The Design, Implementation, and Analysis of an Automated Logic Synthesis and Module Selection System

Gary W. Leive

January 1981

Department of Electrical Engineering
Carnegie-Mellon University
Pittsburgh, Pennsylvania 15213

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

This research was supported by National Science Foundation Grant No. ENG-78-25755.

Table of Contents

List of Figures

List of Tables

Acknowledgements

First and foremost, my profound thanks go to my wife Cora who made this effort possible, and to my son Eric whose act of birth early in this effort and cheerful presence since have made everything seem more worthwhile.

My sincere thanks go to my adviser, Dr. Donald Thomas whose patience, prodding, and ideas have gotten me to this point. His determined belief in an experimental approach was the seed that grew into the design experiment and the design space studies. Dr. Mario Barbacci provided numerous suggestions that improved the software design. His support in the development of the Synthesis Equivalence Language was particularly helpful. Dr. Daniel Siewiorek, and Dr. Charles Eastman have, through their diversified interests, significantly contributed to bringing together a thesis from work that ranged from design space exploration to databasing issues. Dr. Alice Parker provided insight on the relationship of the LSMS step to the D/M and Control allocation steps of CMU-DA.

A special thanks is due to my colleagues, Lou Hafer, Richard Cloutier, and Andy Nagle for implementations of the predecessor and successor steps of CMU-DA. Without Lou's Data/Memory Allocator there would not have been any functional level designs to process. Without Richard and Andy's Control Allocator the correctness of my output would have been far more suspect.

A special thanks is also due the "volunteer" designers who spent perfectly good drinking time helping me determine that this research had actually accomplished something.

My appreciation goes to the many members of the CMU-DA effort who have provided ideas and whose probing questions have made me address and solve problems it would have been all too easy to ignore.

Finally, appreciation must be expressed to Dr. Hans Berliner for providing the world's finest automated backgammon opponent which I found to be the ideal diversion at moments of high "research stress".

Abstract

The research reported in this thesis is an excursion into automation of the Logic Synthesis and Module Selection (LSMS) step of the CMU-DA system. A number of transformations that are necessary to manipulate the structure of a design (while keeping its behavior constant) are identified. A *surrogate designer*, with the judgment to apply transformations, has been implemented and is described.

The automated LSMS system was calibrated against two designs that were each hand processed with two different module sets: TTL modules and the Sandia Laboratories Standard CMOS Cells for LSI implementation. This calibration indicates that the automated system produces designs almost indistinguishable from those expected in a population of relatively good human designers. The calibration occurred toward the optimal end of a design space (minimum cost, delay, and power). The automated system was used to generate many different points throughout design spaces for three descriptions (each with the two different module sets). These design space projections are interesting since they imply a parabolic bound rather than the expected hyperbolic bound.

The design space data led to development of a set of *predictors* which can estimate the bounds of a design space from information that is available before the LSMS level. The estimates are useful for either hand design methods or higher levels of a design automation system. Use of the predictors is demonstrated by estimating the bounds of a design space for a fourth description. Plots of the measured design space and the estimates indicate that a good match was achieved.

Chapter 1
Introduction

"Stay at home in your mind. Don't recite other people's opinions. I hate quotations. Tell me what you know."

-- Ralph Waldo Emerson (1803 - 1877)

1.1 Motivation

During the past decade semiconductors have grown from SSI/MSI devices to the sixteen bit microprocessors available today. The focus of digital design has shifted from the board level to the integrated device level. However, the problems of dealing with a large design remain approximately the same as they were ten years ago. Initially, the desired behavior must be described. A behavioral description must be successively translated to the register transfer level, the functional logic level, the structural (or logic gate) level, and the circuit level. A design must be partitioned, physical positions must be selected for the devices, and the devices must be interconnected. Most of the progress in Computer Aided Design (CAD) to date has addressed the issues near the partitioning, placement, and routing end of the design hierarchy.

Today we stand at the threshold of the VLSI era. The continually increasing size of designs presents an almost intractable problem for unaided translation through the various design levels. However, the demand for VLSI devices requires that the design process be shortened to more quickly produce implementations that exhibit a specified behavior. These conflicting realities require that design aids address issues further up the design hierarchy than the current tools are able to handle.

