	u_4	20

Table 5.23 GH Table for Example 5.3

Number of Registers	=	2
Number of Instructions	=	2
System Cost Limit	=	50

Instruction I1:

max execution time	=	90
weighting factor	=	x

statement group:

$A \leftarrow AOP1(R1)$	00
$\overline{R1} \leftarrow AOP2(A)$	10

Instruction I2:

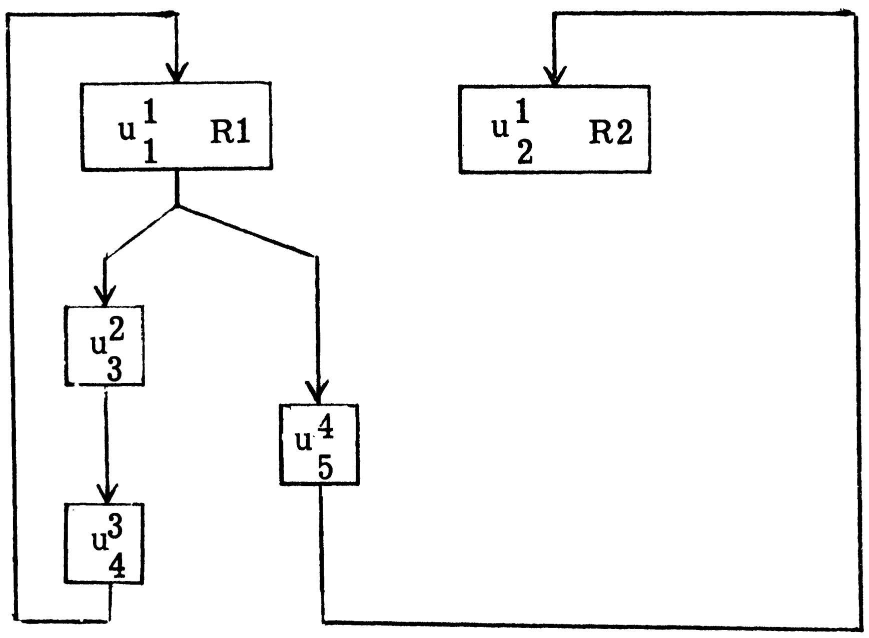
max execution time	=	90
weighting factor	=	y

statement group:

$\overline{R2} \leftarrow AOP3(R1)$

Table 5.24 Architecture for Example 5.3

Date Path:



Unit Number	Unit Type	Operator	Cost	Delay
1	1	HOP0	2	2
2	1	HOP0	2	2
3	2	HOP1	10	10
4	3	HOP3	15	15
5	4	HOP4	20	20

Data path cycle time:

$$T = \max\,[\,(\delta_{u_1} + \delta_{u_2} + \delta_{u_3}),(\delta_{u_1} + \delta_{u_5})]$$

$$= \max\,[\,27,\ 22] \ = \ 27$$

Data path cost:

$$C' = 2 + 2 + 10 + 15 + 20 \ = \ 49$$

Figure 5.5 Data path for Example 5.3, Case 1

Number of Registers	=	2
Number of Instructions	=	2
System Cost Limit	=	50

Instruction I1:

max execution time	=	90
weighting factor	=	x
statement group:		

$A \leftarrow AOP1(R1)$	00
$\overline{R1} \leftarrow AOP2(A)$	10

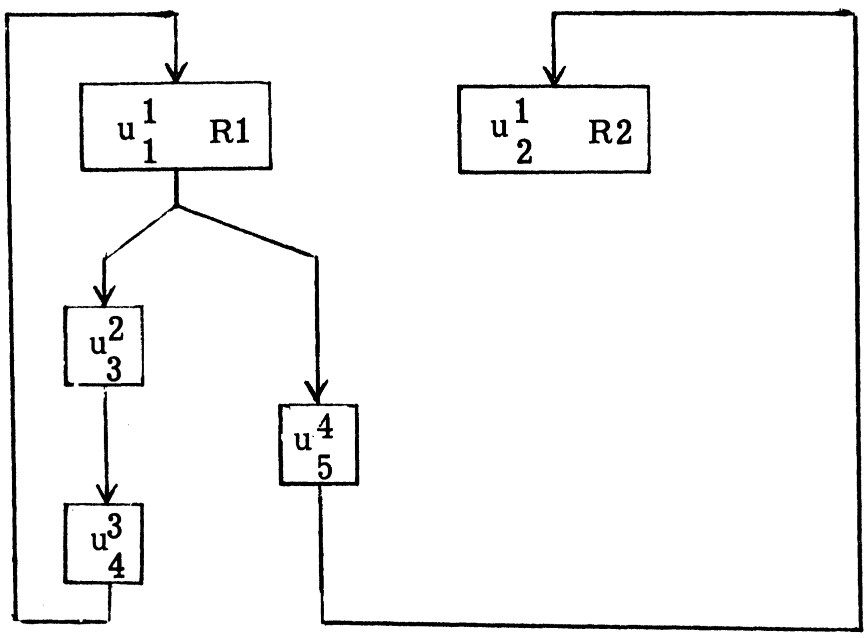
Instruction I2:

max execution time	=	90
weighting factor	=	y
statement group:		

$\overline{R2} \leftarrow AOP3(R1)$

Table 5.24 Architecture for Example 5.3

Date Path:



Unit Number	Unit Type	Operator	Cost	Delay
1	1	HOP0	2	2
2	1	HOP0	2	2
3	2	HOP1	10	10
4	3	HOP3	15	15
5	4	HOP4	20	20

Data path cycle time:

$$T = \max\left[(\delta_{u_1} + \delta_{u_2} + \delta_{u_3}), (\delta_{u_1} + \delta_{u_5}) \right]$$

$$= \max\,[\,27,\ 22\,] = 27$$

Data path cost:

$$C' = 2 + 2 + 10 + 15 + 20 = 49$$

Figure 5.5 Data path for Example 5.3, Case 1

Instruction I1': $I1' = (\psi 1_1)$

$\psi 1_1$:	$p1_2 \leftarrow HOP1(R1)$	000
	$q1_3 \leftarrow HOP0(p1_2)$	100
	$\overline{R1} \leftarrow HOP2(q1_3)$	110

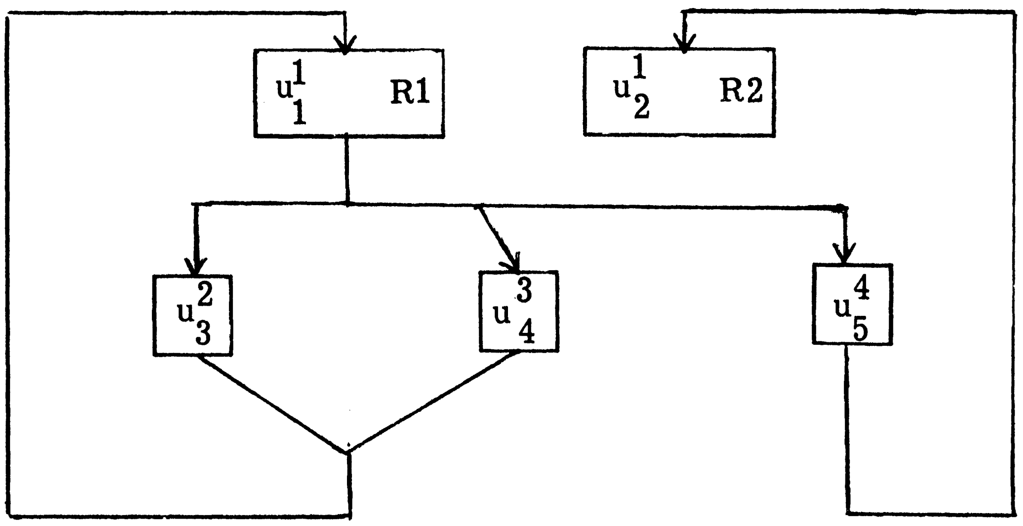
Instruction I2': $I2' = (\psi 2_1)$

$\psi 2_1$: $\overline{R2} \leftarrow HOP3(R1)$

$$\begin{aligned} WAIET &= (|I1'| \cdot \phi_1 + |I2'| \cdot \phi_2) \cdot T \\ &= (1 \cdot 8 + 1 \cdot 2) \cdot 27 \\ &= 270 \end{aligned}$$

Table 5. 25 Instruction implementation for Example 5. 3, Case 1

Data Path:



Unit Number	Unit Type	Operator	Cost	Delay
1	1	HOP0	2	2
2	1	HOP0	2	2
3	2	HOP1	10	10
4	3	HOP2	15	15
5	4	HOP3	20	20

Data path cycle time:

$$T = \max\left[(\delta_{u_1} + \delta_{u_3}), (\delta_{u_1} + \delta_{u_4}), (\delta_{u_1} + \delta_{u_5}) \right]$$

$$= \max [12, 17, 22] = 22$$

Data path cost:

$$C' = 2 + 2 + 10 + 15 + 20 = 49$$

Table 5.6 Data path for Example 5.3, Case 2

Instruction I1': $I1' = (\psi 1_1, \psi 1_2)$

$\psi 1_1$: $\overline{R1} \leftarrow HOP1(R1)$

$\psi 1_2$: $\overline{R1} \leftarrow HOP2(R1)$

Instruction I2': $I2' = (\psi 2_1)$

$\psi 2_1$: $\overline{R1} \leftarrow HOP3(R1)$

$$\begin{aligned} WAIET &= (|I1'| \cdot \phi_1 + |I2'| \cdot \phi_2) \cdot T \\ &= (2 \cdot 2 + 1 \cdot 8) \cdot 22 \\ &= 264 \end{aligned}$$

Table 5.26 Instruction implementation for Example 5.3, Case 2

Architecture	Data Path Solution	WAIET	
Case 1	Case 1	270	*
Case 1	Case 2	396	
Case 2	Case 1	270	
Case 2	Case 2	264	*

* - The solution selected by the program.

Table 5.27 Comparison of some solutions to Example 5.3, Cases 1 and 2

5.4 AN EXAMPLE WITH SYSTEM COST LIMIT VARIATION

In this example we consider three cases where the algorithm library and the hardware library are kept fixed and the architecture is changed only by increasing the system cost limit. This will cause the program to modify the solution from case to case to achieve performance gains as the cost limit rises. This example involves a problem somewhat less trivial than the preceding three examples and perhaps the reader will find the optimal solution less obvious.

For this example we make use of two operators AOP5 and HOP5 which have not been used before. These both have two input parameters rather than one. This is not really a significant change and has not occurred before this simply because one input parameter has been sufficient. In general, we might expect to encounter operators with two input parameters more often than those having one in a real problem environment and for this example we feel two input parameter operators are more appropriate.

We describe the algorithm library in Table 5.28 and provide one algorithm for each operator. The hardware library is defined in Table 5.29 where we see that unit u_2 has two input ports. Finally the corresponding GH Table is given in Table 5.30. There is only one entry again for each of the two operators. These libraries do not change in this example so we will use these tables for all three cases.

The architecture for this example changes only in that the system cost limit is increased from Case 1 through Case 3 so we will specify the architecture in Table 5.31 and indicate the system cost limit as x.

In Case 1 we set

$$x = 30.$$

The program responds to this input with the statement that there is no possible realization of this architecture. Examining the GH Table we see that any data path for this architecture must have a copy of both unit u_2 and u_3 to implement AOP1 and AOP5. Also it must have at least two copies of unit u_1 since there are two architecture registers. So for a unit set of

$$u = \{u_1^1, u_2^1, u_3^2, u_4^3\}$$

the cost is

$$C' = 2 + 2 + 10 + 15 = 29.$$

This is cost feasible since

$$C' = 29 < x = 30$$

but there is insufficient surplus for any additional units. We can examine the possible bus configuration for these units but eventually realize that there is none which will allow the architecture to be realized.

For Case 2 we change the cost limit to

$$x = 35$$

and now there is a solution which is presented in Figure 5.7 and Table 5.32. We immediately see what change has been made. With the increased cost limit the unit set can be augmented by an additional register unit, It is possible now to construct a realization of the architecture in which the instruction is implemented in two cycles yielding

$$\text{WAIET} = 34.$$

In Figure 5.7 we notice that the two input ports of unit u_4 are not clearly differentiated. Rather we indicate only that there are three busses going into the unit without specifying the particular port to which they are connected. This follows from the fact that there is no cost associated with busses. Therefore we may assume all three busses go to both input ports and the program does not bother to keep track of which ports need not receive a particular bus. This is the first time we have encountered this particular phenomenon since this is the first occurrence of a multiple input port unit in a problem solved by the program.