The ultimate goal of design automation is to fully automate the translation steps starting

with a specification for a design and ending with a full set of production documents for implementation. Carnegie-Mellon University has been exploring a series of these translation steps with the goal of understanding the problems at each level well enough to formalize and implement the design tools. This thesis addresses the portion of the Carnegie-Mellon University Design Automation (CMU-DA) system concerned with the translation from the functional logic design level to the structural design level. The functional logic design level is represented by a data part graph (comprised of interconnected nodes representing registers, multiplexors, and ISPS[1] operators), plus control sequencing information. The nodes of the data part graph do not express or imply any information about devices that could be used for implementation. The structural logic design level is represented by a data part graph (comprised of interconnected nodes representing registers, multiplexors, and module set level operators) whose structure has been modified to allow implementation with devices from a specified module set, plus control sequencing information reflecting the structure imposed by the module set level operators.

1.2 Background

This research is a result of two projects at Carnegie-Mellon University: the EXPL project [Barbacci 73] which was the original CMU effort in design automation, and its direct descendant, the CMU-DA project [Siewiorek 76].

1.2.1 Prior Work

The EXPL system used a behavioral description as the design specification and a module set for implementation of the design. A design space was defined by specifying cost and speed attributes for the devices in the module set. The design space was first pruned with heuristics then explored (to the limits of EXPL's transformations) by altering the structure of the design. The design space parameters were changed by manipulating the serial and parallel relationships of the data paths. The EXPL system used the Digital Equipment Corporation (DEC) PDP-16 Register Transfer Modules (RTMs) as the initial module set and was later extended to use Macromodules [Clark 67].

---

[1] Supported by the Instruction Set Processor Specifications (ISPS) language [Barbacci 79].

1.2.2 Related Work

The CMU-DA project is a direct descendant of EXPL. It was conceived to explore and understand the manipulation of behavioral descriptions with the ultimate goal of minimizing the impact of changing technology. The module sets used by EXPL were convenient because they were well structured to provide a complete range of functionality and they both had consistent control protocols. Unfortunately, these module sets had almost no applicability outside of the educational environment. It was also quite difficult to change module sets because EXPL contained all module information within the program. In order to insure technology relevance it was necessary to devise a design automation system that would be far less sensitive to changes in the module sets than EXPL proved to be.

Two of the key concepts embodied in the organization of CMU-DA have been used to minimize the impact of changing hardware technology:

- Design decisions requiring knowledge of the hardware used for implementation are delayed as long as possible.
- Module sets used to provide the hardware dependent information are in databases that may easily be extended and exchanged.

The functional completeness and well structured control protocols of RTMs and Macromodules would have to be abandoned for the evolutionary module sets that are used in real designs. In this thesis, a module set is rather liberally interpreted to be any accumulation of digital devices that are designed to be directly interconnected without additional interfaces.

Figure 1-1 shows the organization of the CMU-DA system. The behavioral description of a design is written in the Instruction Set Processor Specification (ISPS) language [Barbacci 79] and translated into a parse tree (named the Global Data Base) form. The ISPS translator is independent of CMU-DA and is designed to provide an output that can be used for several purposes such as behavioral simulation and Design Automation [Barbacci 81].

The initial step of CMU-DA performs global optimization on the design. These operations were originally conceived by [Snow 78] and are currently being extended by [McFarland 79]. The global optimizations explore the cost/speed tradeoffs between the control and data parts of a design. The Global Data Base (GDB) form used as input is translated into a data structure called a Value Trace (VT). The VT form is particularly well suited to optimizations such as code motion, constant folding, dead code elimination, and redundant subexpression elimination because the nodes of the VT represent values rather than the carriers (registers, memories, or buses) in which values reside.