Now to continue with Case 3, we raise the cost limit to

$$x = 40$$

and display the new solution in Figure 5.8 and Table 5.33. In this case, with the increase in possible data paths provided by the new cost limit, one having improved performance is now available. The additional register of the previous solution has been discarded. The implementation of the two consecutive HOP1 operators can be achieved directly by the use of two sequential copies of unit u_2. The third register of the previous solution was required only to allow a temporary result to be saved so that one copy of unit u_2 could be used to perform the two required HOP1 operations. We also realize that further increases in the system cost limit will not change the solution. There can be no higher performance for this architecture with the given algorithm and hardware libraries. The instruction is executed in one cycle and

$$\text{WAIET} = 22.$$

$$G = \{g_{1,1}, g_{5,1}\}$$

$g_{1,1}$	A ← AOP1(B) : A ← HOP1(B)
$g_{5,1}$	A ← AOP5(B,C) : A ← HOP5(B,C)

Table 5.28 Algorithm library for Example 5.4

$$H = \{u_1, u_2, u_3\}$$

for unit u_1:

Cost = 2 Delay = 2 Number of Functions = 2

$f_{1,1}$	$\overline{r1} \leftarrow$ HOP0(q1)
$f_{1,2}$	p1 $\leftarrow$ HOP0(r1)

for unit u_2:

Cost = 10 Delay = 10 Number of Functions = 1

$f_{2,1}$	p1 $\leftarrow$ HOP5(q1, q2)

for unit u_3:

Cost = 15 Delay = 15 Number of Functions = 1

$f_{3,1}$	p1 $\leftarrow$ HOP1(q1)

Table 5.29 Hardware library for Example 5.4

Entry	Operator	Algorithm Sequence	Unit Set	Delay
1	AOP1	$g_{1,1}$	u_3	15
2	AOP5	$g_{5,1}$	u_2	10

Table 5.30 GH Table for Example 5.4

Number of Registers	=	2
Number of Instructions	=	1
System Cost Limit	=	x

Instruction I1:

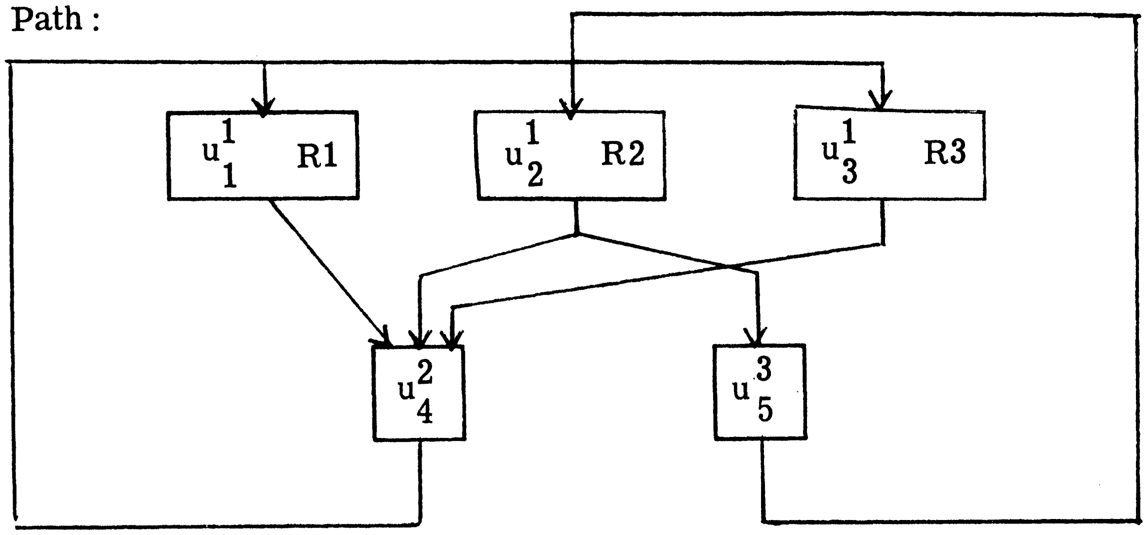
max execution time	=	90
weighting factor	=	1

statement group:

$A \leftarrow AOP5(R1, R2)$	000
$\overline{R1} \leftarrow AOP5(R1, A)$	100
$\overline{R2} \leftarrow AOP1(R2)$	000

Table 5.31 Architecture for Example 5.4

Data Path:



Unit Number	Unit Type	Operator	Cost	Delay
1	1	HOP0	2	2
2	1	HOP0	2	2
3	1	HOP0	2	2
4	2	HOP5	10	10
5	3	HOP1	15	15

Data path cycle time:

$$T = \max[(\delta_{u_1} + \delta_{u_4}), (\delta_{u_1} + \delta_{u_5})]$$

$$= \max[(2 + 10), (2 + 15)] = 17$$

Data path cost:

$$C' = 2 + 2 + 2 + 10 + 15 = 31$$

Figure 5.7 Data path for Example 5.4, Case 2

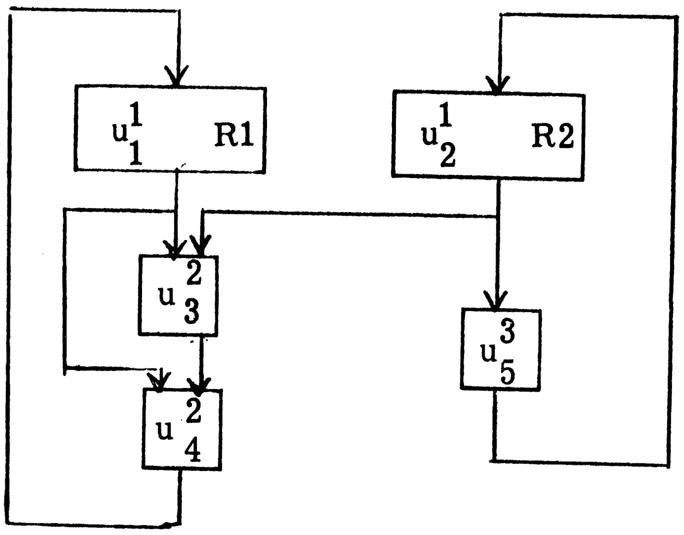
Instruction I': $I' = (\psi_1, \psi_2)$

ψ_1:	$\overline{R3} \leftarrow HOP5(R1,R2)$		00
	$\overline{R2} \leftarrow HOP1(R2)$		00
ψ_2:	$\overline{R1} \leftarrow HOP5(R1,R3)$		

$$\begin{aligned} WAIET &= |I1'| \cdot \phi_1 \cdot T \\ &= 2 \cdot 1 \cdot 17 = 34 \end{aligned}$$

Table 5.32 Instruction implementation for Example 5.4, Case 2

Data Path:



Unit Number	Unit Type	Operator	Cost	Delay
1	1	HOP0	2	2
2	1	HOP0	2	2
3	2	HOP5	10	10
4	2	HOP5	10	10
5	3	HOP1	15	15

Data path cycle time:

$$\begin{aligned} T &= \max\left[(\delta_{u_1} + \delta_{u_3} + \delta_{u_4}), (\delta_{u_2} + \delta_{u_5})\right] \\ &= \max\left[(2 + 10 + 10), (2 + 15)\right] = 22 \end{aligned}$$

Data path cost:

$$C' = 2 + 2 + 10 + 10 + 15 = 39$$

Figure 5.8 Data path for Example 5.4, Case 3

Instruction I1' : I1' = (ψ_1)

ψ_1:	$p1_3$	$\leftarrow$	HOP5(R1,R2)	0000
	$q1_4$	$\leftarrow$	HOP0($p1_3$)	1000
	$\overline{R1}$	$\leftarrow$	HOP5(R1,$q1_4$)	1100
	$\overline{R2}$	$\leftarrow$	HOP1(R2)	0000

$$\text{WAIET} = |I1'| \cdot \phi_1 \cdot T$$

$$= 1 \cdot 1 \cdot 22 = 22$$

Table 5.33 Instruction implementation for Example 5.4, Case 3

5.5 A REPRESENTATIVE COMPUTER

Now we try to present some insight into how an actual computer might be described in this model. The previous examples were all attempts to illustrate certain facets of the programming system, but now we demonstrate the manner in which this program might be applied in a real design environment. This is not intended to be a complete computer description but rather fragments of an architecture which illustrate some of the common features of current production computers.

There are ten operators in the architecture operator set and seven in the hardware operator set. These are listed and defined in Table 5.34 and 5.35, respectively. These operators should have an air of familiarity about them which was not present in the previous examples when we dealt in more abstract terms with computer transformations. We will present the algorithm and hardware libraries first and then discuss the architecture.

Algorithm Library

The algorithm library is kept simple for this example with only one operator having more than one library entry. But we see that four of the ten operators are described purely in terms of other architecture operators. The algorithm library is shown in Table 5.36. Notice that in this example we have used bit modifiers in the statements since we want to be able to refer to subsections of operands.

Hardware Library

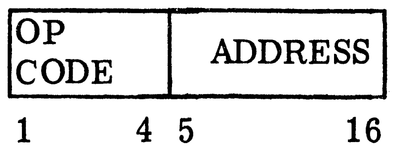
The hardware library contains four unit types. These are: a register unit, an adder, a memory unit, and an operation decoder. The detailed specifications for the library are given in Table 5.37. Since there is freedom to define both sets of operators as well as the algorithms and hardware units, it is possible to express various desired characteristics in many different ways. This representation of a memory, for instance, is only one of a number of different schemes that could be used.

The GH Table

The GH Table for the algorithm and hardware library is given in Table 5.38. There is one entry for each operator but we see that for the last three operators, which are expressed in terms of architecture operators in the algorithm library, a sequence of algorithms is needed for compilation.

Architecture

The architecture will use a sixteen bit word width and a twelve bit address width. We assume an instruction format of four bits for the operation code and twelve bits for an absolute memory address. We can diagram this:



Instruction Format

We will include six registers in the architecture and these are defined in Table 5. 39. The computer listings reproduced in the appendix do not use the mnuemonic names employed throughout this discussion. Instead the registers are simply called R1 through R6. To aid the reader in relating this discussion with the data in the appendix, we include in Table 5.39 a translation between the two sets of names.

The architecture defines a processor having seven instructions, but one of these is a definition of the Instruction Fetching procedure that we assume is common to the other six. We will elaborate on this later. Then there are typical LOAD, ADD, and STORE instructions with the first two being indexed. We also have included two rather unusual instructions that assume a linked-list data structure in memory. Figure 5.9 is a diagram illustrating the particular structure that has been assumed. There is a forward pointer, a backward pointer, and one data item associated with each list element.

It perhaps should be pointed out that the structure for this list is specifically defined in the algorithm library. In the event that the list format were changed, the algorithms for ADNX, ADPR, and VAL would have to be changed. As we saw in Example 5. 2, the architecture could have been written without the use of these particular operators by letting the designer describe the list manipulation instructions directly in terms of memory reading and address incrementing. However, since the operators have been defined, it is convenient to use them and at the same time make the architecture description independent of the particular list structure.

The last instruction of the architecture is a WRITE I/O instruction. We define an output bus in the architecture and this instruction transfers the right half of a memory word to the bus. This is the most natural way to represent an I/O instruction here since the model allows the user to specify the execution sequence of instruction statements but not specific timing relationships. This is satisfactory for describing I/O instructions on the IBM System 360 , for instance, but is less realistic for the DEC PDP-8 where specific time delays are given in the programming manual.

In this example we have chosen to express the Instruction Fetch procedure as a separate instruction in order to illustrate one possible way of expressing conditional control branches. By conditional control branching we mean the altering of the sequence of execution of a single instruction based on some data dependent conditions. This is not to be confused with conditional program branches which alter the sequence of execution of a program by changing the instruction counter. In a microprogrammed computer a control branch would typically imply a hardware provision for changing the next control memory address as a function of specific data conditions.

Since the model does not provide for conditional statement execution in a statement group, one way of representing this is, as we have done in this example, to define the statements which are intended to precede the control branch as a separate instruction. The various statement groups which represent the different execution paths following the control branch are then also defined as separate instructions.

In using this representation we need to be able to define the interface between the end of Instruction Fetch and the beginning of the other six instructions. This interface does not really exist for the programmer because for him each instruction includes the instruction fetching statement group followed by the particular instruction statements. In this example the IR register exists in order to define data between Instruction Fetch and each of the other instructions. It does not correspond to a register the user should actually be able to modify.

Another point to be made about this example is the use of a Status Register, ST, to receive the output of the operation decode operator. We think of the ST register as one boundary between data and control in this example. In this study we do not model the control functions of a computer but do acknowledge the fact that somehow the output of the ST register is affecting the conditional branch discussed above that stands between the Instruction Fetch statement group and each of the other instructions.

The architecture is defined in Table 5.40.

The data path which the program generates as a solution is shown in Figure 5.10 and we can see a few interesting features. First, the registers XR and ST have no input and no output bus, respectively. This is exactly the same as the situation in Example 5.3 where a bus is not included because it is not required by the architecture. No instruction changes that value of XR so there is no input bus, and ST is never an input operand in a statement so it has no output bus. Another interesting point is that there are busses connecting the adder to the memory and memory to the adder which constitutes what we have called a loop in the data path. One or the other of these can be used in calculating the cycle time since the answer is the same in either case, but measuring the delay over a path using both busses simultaneously implies including the delay of one unit twice and must be avoided.

Table 5.41 summarizes the number of cycles required for each instruction and the calculation of WAIET. The actually implementation of the instruction is listed in Table 5.42.

Operator Number	Operator	Definition
0	MOVE(A)	Transfer without alteration (o_0)
1	OPDC(A)	Operation decoding of A
2	ADD(A, B)	Addition of A to B
3	INC1(A)	Increment A by 1
4	INC2(A)	Increment A by 2
5	MRD(A)	Memory read per address A
6	MWRT(A, B)	Write B into memory per address A
7	ADNX(A)	Find address of next list element where A is address of present list element
8	ADPR(A)	Find address of previous list element where A is address of present list element
9	VAL(A)	Find the data item of list element whose address is A

Table 5.34 Architecture operators for Example 5.5

Operator Number	Operator	Definition
0	move(A)	Transfer without alteration (o_0')
1	opdc(A)	Operation decoding of A
2	add(A, B)	Addition of A to B
3	incl(A)	Increment A by 1
4	mad(A)	Memory address decode of A
5	wrt(A)	Write A into memory
6	rd	Read memory

Table 5.35 Hardware operators for Example 5.5

$G = (g_{1,1}, g_{2,1}, g_{2,2}, g_{3,1}, g_{4,1}, g_{5,1}, g_{6,1}, g_{7,1}, g_{8,1}, g_{9,1})$

$g_{1,1}$	B(1,6) ← OPDC(A(1,4)) :	B(1,6) ← opdc(A(1,4))	
$g_{2,1}$	B(1,16) ← ADD(A(1,16),C(1,16)):	B(1,16)←add(A(1,16),C(1,16))	
$g_{2,2}$	B(1,16)←ADD(A(1,16),C(1,16)) :	B(1,16)←add(C(1,16),A(1,16))	
$g_{3,1}$	B(1,12)←INC2(A(1,12)) :	B(1,12)←inc1(A(1,12))	
$g_{4,1}$	B(1,12)←INC2(A(1,12)) :	Z(1,12)← INC1(A(1,12))	00
		B(1,12)←INC1(Z(1,12))	10
$g_{5,1}$	B(1,16)←MRD(A(1,12)) :	0←mad(A(1,12))	00
		B(1,16)← rd(1,16)	10
$g_{6,1}$	0← MWRT(A(1,12),B(1,16)) :	0 ← mad(A(1,12))	00
		0 ← wrt(B(1,16))	10
$g_{7,1}$	B(1,12) ← ADNX(A(1,12)) :	B(1,12) ← MRD (A(1,12))	
$g_{8,1}$	B(1,12) ← ADPR(A(1,12)) :	Z(1,12)←INC1(A(1,12))	00
		B(1,12) ←MRD(Z(1,12))	10
$g_{9,1}$	B(1,16) ← VAL(A(1,12)) :	Z(1,12)←INC2(A(1,12))	00
		B(1,16)←MRD(Z(1,12))	10

Table 5.36 Algorithm library for Example 5.5

$$H = (u_1, u_2, u_3, u_4)$$

For unit u_1:

Cost = 2	Delay = 1	Number of Functions = 2

$f_{1,1}$	$\overline{r1} \leftarrow$ move (q1)
$f_{1,2}$	p1 $\leftarrow$ move (r1)

For unit u_2:

Cost = 15	Delay = 5	Number of Functions = 3

$f_{2,1}$	p1 $\leftarrow$ add (q1, q2)
$f_{2,2}$	p1 $\leftarrow$ add (q2, q1)
$f_{2,3}$	p1 $\leftarrow$ inc1 (q1)

For unit u_3:

Cost = 20	Delay = 4	Number of Functions = 3

$f_{3,1}$	0 $\leftarrow$ mad (q1)
$f_{3,2}$	0 $\leftarrow$ wrt (q2)
$f_{3,3}$	p1 $\leftarrow$ rd

For unit u_4:

Cost = 5	Delay = 5	Number of Functions = 1

$f_{4,1}$	0 $\leftarrow$ opdc (q1)

Table 5.37 Hardware library for Example 5.5

Entry	Operator	Algorithm Sequence	Unit Set	Delay
1	OPDC	$g_{1,1}$	u_4	5
2	ADD	$g_{2,1}$	u_2	5
3	INC1	$g_{3,1}$	u_2	5
4	INC2	$g_{4,1}$	u_2, u_2	10
5	MRD	$g_{5,1}$	u_3, u_3	8
6	MWRT	$g_{6,1}$	u_3, u_3	8
7	ADNX	$g_{7,1}, g_{5,1}$	u_3, u_3	8
8	ADPR	$g_{8,1}, g_{3,1}, g_{5,1}$	u_2, u_3, u_3	13
9	VAL	$g_{9,1}, g_{4,1}, g_{3,1}, g_{3,1}, g_{5,1}$	u_2, u_2, u_3, u_3	18

Table 5.38 GH Table for Example 5.5

Mneumonic Name	Function	Width (in bits)	Program Name
AC	Accumulator	16	R1
DA	Data Address	12	R2
IC	Instruction Counter	12	R3
IR	Instruction Register	16	R4
XR	Index Register	12	R5
ST	Status Register	6	R6

Table 5.39 Architecture registers for Example 5.5

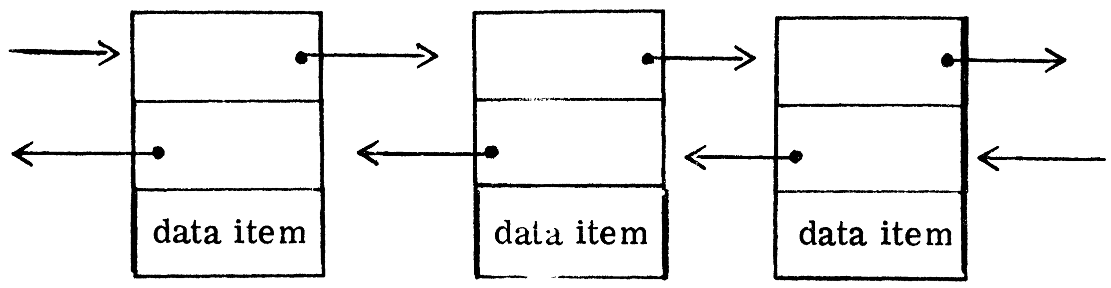


Figure 5.9 The Linked-list structure for Example 5.5

Number of Registers	= 6 (see Table 5.38)
Number of Instructions	= 7
Max System Cost	= 54
Number of Output Busses	= 1
Width of Output Bus	= 8

Instruction 1

Instruction Fetch. This is a description of the common instrction fetch procedure which is assumed to precede each of the other six instructions.

Maximum Execution Time	= 90
Weighting Factor	= 10

$Z(1,16) \leftarrow MRD(IC(1,12))$	0000
$\overline{IC}(1,12) \leftarrow INC1(IC(1,12))$	1000
$\overline{ST}(1,6) \leftarrow OPDC(Z(1,4))$	1000
$\overline{IR}(1,16) \leftarrow MOVE(Z(1,16))$	1000

Table 5.40 Architecture for Example 5.5

Instruction 2

Load the AC per the instruction address - Indexed

Maximum Execution Time = 90

Weighting Factor = 5

Z(1,12) ← ADD(IR(5,12),XR(1,12))	00
$\overline{\text{AC}}$(1,16) ← MRD(Z(1,12))	10

Instruction 3

Add memory per instruction address to AC - Indexed

Maximum Execution Time = 90

Weighting Factor = 5

Z1(1,12) ← ADD(IR(5,12(,XR(1,12))	000
Z2(1,16) ← MRD(Z1(1,12))	100
$\overline{\text{AC}}$(1,16) ← ADD(AC(1,16),Z2(1,16))	110

Instruction 4

Store AC per the instruction address

Maximum Execution Time = 90

Weighting Factor = 4

0 ← MWRT(IR(5,12),AC(1,16))

Table 5.40 Architecture for Example 5.5 (Continued)

Instruction 5

Load DA with address of next list element and load AC with data item of the next list element.

Maximum Execution Time = 90

Weighting Factor = 3

$Z(1,12) \leftarrow ADNX(DA(1,12))$	000
$\overline{AC}(1,16) \leftarrow VAL(Z(1,12))$	100
$\overline{DA}(1,12) \leftarrow MOVE(Z(1,12))$	100

Instruction 6

Load DA with address of previous list element and load AC with data item of the previous list element.

Maximum Execution Time = 90

Weighting Factor = 2

$Z(1,12) \leftarrow ADPR(DA(1,12)$	000
$\overline{AC}(1,16) \leftarrow VAL(Z(1,12))$	100
$\overline{DA}(1,12) \leftarrow MOVE(Z(1,12))$	100

Table 5.40 Architecture for Example 5.5 (Continued)

Instruction 7

Maximum Execution Time = 90

Weighting Factor = 1

Write I/O. Transfer the right half of a memory word to the output bus, Q1.

$$Q1(1,8) \leftarrow MRD(IR(5,12))(8,8)$$

Table 5.40 Architecture for Example 5.5 (Continued)

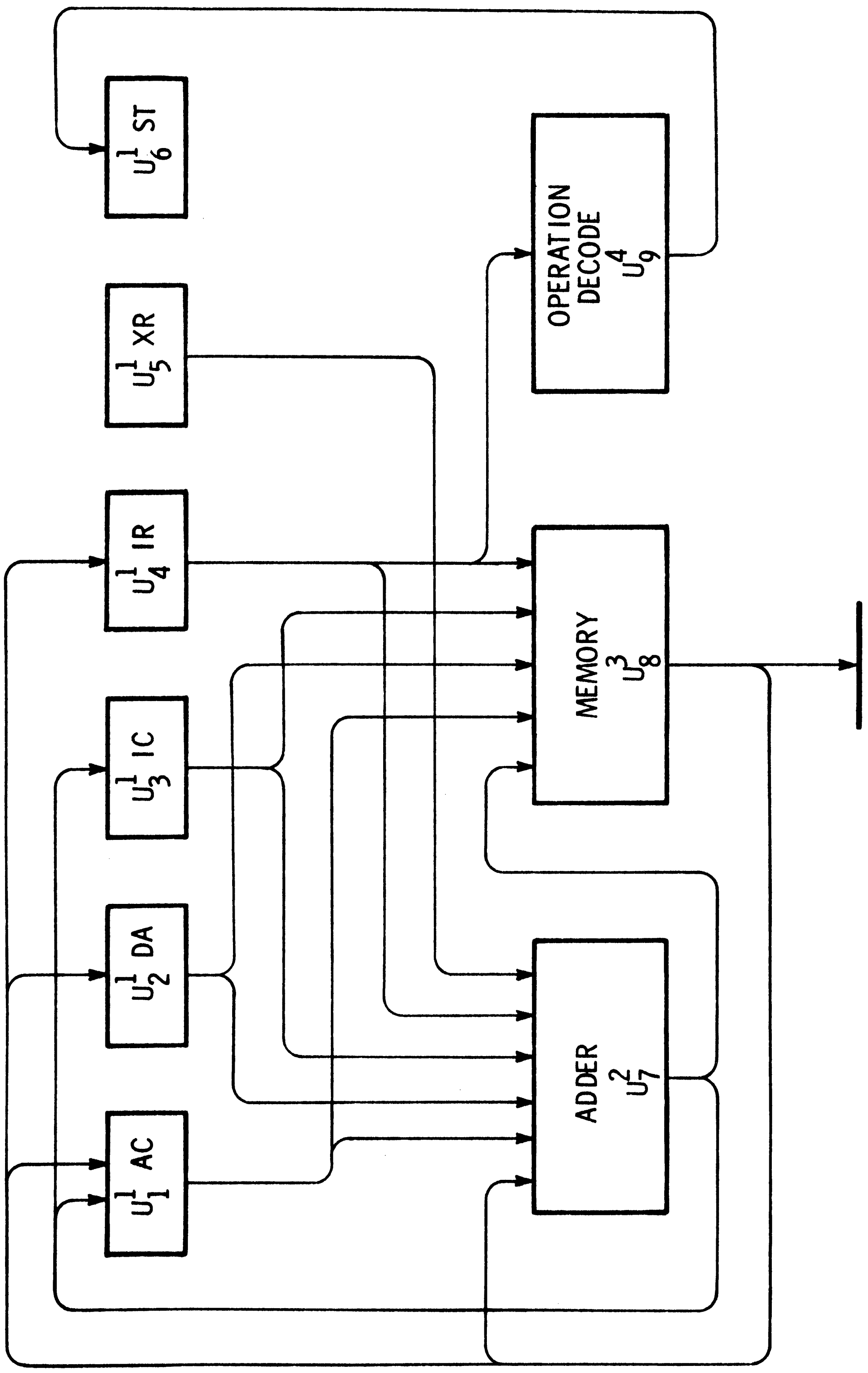


Figure 5.10 Data path for Example 5.5

Unit Number	Unit Type	Operator	Cost	Delay
1	1	move	2	1
2	1	move	2	1
3	1	move	2	1
4	1	move	2	1
5	1	move	2	1
6	1	move	2	1
7	2	add, incl	15	5
8	3	mad, wrt, rd	20	4
9	4	odpc	5	5