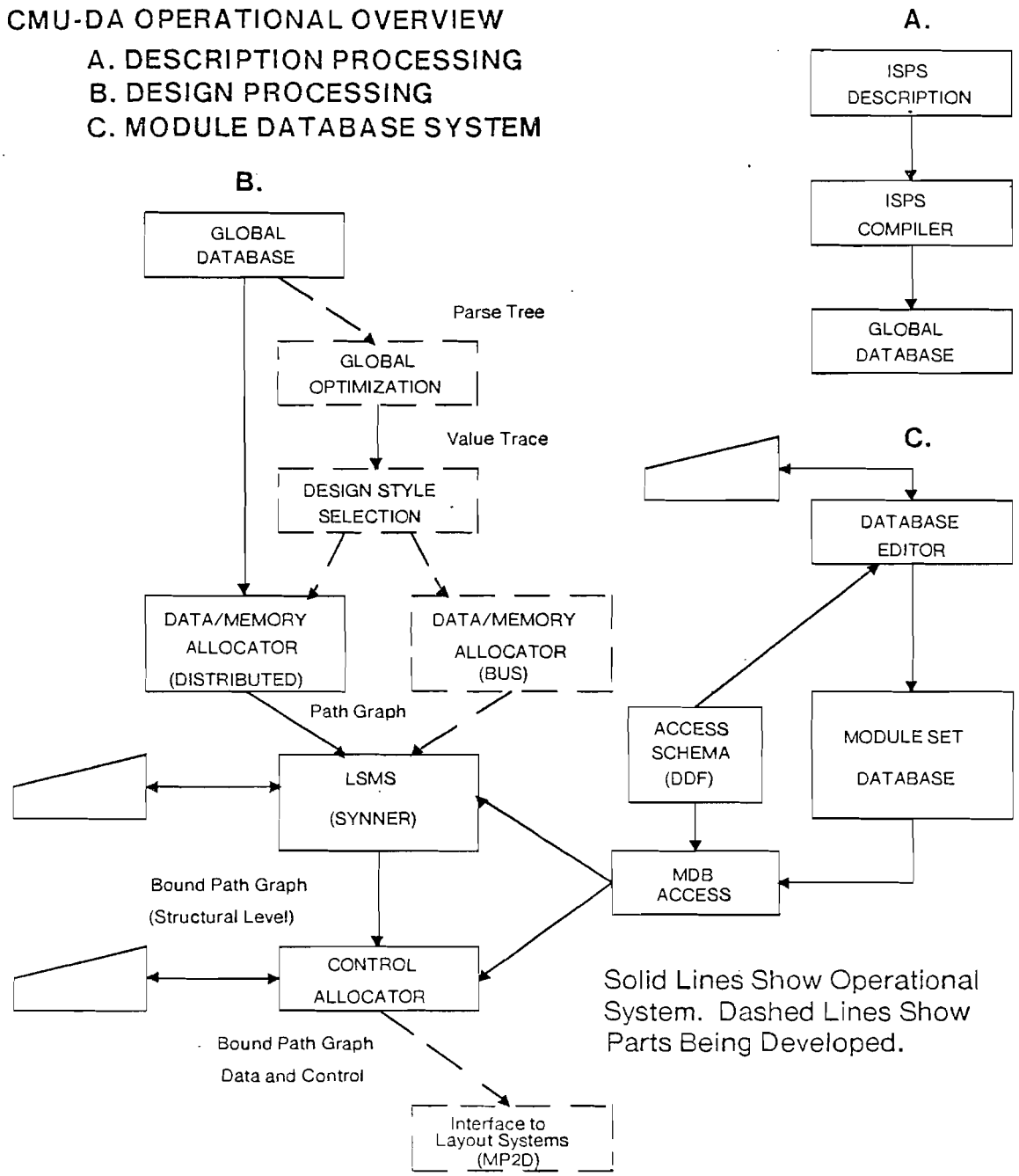


Figure 1-1: The CMU-DA System

A globally optimized design is passed to the Design Style Selection step [Thomas 81, Lawson 78]. Design Style Selection classifies a design into a category that will allow the best implementation. Design Styles that have been identified are:

- Distributed (random logic)
- Bus
- Microprocessors
- Bit-Slice Microprocessor
- Pipeline Processing

The classification is done by making measurements on a design (number of registers, number of operators, estimate of maximum delay path, etc.) and evaluating the measurements against summary information from module sets that are candidates for implementing the design in each of the styles.

Once a design style has been identified, a design is routed to an appropriate Data/Memory Allocator. A Data/Memory Allocator translates a design from the abstract behavioral form to a functional logic level design called a **path graph**. Nodes of a path graph describe the variables, operators, switches, and data links. The data flow is described by directed edges. A path graph may describe any degree of complexity and may be non-planar, cyclic, and disconnected. Information provided with a path graph defines the control sequence required to activate the data paths. Although Data/Memory allocators for each design style are planned, the only operational one at this time is the Distributed Design Style allocator [Hafer 78].