Data path cycle time:

$$T = \delta_{reg} + \delta_{u_7} + \delta_{u_8}$$

$$T = 1 + 5 + 4 = 10$$

Data path cost:

$$C' = 6 \cdot 2 + 15 + 20 + 5 = 52$$

Figure 5.10 Data path for Example 5.5 (Continued)

Instruction Number	Function	Weighting Factor	Number of Cycles
1	Instruction Fetch	10	3
2	Indexed Load	5	2
3	Indexed Add	5	2
4	Store	4	2
5	Get Next Element	3	4
6	Get Previous Element	2	4
7	Write I/O	1	2

$$\text{WAIET} = T \cdot \sum_{i=1}^{7} |I_i'| \cdot \phi_i$$

$$= 10(3 \cdot 10 + 2 \cdot 5 + 2 \cdot 5 + 2 \cdot 4 + 4 \cdot 3 + 4 \cdot 2 + 2 \cdot 1)$$

$$= 800$$

Table 5.41 Number of cycles and WAIET calculation for Example 5.5

Instruction Implementation

Instruction 1 : Instruction Fetch

$$I1' = (\psi 1_1, \psi 1_2, \psi 1_3)$$

$\psi 1_1$:	$0 \leftarrow \text{mad(IC)}$	
$\psi 1_2$:	$\overline{\text{IR}} \leftarrow \text{mrd}$	00
	$\overline{\text{IC}} \leftarrow \text{incl(IC)}$	00
$\psi 1_3$:	$\overline{\text{ST}} \leftarrow \text{opdc(IR(1, 4))}$	

Instruction 2 : Indexed Load

$$I2' = (\psi 2_1, \psi 2_2)$$

$\psi 2_1$:	$p1_7 \leftarrow \text{add(IR(5, 12), XR)}$	000
	$q1_8 \leftarrow \text{move}(p1_7)$	100
	$0 \leftarrow \text{mad}(q1_8)$	110
$\psi 2_2$:	$\overline{\text{AC}} \leftarrow \text{mrd}$	

Table 5.42 Instruction implementation for Example 5.5

Instruction 3 : Indexed Add

$I3' = (\psi 3_1, \psi 3_2)$

$\psi 3_1$:	$p1_7 \leftarrow$ add (IR(5,12),XR)	000
	$q1_8 \leftarrow$ move ($p1_7$)	100
	$0 \leftarrow$ mad ($q1_8$)	110
$\psi 3_2$:	$p1_8 \leftarrow$ mrd	000
	$q1_7 \leftarrow$ move ($p1_8$)	100
	$\overline{AC} \leftarrow$ add ($q1_7$, AC)	110

Instruction 4 : Store

$I4' = (\psi 4_1, \psi 4_2)$

$\psi 4_1$:	$0 \leftarrow$ mad (IR)	00
$\psi 4_2$:	$0 \leftarrow$ mwrt (AC)	10

Table 5.42 Instruction implementation for Example 5.5 (Continued)

Instruction 5 : Get Next Element

$I5' = (\psi 5_1, \psi 5_2, \psi 5_3, \psi 5_4)$

$\psi 5_1$:	$0 \leftarrow \text{mad (DA)}$	
$\psi 5_2$:	$pl_8 \leftarrow \text{mrd}$	0000
	$\overline{DA} \leftarrow \text{move}(pl_8)$	1000
	$ql_7 \leftarrow \text{move }(pl_8)$	1000
	$\overline{AC} \leftarrow \text{incl }(ql_7)$	1100
$\psi 5_3$:	$pl_7 \leftarrow \text{incl(AC)}$	000
	$ql_8 \leftarrow \text{move}(pl_7)$	100
	$0 \leftarrow \text{mad}(ql_8)$	110
$\psi 5_4$:	$\overline{AC} \leftarrow \text{mrd}$	

Instruction 6 : Get Previous Element

$I6'$: $(\psi 6_1, \psi 6_2, \psi 6_3, \psi 6_4)$

$\psi 6_1$:	$pl_7 \leftarrow \text{incl(DA)}$	000
	$ql_8 \leftarrow \text{move}(pl_7)$	100
	$0 \leftarrow \text{mad}(ql_8)$	110
$\psi 6_2$:	$pl_8 \leftarrow \text{mrd}$	0000
	$\overline{DA} \leftarrow \text{move}(pl_8)$	1000
	$ql_7 \leftarrow \text{move}(pl_8)$	1000
	$\overline{AC} \leftarrow \text{incl}(ql_7)$	1100

Table 5.42 Instruction implementation for Example 5.5 (Continued)

$\psi 6_3$:	$p1_7 \leftarrow incl(AC)$	000
	$q1_8 \leftarrow move(p1_7)$	100
	$0 \leftarrow mad(q1_8)$	110
$\psi 6_4$:	$\overline{AC} \leftarrow mrd$	

Instruction 7 : Write I/O

$I7' : (\psi 7_1, \psi 7_2)$

$\psi 7_1$:	$0 \leftarrow mad(IR)$
$\psi 7_2$:	$Q1 \leftarrow mrd$

Table 5.42 Instruction implementation for Example 5.5 (Continued)

CHAPTER VI

CONCLUDING REMARKS

In reviewing the value and accomplishments of this research we must consider both the mathematical model which is developed and the computer implementation of the model. Each has its own merits and weaknesses.

We believe the mathematical model is a successful attempt to formalize the central processor of a digital computer system. It allows a precise definition of the components of a data path and the transformations that can be accomplished by the data path during a cycle or an instruction. A particular strong point is the definition used for units. We have made this quite general and powerful so that complex units can be included which can themselves be independent digital systems. In addition, the definition of the interconnecting busses is adequate to describe very sophisticated gating schemes between units. Another feature of this model is that we have retained independence from any fixed set of operators for either hardware or architecture definitions. We use instead an algorithm library which allows great latitude in the modeling of particular applications.

Certainly many extensions of the model could be considered. One instance would be the addition of a cost function for the gating and bussing. To some extent these can be covered by making allowances for the gating in the unit cost functions. But an explicit definition of this cost would be more accurate and might be considered. One might also choose to expand the definition of units to include limits for fan-in and fan-out on the busses.

Another extension of the model could be the provision for more sophisticated definition of timing relationships. This would allow better modeling of memory and I/O units. Another reasonable addition would be the definition of a cost function which would model the cost of the data path controls. This could be simply a "cost per CPU cycle" or perhaps a more elaborate function representing the decoding and sequencing of a microprogram control memory.

We feel the model as presented here, however, is sufficiently complete and sophisticated without these extensions for the problems we are treating and the addition of these functions could only obscure the use of the model in a practical application. That is to say, although the model will easily accomodate further extensions, at this time the implications of these on any computer realization of the model could be quite severe.

Turning to the computer implementation of this research, we have a system of programs that we believe does demonstrate the potential in this area of automated design. In deciding which portions of the model should be included in the implementation and which, because of practical constraints on the execution time, would have to be omitted, we have attempted to find a middle ground between fidelity to the model and the ability to guarantee true optimal results. We saw in Chapter IV that both of these goals should not be simultaneously pursued at this time. It is hoped that a middle ground approach would best illuminate the entire spectrum of possible implementations.

Now that the program is complete, it is natural to consider possible modifications. Generally speaking, the development of the computer program has slanted toward achieving an accurate measure of the performance of a data path at the sacrifice of relying on estimates or approximations for the compilation and selection of a unit set. The experience gained by using the program will undoubtedly indicate areas where the intuitive decisions made in the development of the system can be improved. Certain facets of the model may now appear more critical than was initially assumed. Similarly, the estimates of program execution may prove to be overly conservative, and thus, increased sophistication of various algorithms of the program could be possible. Probably use of this program will lead to a desire to include more features of the model.

These are conclusions one expects to reach after the development of a prototype system, but they should not interfere with an appreciation of the success of a first implementation of this model. The programming system can in fact compute reasonable data path designs for non-trivial architectures. Realistically it is not expected that these solutions can lead directly to hardware design. The level of optimization would be considered unsatisfactory to people involved in manufacturing a real computer. However, we do feel this work does have immediate application as a tool for the investigation of various architecture definitions, hardware units, and algorithms and can be used to provide a starting point from which a final, detailed design can evolve.

APPENDIX

COMPUTER LISTINGS FOR THE EXAMPLES OF CHAPTER V

This appendix is a record of all the input and output data from the computer solutions of the examples in Chapter V. Each of the eleven cases that constitute the five examples has four associated listings:

1. The algorithm and hardware library
2. The GH Table
3. The architecture
4. The results

For the GH Table generation program, item 1 is the input and item 2 is the output. For the data path generation program items 1, 2, and 3 are input and item 4 is the output. Some translation and interpretation have been made in preparing this data for presentation in Chapter V. The most significant steps in this translation involve the result listings and are as follows:

1. The procedure for selecting the data path unit set inserts many extra registers if the cost limit is relatively high. Where any of these have gone unused in the solution, they have been manually omitted from the tabulation in Chapter V and the system cost appropriately adjusted. The fact that a register is not being used is indicated by the register always being empty and the data path connection

matrix having no input or output busses to it.

2. For two cases (Example 2, Case 3 and Example 4, Case 3), the program computed more than one data path solution. Each of these solutions is a complete and independent realization of the archtecture differing in the number of registers contained in the data path. In these cases the solution iteration having the minimum WAIET was manually selected as the result described in Chapter V.

3. The microprogram portion of the result is printed as a sequence of cycles

$$\psi = \langle \vec{F}, \vec{M} \rangle$$

for each instruction. From this information the cycle statement groups are constructed for tabulation in Chapter V.

4. The matrices that define the busses use a "T" or "F" to indicate the presence or absence of a bus respectively.

Algorithm and Hardware Libraries for Example 5.1, Case 1

```
02                                          NO OF ARCH OPERATORS
AOPO 1  AOP1 1
03                                          NO OF HARD OPERATORS
HOPO 1  HOP1 1  HOP2 1
02                                          NO OF ALGORITHMS
A(1,8):AOPO(B(1,8))(1,8) 1                  IST ALGORITHM
A:HOPO(B)(1,8)
A(1,8):AOP1(B(1,8))(1,8) 2                  2ND ALG
C(1,8):HOP1(B)(1,8) 00
A:HOP2(C)(1,8) 10
03                                          NO OF HARDWARE UNITS
P1                                          NO OF P PORTS
  8                                         WIDTH
Q1                                          NO OF Q PORTS
  8                                         WIDTH
COST=  2 DELAY=  2 FUNCS= 2
:HOPO(Q1)(1,8)                              FUNCTIONS
P1:HOPO(1,8)
P1                                          SAME FOR UNIT 2
  8
Q1
  8
COST= 10 DELAY= 10 FUNCS= 2
P1:HOP1(Q1)(1,8)
P1:HOP2(Q1)(1,8)
P1                                            SAME FOR UNIT 3
  8
Q1
  8
COST= 15 DELAY= 12 FUNCS= 1
P1:HOP2(Q1)(1,8)
```

GH Table for Example 5.1, Case 1

```
   1 ENTRIES FOR OPERATOR 1
  2     1  0  0  0  0  0  0  0    1  0  0
   2 ENTRIES FOR OPERATOR 2
 20     2  0  0  0  0  0  0  0    0  2  0
 22     2  0  0  0  0  0  0  0    0  1  1
```

Architecture for Example 5.1, Case 1

```
01                                       NO OF REGISTERS
 08                                      WIDTH
0                                        NO OF P PORTS
0                                        NO OF Q PORTS
50                                       COST LIMIT OF SYSTEM
01                                       NO OF INSTRUCTIONS
 01 01 90                                STMNTS,WTFCT,MAXTM
R1:AOP1(R1)(1,8)                          *
```

Results for Example 5.1, Case 1

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****


FOR ARCHITECTURE OPERATORS:  1  2
 THE GH TABLE SELECTION IS:  0  2



          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  00000000  00000000  00000000  00000000
00000000  00000000  00000000  00000000  00000000

        FOR UNITS:  11  12
THE FUNCTIONS ARE:   1   2

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                    F F F F F F F F F F F T
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    T F F F F F F F F F F F
                    F F F F F F F F F F T F

  FINAL REGISTER CONTENTS:
010B0108  00000000  00000000  00000000  00000000
00000000  00000000  00000000  00000000  00000000
```

```
        THE HARDWARE DATA PATH IS COMPOSED OF:

NO OF UNITS= 12   TOTAL REGISTERS= 10   SYSTEM COST=  40
WT AVE EXEC TIME=     22      CYCLE TIME=  22

DATA PATH UNIT NUMBER:  1  2  3  4  5  6  7  8  9 10 11 12
UNIT TYPE NUMBERS ARE:  1  1  1  1  1  1  1  1  1  1  2  2

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)

                   F F F F F F F F F F F T
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   T F F F F F F F F F F F
                   F F F F F F F F F F T F
```

Algorithm and Hardware Libraries for Example 5.1, Case 2

```
02                                  NO OF ARCH OPERATORS
AOPO 1  AOP1 1
03                                  NO OF HARD OPERATORS
HOPO 1  HOP1 1  HOP2 1
02                                  NO OF ALGORITHMS
A(1,8):AOPO(B(1,8))(1,8) 1          IST ALGORITHM
A:HOPO(B)(1,8)
A(1,8):AOP1(B(1,8))(1,8) 2          2ND ALG
C(1,8):HOP1(B)(1,8) 00
A:HOP2(C)(1,8) 10
03                                  NO OF HARDWARE UNITS
P1                                  NO OF P PORTS
  8                                 WIDTH
Q1                                  NO OF Q PORTS
  8                                 WIDTH
COST=  2 DELAY=  2 FUNCS= 2
:HOPO(Q1)(1,8)                      FUNCTIONS
P1:HOPO(1,8)
P1                                  SAME FOR UNIT 2
  8
Q1
  8
COST= 10 DELAY= 10 FUNCS= 2
P1:HOP1(Q1)(1,8)
P1:HOP2(Q1)(1,8)
P1                                    SAME FOR UNIT 3
  8
Q1
  8
COST= 15 DELAY=  8 FUNCS= 1
P1:HOP2(Q1)(1,8)
```

GH Table for Example 5.1, Case 2

```
    1 ENTRIES FOR OPERATOR 1
  2      1  0  0  0  0  0  0  0     1  0  0
    2 ENTRIES FOR OPERATOR 2
 20      2  0  0  0  0  0  0  0     0  2  0
 18      2  0  0  0  0  0  0  0     0  1  1
```

Architecture for Example 5.1, Case 2

```
01                                  NO OF REGISTERS
 08                                 WIDTH
0                                   NO OF P PORTS
0                                   NO OF Q PORTS
50                                  COST LIMIT OF SYSTEM
01                                  NO OF INSTRUCTIONS
 01 01 90                           STMNTS,WTFCT,MAXTM
R1:AOP1(R1)(1,8)                     *
```

Results for Example 5.1, Case 2

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****

FOR ARCHITECTURE OPERATORS:  1  2
 THE GH TABLE SELECTION IS:  0  3



          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  00000000  00000000  00000000  00000000
00000000  00000000  00000000  00000000  00000000

        FOR UNITS:  11  12
THE FUNCTIONS ARE:   1   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                            F F F F F F F F F F F T
                            F F F F F F F F F F F F
                            F F F F F F F F F F F F
                            F F F F F F F F F F F F
                            F F F F F F F F F F F F
                            F F F F F F F F F F F F
                            F F F F F F F F F F F F
                            F F F F F F F F F F F F
                            F F F F F F F F F F F F
                            F F F F F F F F F F F F
                            T F F F F F F F F F F F
                            F F F F F F F F F F T F

  FINAL REGISTER CONTENTS:
010B0108  00000000  00000000  00000000  00000000
00000000  00000000  00000000  00000000  00000000
```

```
      THE HARDWARE DATA PATH IS COMPOSED OF:

NO OF UNITS= 12   TOTAL REGISTERS= 10   SYSTEM COST=  45
WT AVE EXEC TIME=    20      CYCLE TIME=  20

DATA PATH UNIT NUMBER:  1  2  3  4  5  6  7  8  9 10 11 12
UNIT TYPE NUMBERS ARE:  1  1  1  1  1  1  1  1  1  1  2  3

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)

                        F F F F F F F F F F F T
                        F F F F F F F F F F F F
                        F F F F F F F F F F F F
                        F F F F F F F F F F F F
                        F F F F F F F F F F F F
                        F F F F F F F F F F F F
                        F F F F F F F F F F F F
                        F F F F F F F F F F F F
                        F F F F F F F F F F F F
                        F F F F F F F F F F F F
                        T F F F F F F F F F F F
                        F F F F F F F F F F T F
```

Algorithm and Hardware Libraries for Example 5.2, Case 1

```
04
AOP0 1  AOP1 1  AOP2 1  AOP3 1
04
HOP0 1  HOP1 1  HOP2 1  HOP3 1
04                                NO OF ALGORITHMS
A(1,8):AOP0(B(1,8))(1,8) 1        IST ALGORITHM
A:HOP0(B)(1,8)
A(1,8):AOP1(B(1,8))(1,8) 1        2ND ALG
A:HOP1(B)(1,8)
A(1,8):AOP2(B(1,8))(1,8) 1        3RD ALG
A:HOP2(B)(1,8)
A(1,8):AOP3(B(1,8))(1,8) 1        4TH ALG
A:HOP3(B)(1,8)
04                                NO OF HARDWARE UNITS
P1                                NO OF P PORTS
  8                               WIDTH
Q1                                NO OF Q PORTS
  8                               WIDTH
COST=  2 DELAY=  2 FUNCS= 2
:HOP0(Q1)(1,8)                    FUNCTIONS
P1:HOP0(1,8)
P1                                SAME FOR UNIT 2
  8
Q1
  8
COST= 10 DELAY= 10 FUNCS= 1
P1:HOP1(Q1)(1,8)
P1                                SAME FOR UNIT 3
  8
Q1
  8
COST= 15 DELAY= 15 FUNCS= 1
P1:HOP2(Q1)(1,8)
P1                                SAME FOR UNIT 4
  8
Q1
  8
COST= 20 DELAY= 20 FUNCS= 1
P1:HOP3(Q1)(1,8)
```

GH Table for Example 5. 2, Case 1

```
   1 ENTRIES FOR OPERATOR 1
  2     1  0  0  0  0  0  0  0    1  0  0  0
   1 ENTRIES FOR OPERATOR 2
 10     2  0  0  0  0  0  0  0    0  1  0  0
   1 ENTRIES FOR OPERATOR 3
 15     3  0  0  0  0  0  0  0    0  0  1  0
   1 ENTRIES FOR OPERATOR 4
 20     4  0  0  0  0  0  0  0    0  0  0  1
```

Architecture for Example 5. 2, Case 1

```
01                                  NO OF REGISTERS
 08                                 WIDTH
0                                   NO OF P PORTS
0                                   NO OF Q PORTS
50                                  COST LIMIT OF SYSTEM
01                                  NO OF INSTRUCTIONS
 03 01 90                           STMNTS,WTFCT,MAXTM
A(1,8):AOP1(R1)(1,8) 000
B(1,8):AOP2(A)(1,8) 100
R1:AOP3(B)(1,8) 110
```

Results for Example 5. 2, Case 1

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****

FOR ARCHITECTURE OPERATORS:  1  2  3  4
 THE GH TABLE SELECTION IS:  0  2  3  4



          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  00000000

        FOR UNITS:   3   4   5
THE FUNCTIONS ARE:   1   1   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                    F F F F T
                    F F F F F
                    T F F F F
                    F F T F F
                    F F F T F

  FINAL REGISTER CONTENTS:
010B0108  00000000
```

```
      THE HARDWARE DATA PATH IS COMPOSED OF:


NO OF UNITS=  5   TOTAL REGISTERS=  2   SYSTEM COST=  49
WT AVE EXEC TIME=    47     CYCLE TIME=  47


DATA PATH UNIT NUMBER:  1  2  3  4  5
UNIT TYPE NUMBERS ARE:  1  1  2  3  4

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)


                  F F F F T
                  F F F F F
                  T F F F F
                  F F T F F
                  F F F T F
```

Algorithm and Hardware Libraries for Example 5.2, Case 2

```
05
AOP0 1  AOP1 1  AOP2 1  AOP3 1  AOP4 1
04
HOP0 1  HOP1 1  HOP2 1  HOP3 1
05                                  NO OF ALGORITHMS
A(1,8):AOP0(B(1,8))(1,8) 1          IST ALGORITHM
A:HOP0(B)(1,8)
A(1,8):AOP1(B(1,8))(1,8) 1          2ND ALG
A:HOP1(B)(1,8)
A(1,8):AOP2(B(1,8))(1,8) 1          3RD ALG
A:HOP2(B)(1,8)
A(1,8):AOP3(B(1,8))(1,8) 1          4TH ALG
A:HOP3(B)(1,8)
A(1,8):AOP4(B(1,8))(1,8) 3          5TH ALG
C(1,8):AOP1(B)(1,8) 000
D(1,8):AOP2(C)(1,8) 100
A:AOP3(D)(1,8) 110
04                                  NO OF HARDWARE UNITS
P1                                  NO OF P PORTS
  8                                 WIDTH
Q1                                  NO OF Q PORTS
  8                                 WIDTH
COST=  2 DELAY=  2 FUNCS= 2
:HOP0(Q1)(1,8)                      FUNCTIONS
P1:HOP0(1,8)
P1                                  SAME FOR UNIT 2
  8
Q1
  8
COST= 10 DELAY= 10 FUNCS= 1
P1:HOP1(Q1)(1,8)
P1                                  SAME FOR UNIT 3
  8
Q1
  8
COST= 15 DELAY= 15 FUNCS= 1
P1:HOP2(Q1)(1,8)
P1                                  SAME FOR UNIT 4
  8
Q1
  8
COST= 20 DELAY= 20 FUNCS= 1
P1:HOP3(Q1)(1,8)
```

GH Table for Example 5.2, Case 2

```
  1 ENTRIES FOR OPERATOR 1
 2      1  0  0  0  0  0  0  0    1  0  0  0
  1 ENTRIES FOR OPERATOR 2
10      2  0  0  0  0  0  0  0    0  1  0  0
  1 ENTRIES FOR OPERATOR 3
15      3  0  0  0  0  0  0  0    0  0  1  0
  1 ENTRIES FOR OPERATOR 4
20      4  0  0  0  0  0  0  0    0  0  0  1
  1 ENTRIES FOR OPERATOR 5
45      5  2  3  4  0  0  0  0    0  1  1  1
```

Architecture for Example 5.2, Case 2

```
01                                   NO OF REGISTERS
  8                                  WIDTH
0                                    NO OF P PORTS
0
50                                   COST LIMIT
01                                   NO OF INSTRUCTIONS
 01 01 90                            STMNTS,WTFCT,MAXTM
R1:AOP4(R1)(1,8)
```

Results for Example 5.2, Case 2

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****


FOR ARCHITECTURE OPERATORS:  1  2  3  4  5
 THE GH TABLE SELECTION IS:  0  0  0  0  5



          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  00000000

        FOR UNITS:   3   4   5
THE FUNCTIONS ARE:   1   1   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                   F F F F T
                   F F F F F
                   T F F F F
                   F F T F F
                   F F F T F


  FINAL REGISTER CONTENTS:
010B0108  00000000
```

```
      THE HARDWARE DATA PATH IS COMPOSED OF:

NO OF UNITS=  5   TOTAL REGISTERS=  2   SYSTEM COST=  49
WT AVE EXEC TIME=    47      CYCLE TIME=  47

DATA PATH UNIT NUMBER:  1  2  3  4  5
UNIT TYPE NUMBERS ARE:  1  1  2  3  4

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)

                  F F F F T
                  F F F F F
                  T F F F F
                  F F T F F
                  F F F T F
```

Algorithm and Hardware Libraries for Example 5.2, Case 3

```
05
AOP0 1  AOP1 1  AOP2 1  AOP3 1  AOP4 1
04
HOP0 1  HOP1 1  HOP2 1  HOP3 1
06
A(1,8):AOP0(B(1,8))(1,8) 1         1ST ALGORITHM
A:HOP0(B)(1,8)
A(1,8):AOP1(B(1,8))(1,8) 1         2ND ALG
A:HOP1(B)(1,8)
A(1,8):AOP2(B(1,8))(1,8) 1         3RD ALG
A:HOP2(B)(1,8)
A(1,8):AOP3(B(1,8))(1,8) 1         4TH ALG
A:HOP3(B)(1,8)
A(1,8):AOP4(B(1,8))(1,8) 3         5TH ALG
C(1,8):AOP1(B)(1,8) 000
D(1,8):AOP2(C)(1,8) 100
A:AOP3(D)(1,8) 110
A(1,8):AOP4(B(1,8))(1,8) 3         6TH ALG
C(1,8):AOP1(B)(1,8) 000
D(1,8):AOP1(C)(1,8) 100
A:AOP2(D)(1,8) 110
04                                 NO OF HARDWARE UNITS
P1                                 NO OF P PORTS
  8                                WIDTH
Q1                                 NO OF Q PORTS
  8                                WIDTH
COST=  2 DELAY=  2 FUNCS= 2
:HOP0(Q1)(1,8)                     FUNCTIONS
P1:HOP0(1,8)
P1                                 SAME FOR UNIT 2
  8
Q1
  8
COST= 10 DELAY= 10 FUNCS= 1
P1:HOP1(Q1)(1,8)
P1                                 SAME FOR UNIT 3
  8
Q1
  8
COST= 15 DELAY= 15 FUNCS= 1
P1:HOP2(Q1)(1,8)
P1                                 SAME FOR UNIT 4
  8
Q1
  8
COST= 20 DELAY= 20 FUNCS= 1
P1:HOP3(Q1)(1,8)
```

GH Table for Example 5. 2, Case 3

```
  1 ENTRIES FOR OPERATOR 1
 2      1  0  0  0  0  0  0  0   1  0  0  0
  1 ENTRIES FOR OPERATOR 2
10      2  0  0  0  0  0  0  0   0  1  0  0
  1 ENTRIES FOR OPERATOR 3
15      3  0  0  0  0  0  0  0   0  0  1  0
  1 ENTRIES FOR OPERATOR 4
20      4  0  0  0  0  0  0  0   0  0  0  1
  2 ENTRIES FOR OPERATOR 5
45      5  2  3  4  0  0  0  0   0  1  1  1
35      6  2  2  3  0  0  0  0   0  2  1  0
```

Architecture for Example 5. 2, Case 3

```
01                                    NO OF REGISTERS
  8                                   WIDTH
0                                     NO OF P PORTS
0
50                                    COST LIMIT
01                                    NO OF INSTRUCTIONS
 01 01 90                             STMNTS,WTFCT,MAXTM
R1:AOP4(R1)(1,8)
```

Results for Example 5.2, Case 3
(first iteration)

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****


FOR ARCHITECTURE OPERATORS:  1  2  3  4  5
 THE GH TABLE SELECTION IS:  0  0  0  0  6



          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  00000000  00000000  00000000  00000000
00000000  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   1   1   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                   F F F F F F F F F T
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   T F F F F F F F F F
                   F F F F F F F T F F
                   F F F F F F F F T F

  FINAL REGISTER CONTENTS:
010B0108  00000000  00000000  00000000  00000000
00000000  00000000
```

```
      THE HARDWARE DATA PATH IS COMPOSED OF:

NO OF UNITS= 10   TOTAL REGISTERS=  7   SYSTEM COST=  49
WT AVE EXEC TIME=    37      CYCLE TIME=  37

DATA PATH UNIT NUMBER:  1  2  3  4  5  6  7  8  9 10
UNIT TYPE NUMBERS ARE:  1  1  1  1  1  1  1  2  2  3

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)

                    F F F F F F F F F T
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    T F F F F F F F F F
                    F F F F F F F T F F
                    F F F F F F F F T F
```

Results for Example 5.2, Case 3
(second iteration)

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****

FOR ARCHITECTURE OPERATORS:  1  2  3  4  5
 THE GH TABLE SELECTION IS:  0  0  0  0  6


          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  00000000  00000000  00000000  00000000
00000000  00000000  00000000  00000000  00000000

        FOR UNITS:  11  12
THE FUNCTIONS ARE:   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                   F F F F F F F F F F T F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   F F F F F F F F F F F F
                   T F F F F F F F F F F F
                   F F F F F F F F F F F F
          CYCLE NUMBER  2


INITIAL REGISTER CONTENTS:
02660108  00000000  00000000  00000000  00000000
00000000  00000000  00000000  00000000  00000000

        FOR UNITS:  11  12
THE FUNCTIONS ARE:   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                     F F F F F F F F F F T F
                     F F F F F F F F F F F F
                     F F F F F F F F F F F F
                     F F F F F F F F F F F F
                     F F F F F F F F F F F F
                     F F F F F F F F F F F F
                     F F F F F F F F F F F F
                     F F F F F F F F F F F F
                     F F F F F F F F F F F F
                     F F F F F F F F F F F F
                     T F F F F F F F F F F F
                     F F F F F F F F F F F F
          CYCLE NUMBER  3


INITIAL REGISTER CONTENTS:
02670108  00000000  00000000  00000000  00000000
00000000  00000000  00000000  00000000  00000000

       FOR UNITS:  11  12
THE FUNCTIONS ARE:  0   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                    F F F F F F F F F F F T
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    F F F F F F F F F F F F
                    T F F F F F F F F F F F

 FINAL REGISTER CONTENTS:
010B0108  00000000  00000000  00000000  00000000
00000000  00000000  00000000  00000000  00000000
```

```
     THE HARDWARE DATA PATH IS COMPOSED OF:

NO OF UNITS= 12   TOTAL REGISTERS= 10   SYSTEM COST=  45
WT AVE EXEC TIME=    51      CYCLE TIME=  17

DATA PATH UNIT NUMBER:  1  2  3  4  5  6  7  8  9 10 11 12
UNIT TYPE NUMBERS ARE:  1  1  1  1  1  1  1  1  1  1  2  3

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)

                  F F F F F F F F F F T T
                  F F F F F F F F F F F F
                  F F F F F F F F F F F F
                  F F F F F F F F F F F F
                  F F F F F F F F F F F F
                  F F F F F F F F F F F F
                  F F F F F F F F F F F F
                  F F F F F F F F F F F F
                  F F F F F F F F F F F F
                  F F F F F F F F F F F F
                  T F F F F F F F F F F F
                  T F F F F F F F F F F F
```

Algorithm and Hardware Libraries
for Example 5.3, Cases 1 and 2

```
04
AOP0 1  AOP1 1  AOP2 1  AOP3 1
04
HOP0 1  HOP1 1  HOP2 1  HOP3 1
04                                  NO OF ALGORITHMS
A(1,8):AOP0(B(1,8))(1,8) 1          IST ALGORITHM
A:HOP0(B)(1,8)
A(1,8):AOP1(B(1,8))(1,8) 1          2ND ALG
A:HOP1(B)(1,8)
A(1,8):AOP2(B(1,8))(1,8) 1          3RD ALG
A:HOP2(B)(1,8)
A(1,8):AOP3(B(1,8))(1,8) 1          4TH ALG
A:HOP3(B)(1,8)
04                                  NO OF HARDWARE UNITS
P1                                  NO OF P PORTS
  8                                 WIDTH
Q1                                  NO OF Q PORTS
  8                                 WIDTH
COST=  2 DELAY=  2 FUNCS= 2
:HOP0(Q1)(1,8)                      FUNCTIONS
P1:HOP0(1,8)
P1                                  SAME FOR UNIT 2
  8
Q1
  8
COST= 10 DELAY= 10 FUNCS= 1
P1:HOP1(Q1)(1,8)
P1                                  SAME FOR UNIT 3
  8
Q1
  8
COST= 15 DELAY= 15 FUNCS= 1
P1:HOP2(Q1)(1,8)
P1                                  SAME FOR UNIT 4
  8
Q1
  8
COST= 20 DELAY= 20 FUNCS= 1
P1:HOP3(Q1)(1,8)
```

GH Table for Example 5.3, Cases 1 and 2

```
 1 ENTRIES FOR OPERATOR 1
 2      1  0  0  0  0  0  0  0    1  0  0  0
 1 ENTRIES FOR OPERATOR 2
10      2  0  0  0  0  0  0  0    0  1  0  0
 1 ENTRIES FOR OPERATOR 3
15      3  0  0  0  0  0  0  0    0  0  1  0
 1 ENTRIES FOR OPERATOR 4
20      4  0  0  0  0  0  0  0    0  0  0  1
```

Architecture for Example 5.3, Case 1

```
02                                  NO OF REGISTERS
  8  8                              WIDTHS
0                                   NO OF P PORTS
0                                   NO OF Q PORTS
50                                  COST LIMIT
02                                  NO OF INSTRUCTIONS
 02 08 90                           STMNTS,WTFCT,MAXTM
A(1,8):AOP1(R1)(1,8) 00
R1:AOP2(A)(1,8) 10
 01 02 90                           STMNTS,WTFCT,MAXTM
R2:AOP3(R1)(1,8)
```

Architecture for Example 5.3, Case 2

```
02                                  NO OF REGISTERS
  8  8                              WIDTHS
0                                   NO OF P PORTS
0                                   NO OF Q PORTS
50                                  COST LIMIT
02                                  NO OF INSTRUCTIONS
 02 02 90                           STMNTS,WTFCT,MAXTM
A(1,8):AOP1(R1)(1,8) 00
R1:AOP2(A)(1,8) 10
 01 08 90                           STMNTS,WTFCT,MAXTM
R2:AOP3(R1)(1,8)
```

Results for Example 5.3, Case 1

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****

FOR ARCHITECTURE OPERATORS:  1  2  3  4
 THE GH TABLE SELECTION IS:  0  2  3  4


          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  01020108

        FOR UNITS:   3   4   5
THE FUNCTIONS ARE:   1   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                   F F F T F
                   F F F F F
                   T F F F F
                   F F T F F
                   F F F F F

  FINAL REGISTER CONTENTS:
010B0108  01020108
```

```
          THE MICROCODE FOR INSTRUCTION  2 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  01020108

        FOR UNITS:   3   4   5
THE FUNCTIONS ARE:   0   0   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                   F F F F F
                   F F F F T
                   F F F F F
                   F F F F F
                   T F F F F

  FINAL REGISTER CONTENTS:
01010108  010C0108




        THE HARDWARE DATA PATH IS COMPOSED OF:


NO OF UNITS=  5   TOTAL REGISTERS=  2   SYSTEM COST=  49
WT AVE EXEC TIME=   270     CYCLE TIME=  27


DATA PATH UNIT NUMBER:  1  2  3  4  5
UNIT TYPE NUMBERS ARE:  1  1  2  3  4

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)


                   F F F T F
                   F F F F T
                   T F F F F
                   F F T F F
                   T F F F F
```

Results for Example 5.3, Case 2

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****


FOR ARCHITECTURE OPERATORS:  1  2  3  4
 THE GH TABLE SELECTION IS:  0  2  3  4



          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  01020108

        FOR UNITS:   3   4   5
THE FUNCTIONS ARE:   1   0   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                   F F T F F
                   F F F F F
                   T F F F F
                   F F F F F
                   F F F F F
          CYCLE NUMBER  2


INITIAL REGISTER CONTENTS:
02290108  01020108

        FOR UNITS:   3   4   5
THE FUNCTIONS ARE:   0   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                     F F F T F
                     F F F F F
                     F F F F F
                     T F F F F
                     F F F F F

  FINAL REGISTER CONTENTS:
010B0108  01020108
```

```
          THE MICROCODE FOR INSTRUCTION  2 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  01020108

        FOR UNITS:   3   4   5
THE FUNCTIONS ARE:   0   0   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                  F F F F F
                  F F F F T
                  F F F F F
                  F F F F F
                  T F F F F

  FINAL REGISTER CONTENTS:
01010108  010C0108




      THE HARDWARE DATA PATH IS COMPOSED OF:


NO OF UNITS=  5   TOTAL REGISTERS=  2   SYSTEM COST=  49
WT AVE EXEC TIME=   264     CYCLE TIME=  22


DATA PATH UNIT NUMBER:  1  2  3  4  5
UNIT TYPE NUMBERS ARE:  1  1  2  3  4

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)


                  F F T T F
                  F F F F T
                  T F F F F
                  T F F F F
                  T F F F F
```

Algorithm and Hardware Libraries for Example 5.3, Cases 1, 2, and 3

```
03                                       NO OF ARCH OPS
AOP0 1  AOP1 1  AOP5 2
03                                       NO OF HARD OPS
HOP0 1  HOP1 1  HOP5 2
03                                       NO OF ALGORITHMS
A(1,8):AOP0(B(1,8))(1,8) 1               ALG 1
A:HOP0(B)(1,8)
A(1,8):AOP1(B(1,8))(1,8) 1               ALG 2
A:HOP1(B)(1,8)
A(1,8):AOP5(B(1,8),C(1,8))(1,8) 1             ALG 3
A:HOP5(B,C)(1,8)
03                                       NO OF UNITS
P1                                       UNIT 1, NO OF PORTS
  8                                      WIDTH
Q1                                       NO OF Q PORTS
  8                                      WIDTH
COST=  2 DELAY=  2 FUNCS= 2
:HOP0(Q1)(1,8)
P1:HOP0(1,8)
P1                                       SAME FOR UNIT 2
  8
Q2
  8  8
COST= 10 DELAY= 10 FUNCS= 1
P1:HOP5(Q1,Q2)(1,8)
P1                                       SAME FOR UNIT 3

Q1
  8
COST= 15 DELAY= 15 FUNCS= 1
P1:HOP1(Q1)(1,8)
```

GH Table for Example 5.4, Cases 1,2, and 3

```
   1 ENTRIES FOR OPERATOR 1
  2     1  0  0  0  0  0  0  0    1  0  0
   1 ENTRIES FOR OPERATOR 2
 15     2  0  0  0  0  0  0  0    0  0  1
   1 ENTRIES FOR OPERATOR 3
 10     3  0  0  0  0  0  0  0    0  1  0
```

Architecture for Example 5.4, Case 1

```
02                                    NO OF REGISTERS
  8  8
0                                     NO OF P PORTS
0                                     NO OF Q PORTS
30                                    COST LIMIT
01                                    NO OF INSTRUCTIONS
 03 01 90                             STMNTS,WTFCT,MAXTM
A(1,8):AOP5(R1,R2)(1,8) 000
R1:AOP5(R1,A)(1,8) 100
R2:AOP1(R2)(1,8) 000
```

Architecture for Example 5.4, Case 2

```
02                                             NO OF REGISTERS
  8   8
0                                              NO OF P PORTS
0                                              NO OF Q PORTS
35                                             COST LIMIT
01                                             NO OF INSTRUCTIONS
 03 01 90                                      STMNTS,WTFCT,MAXTM
A(1,8):AOP5(R1,R2)(1,8) 000
R1:AOP5(R1,A)(1,8) 100
R2:AOP1(R2)(1,8) 000
```

Architecture for Example 5.4, Case 3

```
02                                             NO OF REGISTERS
  8   8
0                                              NO OF P PORTS
0                                              NO OF Q PORTS
40                                             COST LIMIT
01                                             NO OF INSTRUCTIONS
 03 01 90                                      STMNTS,WTFCT,MAXTM
A(1,8):AOP5(R1,R2)(1,8) 000
R1:AOP5(R1,A)(1,8) 100
R2:AOP1(R2)(1,8) 000
```

Results for Example 5.4, Case 1

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****

               NO SATISFACTORY IMPLEMENTATION
```

Results for Example 5.4, Case 2

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****


FOR ARCHITECTURE OPERATORS:  1  2  3
 THE GH TABLE SELECTION IS:  0  2  3



          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  01020108  00000000  00000000  00000000


      FOR UNITS:   6   7
THE FUNCTIONS ARE:   1   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                    F F F F F F F
                    F F F F F F T
                    F F F F F T F
                    F F F F F F F
                    F F F F F F F
                    T T F F F F F
                    F T F F F F F
          CYCLE NUMBER  2

INITIAL REGISTER CONTENTS:
01010108  010C0108  02290108  00000000  00000000


        FOR UNITS:   6   7
THE FUNCTIONS ARE:   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                    F F F F F T F
                    F F F F F F F
                    F F F F F F F
                    F F F F F F F
                    F F F F F F F
                    T F T F F F F
                    F F F F F F F

  FINAL REGISTER CONTENTS:
010B0108  010C0108  00000000  00000000  00000000




      THE HARDWARE DATA PATH IS COMPOSED OF:


NO OF UNITS=  7   TOTAL REGISTERS=  5   SYSTEM COST=  35
WT AVE EXEC TIME=    34     CYCLE TIME=  17


DATA PATH UNIT NUMBER:  1  2  3  4  5  6  7
UNIT TYPE NUMBERS ARE:  1  1  1  1  1  2  3

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)


                    F F F F F T F
                    F F F F F F T
                    F F F F F T F
                    F F F F F F F
                    F F F F F F F
                    T T T F F F F
                    F T F F F F F
```

Results of Example 5.4, Case 3
(first iteration)

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****

FOR ARCHITECTURE OPERATORS:  1  2  3
 THE GH TABLE SELECTION IS:  0  2  3



          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  01020108

        FOR UNITS:   3   4   5
THE FUNCTIONS ARE:   1   1   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                    F F F T F
                    F F F F T
                    T T F F F
                    T F T F F
                    F T F F F

  FINAL REGISTER CONTENTS:
010B0108  010C0108
```

```
      THE HARDWARE DATA PATH IS COMPOSED OF:


NO OF UNITS=  5   TOTAL REGISTERS=  2   SYSTEM COST=  39
WT AVE EXEC TIME=    22      CYCLE TIME=  22


DATA PATH UNIT NUMBER:  1  2  3  4  5
UNIT TYPE NUMBERS ARE:  1  1  2  2  3

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)


                 F F F T F
                 F F F F T
                 T T F F F
                 T F T F F
                 F T F F F
```

Results for Example 5.4, Case 3
(second iteration)

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****


FOR ARCHITECTURE OPERATORS:  1  2  3
 THE GH TABLE SELECTION IS:  0  2  3



          THE MICROCODE FOR INSTRUCTION  1 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010108  01020108  00000000  00000000  00000000
00000000  00000000

        FOR UNITS:   8   9
THE FUNCTIONS ARE:   1   1

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                         F F F F F F F F F
                         F F F F T F F F T
                         F F F F F F F T F
                         F F F F F F F F F
                         F F F F F F F F F
                         F F F F F F F F F
                         F F F F F F F F F
                         T T F F F F F F F
                         F T F F F F F F F
          CYCLE NUMBER  2


INITIAL REGISTER CONTENTS:
01010108  010C0108  02290108  00000000  00000000
00000000  00000000

        FOR UNITS:   8   9
THE FUNCTIONS ARE:   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                     F F F F F F F T F
                     F F F F F F F F F
                     F F F F F F F F F
                     F F F F F F F F F
                     F F F F F F F F F
                     F F F F F F F F F
                     F F F F F F F F F
                     T F T F F F F F F
                     F F F F F F F F F

  FINAL REGISTER CONTENTS:
010B0108  010C0108  00000000  00000000  00000000
00000000  00000000
```

```
      THE HARDWARE DATA PATH IS COMPOSED OF:


NO OF UNITS=  9   TOTAL REGISTERS=  7   SYSTEM COST=  39
WT AVE EXEC TIME=    34      CYCLE TIME=  17


DATA PATH UNIT NUMBER:  1  2  3  4  5  6  7  8  9
UNIT TYPE NUMBERS ARE:  1  1  1  1  1  1  1  2  3

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)


                        F  F  F  F  F  F  F  T  F
                        F  F  F  F  T  F  F  F  T
                        F  F  F  F  F  F  F  T  F
                        F  F  F  F  F  F  F  F  F
                        F  F  F  F  F  F  F  F  F
                        F  F  F  F  F  F  F  F  F
                        F  F  F  F  F  F  F  F  F
                        T  T  T  F  F  F  F  F  F
                        F  T  F  F  F  F  F  F  F
```

Algorithm and Hardware Libraries for Example 5.5

```
10                                          NUMBER OF ARCH OPS
MOVE 1   OPDC 1   ADD  2   INC1 1   INC2 1   MRD  1   MWRT 2
ADNX 1   ADPR 1   VAL  1
07                                          NUMBER OF HARD OPS
ZOVE 1   ZPDC 1   ZADD 2   ZNC1 1   ZMAD 1   ZWRT 1   ZRD  1
11                                          NO OF ALGORITHMS
B(1,16):MOVE(A(1,16))(1,16) 1               1ST ALG
B:ZOVE(A)(1,16)
B(1,6):OPDC(A(1,4))(1,6) 1                  2ND ALG
B:ZPDC(A)(1,6)
B(1,16):ADD(A(1,16),C(1,16))(1,16) 1    3RD ALG
B:ZADD(A,C)(1,16)
B(1,16):ADD(A(1,16),C(1,16))(1,16) 1    4TH ALG
B:ZADD(C,A)(1,16)
B(1,12):INC1(A(1,12))(1,12) 1               5TH ALG
B:ZNC1(A)(1,12)
B(1,12):INC2(A(1,12))(1,12) 2               6TH ALG
Z(1,12):INC1(A(1,12))(1,12) 00
B:INC1(Z)(1,12) 10
B(1,16):MRD(A(1,12))(1,16) 2                7TH ALG
:ZMAD(A)(1,12) 00
B:ZRD(1,16) 10
:MWRT(A(1,12),B(1,16)) 2                    8TH ALG
:ZMAD(A(1,12)) 00
:ZWRT(B(1,12)) 10
B(1,12):ADNX(A(1,12)) 1                     9TH ALG
B:MRD(A)(1,12)
B(1,12):ADPR(A(1,12)) 2                     10TH ALG
Z(1,12):INC1(A)(1,12) 00
B:MRD(Z)(1,16) 10
B(1,16):VAL(A(1,12))(1,16) 2                11TH ALG
Z(1,12):INC2(A)(1,12) 00
B:MRD(Z)(1,16) 10
04                                          NO OF UNITS
P1                                          NO OF P PORTS
 16                                         WIDTH
Q1                                          NO OF Q PORTS
 16                                         WIDTH
COST=  2 DELAY=  1 FUNCS= 2
:ZOVE(Q1)(1,16)                             FUNCTIONS
P1:ZOVE(1,16)
```

```
P1                                       SAME FOR UNIT 2
 16
Q2
 16 16
COST= 15 DELAY=  5 FUNCS= 3
P1:ZADD(Q1,Q2)(1,16)
P1:ZADD(Q2,Q1)(1,16)
P1:ZNC1(Q1)(1,12)
P1                                       SAME FOR UNIT 3
 16
Q2
 12 16
COST= 20 DELAY=  4 FUNCS= 3
:ZMAD(Q2)(1,12)
:ZWRT(Q1)(1,16)
P1:ZRD(1,16)
P1                                       SAME FOR UNIT 4
  6
Q1
  4
COST=  5 DELAY=  5 FUNCS= 1
P1:ZPDC(Q1)(1,6)
```

GH Table for Example 5.5

```
 1 ENTRIES FOR OPERATOR 1
1     1  0  0  0  0  0  0  0    1  0  0  0
 1 ENTRIES FOR OPERATOR 2
5     2  0  0  0  0  0  0  0    0  0  0  1
 1 ENTRIES FOR OPERATOR 3
5     3  0  0  0  0  0  0  0    0  1  0  0
 1 ENTRIES FOR OPERATOR 4
5     5  0  0  0  0  0  0  0    0  1  0  0
 1 ENTRIES FOR OPERATOR 5
10    6  5  5  0  0  0  0  0    0  2  0  0
 1 ENTRIES FOR OPERATOR 6
8     7  0  0  0  0  0  0  0    0  0  2  0
 1 ENTRIES FOR OPERATOR 7
8     8  0  0  0  0  0  0  0    0  0  2  0
 1 ENTRIES FOR OPERATOR 8
8     9  7  0  0  0  0  0  0    0  0  2  0
 1 ENTRIES FOR OPERATOR 9
13   10  5  7  0  0  0  0  0    0  1  2  0
 1 ENTRIES FOR OPERATOR10
18   11  6  5  5  7  0  0  0    0  2  2  0
```

Architecture for Example 5.5

```
06             AC=R1,DA=R2,IC=R3,IR=R4,XR=R5,STATUS=R6
 16 12 12 16 12 06
0                                NO OF P PORTS
01                               NO OF Q PORTS
  8                              WIDTH OF Q1
54                               COST LIMIT
 7                               NO OF INSTS
 04 10 90                        INST 1, INST FETCH
Z(1,16):MRD(R3)(1,16) 0000
R3:INC1(R3)(1,12) 1000
R6:OPDC(Z(1,4))(1,6) 1000
R4:MOVE(Z)(1,16) 1000
 02 05 90                        INST 2, INDEXED LOAD
Z(1,12):ADD(R4(5,12),R5)(1,12) 00
R1:MRD(Z)(1,16) 10
 03 05 90                        INST 3, INDEXED ADD
Z1(1,12):ADD(R4(5,12),R5)(1,12) 000
Z2(1,16):MRD(Z1)(1,16) 100
R1:ADD(R1,Z2)(1,16) 110
 01  4 90                        INST 4, STORE
:MWRT(R4(5,12),R1)(1,16)               *
 03  3 90                        INST 5, GET NEXT ELEM
Z(1,12):ADNX(R2)(1,12) 000
R1:VAL(Z)(1,16) 100
R2:MOVE(Z)(1,12) 100
 03  2 90                        INST 6,GET PREV ELEM
Z(1,12):ADPR(R2)(1,12) 000
R1:VAL(Z)(1,16) 100
R2:MOVE(Z)(1,12) 100
 01  1 90                        INST 7, WRITE I/O
Q1:MRD(R4(5,12))(8,8)                  *
```

Results for Example 5.5

```
**** RESULTS OF THE DESIGN PROCESS ON THIS ITERATION ****

FOR ARCHITECTURE OPERATORS:  1  2  3  4  5  6  7  8  9 10
 THE GH TABLE SELECTION IS:  1  2  3  4  0  6  7  8  9 10


          THE MICROCODE FOR INSTRUCTION  1 :

          CYCLE NUMBER  1

INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:  8   9  10
THE FUNCTIONS ARE:  0   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F T F F F F F F F
                    F F F F F F F F F F
          CYCLE NUMBER  2


INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  00000000  0105010C
00000000  00000000

        FOR UNITS:    8   9  10
THE FUNCTIONS ARE:    3   3   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F T F F
                    F F F F F F F F T F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F T F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
          CYCLE NUMBER  3


INITIAL REGISTER CONTENTS:
01010110  0102010C  010D010C  02290110  0105010C
00000000  00000000

        FOR UNITS:     8    9   10
THE FUNCTIONS ARE:     0    0    1

                     (SOURCE=COLUMN; SINK=ROW)


                     F  F  F  F  F  F  F  F  F  F
                     F  F  F  F  F  F  F  F  F  F
                     F  F  F  F  F  F  F  F  F  F
                     F  F  F  F  F  F  F  F  F  F
                     F  F  F  F  F  F  F  F  F  F
                     F  F  F  F  F  F  F  F  F  T
                     F  F  F  F  F  F  F  F  F  F
                     F  F  F  F  F  F  F  F  F  F
                     F  F  F  F  F  F  F  F  F  F
                     F  F  F  T  F  F  F  F  F  F

  FINAL REGISTER CONTENTS:
01010110  0102010C  010D010C  02290110  0105010C
01100106  00000000
```

```
          THE MICROCODE FOR INSTRUCTION  2 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   1   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F T T F F F F F
                    F F F F F F F T F F
                    F F F F F F F F F F
          CYCLE NUMBER  2

INITIAL REGISTER CONTENTS:
00000000  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:    8   9  10
THE FUNCTIONS ARE:    0   3   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                    F F F F F F F F T F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F

  FINAL REGISTER CONTENTS:
010B0110  0102010C  0103010C  01040110  0105010C
01060106  00000000
```

```
          THE MICROCODE FOR INSTRUCTION  3 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   1   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F T T F F F F F
                    F F F F F F F T F F
                    F F F F F F F F F F
          CYCLE NUMBER  2


INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   1   3   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                   F F F F F F F T F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   T F F F F F F F T F
                   F F F F F F F F F F
                   F F F F F F F F F F

  FINAL REGISTER CONTENTS:
010B0110  0102010C  0103010C  01040110  0105010C
01060106  00000000
```

```
          THE MICROCODE FOR INSTRUCTION  4 :

          CYCLE NUMBER  1

INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

       FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   0   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F T F F F F F F
                    F F F F F F F F F F
          CYCLE NUMBER  2

INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   0   2   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     T F F F F F F F F F
                     F F F F F F F F F F

  FINAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000
```

```
          THE MICROCODE FOR INSTRUCTION  5 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   0   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F T F F F F F F F F
                   F F F F F F F F F F
          CYCLE NUMBER  2


INITIAL REGISTER CONTENTS:
00000000  00000000  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   3   3   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                    F F F F F F F T F F
                    F F F F F F F F T F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F T F
                    F F F F F F F F F F
                    F F F F F F F F F F
          CYCLE NUMBER  3


INITIAL REGISTER CONTENTS:
0267010C  02290110  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:    8   9  10
THE FUNCTIONS ARE:    3   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   T F F F F F F F F F
                   F F F F F F F T F F
                   F F F F F F F F F F
          CYCLE NUMBER  4

INITIAL REGISTER CONTENTS:
00000000  02290110  0103010C  01040110  0105010C
01060106  00000000

         FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   0   3   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                    F F F F F F F F T F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F

  FINAL REGISTER CONTENTS:
010B0110  02290110  0103010C  01040110  0105010C
01060106  00000000
```

```
          THE MICROCODE FOR INSTRUCTION  6 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:    8   9  10
THE FUNCTIONS ARE:    3   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                      F F F F F F F F F F
                      F F F F F F F F F F
                      F F F F F F F F F F
                      F F F F F F F F F F
                      F F F F F F F F F F
                      F F F F F F F F F F
                      F F F F F F F F F F
                      F T F F F F F F F F
                      F F F F F F F T F F
                      F F F F F F F F F F
```

```
           CYCLE NUMBER  2

INITIAL REGISTER CONTENTS:
00000000  00000000  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:    8   9  10
THE FUNCTIONS ARE:    3   3   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                   F F F F F F F T F F
                   F F F F F F F F T F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F T F
                   F F F F F F F F F F
                   F F F F F F F F F F
```

```
          CYCLE NUMBER  3


INITIAL REGISTER CONTENTS:
026A010C  02290110  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   3   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   F F F F F F F F F F
                   T F F F F F F F F F
                   F F F F F F F T F F
                   F F F F F F F F F F
          CYCLE NUMBER  4


INITIAL REGISTER CONTENTS:
00000000  02290110  0103010C  01040110  0105010C
01060106  00000000

         FOR UNITS:    8   9  10
THE FUNCTIONS ARE:    0   3   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                    F F F F F F F F T F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F

  FINAL REGISTER CONTENTS:
010B0110  02290110  0103010C  01040110  0105010C
01060106  00000000
```

```
          THE MICROCODE FOR INSTRUCTION  7 :


          CYCLE NUMBER  1


INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   0   1   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)

                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F F F F F F F F
                     F F F T F F F F F F
                     F F F F F F F F F F
          CYCLE NUMBER  2


INITIAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000

        FOR UNITS:   8   9  10
THE FUNCTIONS ARE:   0   3   0

THE GATING MATRIX:  (SOURCE=COLUMN; SINK=ROW)


                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F
                    F F F F F F F F F F

  FINAL REGISTER CONTENTS:
01010110  0102010C  0103010C  01040110  0105010C
01060106  00000000
```

```
      THE HARDWARE DATA PATH IS COMPOSED OF:

NO OF UNITS= 10   TOTAL REGISTERS=  7   SYSTEM COST=  54
WT AVE EXEC TIME=   800     CYCLE TIME=  10


DATA PATH UNIT NUMBER:  1  2  3  4  5  6  7  8  9 10
UNIT TYPE NUMBERS ARE:  1  1  1  1  1  1  1  2  3  4

THE DATA PATH CONNECTION MATRIX: (SOURCE=COLUMN; SINK=ROW)

                   F F F F F F F T T F
                   F F F F F F F F T F
                   F F F F F F F T F F
                   F F F F F F F F T F
                   F F F F F F F F F F
                   F F F F F F F F F T
                   F F F F F F F F F F
                   T T T T T F F F T F
                   T T T T F F F T F F
                   F F F T F F F F F F
```

REFERENCES

[1] Bell, C. G., and A. Newell , Computer Structures: Readings and Examples, McGraw-Hill Co., New York, 1970, Chapter 2.

[2] Bell, C. G., and A. Newell, "PMS and ISP Descriptive Systems for Computer Structures," AFIPS Conference Proceedings, vol. 36, May 1970, pp. 351-374.

[3] Breuer, M. A., "General Survey of Design Automation of Digital Computers," IEEE Proceedings, vol. 54, No. 12, Dec. 1966, pp. 1708-1721.

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

[5] Breuer, M. A., W. E. Donath, D. F. Gorman, J. M. Kurtzberg, and R. L. Russo, "Computer Design Automation; What Now and What Next," AFIPS Conference Proceedings, vol. 33, pt. 2, Dec. 1968, pp. 1499-1503.

[6] Chu, Y., "An ALGOL - Like Computer Design Language," Communications of the ACM, vol. 8, No. 10, Oct. 1965, pp. 607-615.

[7] Darringer, J. A., The Description, Simulation and Automated Implementation of Digital Computer Processors, PhD Dissertation, Carnegie-Melon University, Pittsburg, Pa., May 1969.

[8] Darringer, J. A., "A Language for the Description of Digital Computer Processors," report, Carnegie-Melon University, Pittsburg, Pa., May 1969.

[9] Dreyfus, S. E., "An Appraisal of Some Shortest-Path Algorithms," Operations Research, vol. 17, No. 3, May-June 1969, pp. 395-412.

[10] Duley, J. R. and D. L. Dietmeyer, "A digital Design System Language," IEEE Transactions on Computers, vol. EC-17, No. 9, Sept. 1968, pp. 850-861.

[11] Duley, J. R., and D. L. Dietmeyer, "Translation of a DDL Digital System Specification to Boolean Equations," IEEE Transactions on Computers, vol. C-18, No. 4, April 1969, pp. 305-318.

[12] Feldman J., and D. Gries, "Translator Writing Systems," Communication of the ACM, vol. 11, Feb. 1968, pp. 77-113.

[13] Friedman, T. D., "ALERT- A Program to Compile Logic Designs of New Computers," IEEE Digest on the 1st Annual IEEE Computer Conference, No. 16C51, Aug. 1967, pp. 128-130.

[14] Friedman, T. D., "Quality of Designs from an Automatic Logic Generator," IBM Research Report, R. C. 2068, April 1968.

[15] Friedman, T. D., "Methods Used in an Automatic Logic Design Generator (ALERT)," IEEE Transaction on Computers, vol. C-18, No. 7, July 1969, pp. 593-613.

[16] Gerace, G. B., "Digital Systems Design Automation," IEEE Transaction on Computers, vol. C-17, No. 11, Nov. 1968, pp. 1044-1061.

[17] Gorman, D. F., and J. P. Anderson, "A Logic Design Translator," AFIPS Conference Proceedings, Fall 1962, pp. 251-261.

[18] Gorman, D. F., "Systems Level Design Automation: A Progress Report on the Systems Descriptive Language (SLD II)," IEEE Digest of the 1st Annual IEEE Computer Conference, No. 16C51, Aug. 1967, pp. 131-134.

[19] Iverson, K. E., "A ProgrammingLanguage," AFIPS Conference Proceedings, Spring 1962, pp. 345-351.

[20] Iverson, K. E., "A Common Language for Hardware, Software, and Applications," AFIPS Conference Proceedings, Fall 1962, pp. 121-129.

[21] Lawler, E. L., "Notes on Combinatorial Optimization," University of Michigan, 1969.

[22] Metze, G., and S. Seshu, "Proposal for a Computer Compiler," AFIPS Conference Proceedings, Spring 1966, pp. 253-263.

[23] Proctor, R., "Logic Design Translator Experiment Demonstrating Relationship of Language to Systems and Logic Design," IEEE Transaction on Computers, vol. EC-13, August 1964, pp. 422-430.

[24] Schlaeppi, H. P., "Formal Language for Describing Machine Logic, Timing, and Sequencing (LOTIS)," IEEE Transactions on Computers, vol. EC-13, August 1964, pp. 439-448.

[25] Shorr, H., "Computer Aided Digital System Design and Analysis Using a Register Transfer Language," IEEE Transactions on Computers, vol. EC-13, Dec. 1964, pp. 730-737.

[26] Zucker, M. S., "LOCS - An EDP Machine Logic and Control Simulator," IEEE Transaction on Computers, vol. EC-14, June 1965, pp. 403-416.