To this point, a design has been processed without requiring information about the specific hardware that would be used to implement it. This means that a path graph remains a general enough form to be processed into implementations as diverse as board level (using TTL packages) or integrated circuits (using cells) by the remaining steps of CMU-DA. The Logic Synthesis and Module Selection (LSMS) step of CMU-DA performs the translation of a path graph into a design with the modules selected for each node of a design's data part. The LSMS step of CMU-DA is the topic of this thesis and a detailed discussion of the transformations will be deferred to the next chapter.

After the data part of a design is synthesized and modules have been selected for implementation, enough information is available to design a controller. This is done by a Control Allocator [Cloutier 80, Nagle 80]. The Control Allocator synthesizes a ROM based microprogrammed sequencer to activate the data part modules.

The result of processing by CMU-DA is a design in the form of a graph that contains completely specified data and control parts. Additional research [Parker 79, Kim 79] is investigating the problems of interfacing the output of CMU-DA to layout and routing systems such as the RCA/ARMY MP2D program.

1.2.3 Different Work

There are several approaches to Design Automation under active study by various universities and organizations. Each approach has merit and will add to the overall understanding of design automation problems. It is probable that future systems will use a mixture of the best results from various investigations in order to develop a comprehensive design automation tool.

Since CMU-DA represents just one of several partitions of design automation research, three of the alternative viewpoints will be summarized in order to show that this work is unique and that there is a need for several simultaneous investigations. The three design automation systems chosen to represent the range of current investigations are the California Institute of Technology Silicon Compiler, the Massachusetts Institute of Technology Design Procedure System, and the University of Kiel (West Germany) MIMOLA design system.

1.2.3.1 The CIT Silicon Compiler

The California Institute of Technology Silicon Compiler [Johannsen 79]is possibly the most highly publicized design automation research effort. The stated objective is to "produce an entire LSI mask set from a single page, high level description of the integrated circuit." The high level description appears to consist of a very structured expectation of the resulting chip. One section of the description defines the microcode word width and requires an identification of the fields within the word. A second section defines the data path word width and the device interconnection paths (required to be buses). The final section defines the elements in the data part of the design (specific devices are assigned with the understanding that the silicon compiler will handle placement and manipulation of the specific geometries).

There are several points in the silicon compiler philosophy that contrast with the CMU-DA philosophy. Probably the most apparent contrast is the disparity between the ideas of what constitutes a high level description. The ISPS language used with CMU-DA describes the desired *behavior* of a design as a set of register transfer statements. This approach does not impose any preconception about either the architecture or the technology used to implement the design. The silicon compiler expects that the data part will be in a well defined "core"

area of the chip, the communication between core elements will be by bus, and the control part will go in a well defined decoder on a specific area of the chip. CMU-DA defers decisions about the control structure (including microcode word width and field definitions) until data part synthesis is completed.

The silicon compiler research appears to overlap the lower ends of CMU-DA (particularly control allocation which, while more flexible, performs a similar function by producing microcode). The silicon compiler addresses the lower level issues of partitioning, layout, and interconnection while the CMU-DA approach is to manipulate a design through higher levels and interface the system output with other systems to perform partitioning, layout, and interconnection.

1.2.3.2 The MIT Design Procedure System

The Massachusetts Institute of Technology Artificial Intelligence Laboratory design automation investigation [Sussman 79] is addressing methods for aiding the development of very large designs. To this end, they have envisioned a system that can act as a designer's assistant. The system would take care of the many accounting details involved in developing a large design. Multilevel description consistency would be maintained, and the system would be able to advise a designer on the consequences of any proposed modifications. Formalization of the design process is a major subgoal of this study. To that end there is an effort to develop a performance theory of engineering design. This work is still in the very early stages, but a demonstration of the possibilities has been performed by constructing some specially tailored software to produce a LISP interpreter on a single LSI chip. The chip has actually been fabricated from the output of this demonstration system.

The MIT effort differs from CMU-DA in both concept and implementation. CMU-DA has the less ambitious goals of understanding the problems at several hierarchical levels of the digital design process and implementing software tools to deal with those problems. The MIT goal is to discover a formal theory of the entire engineering design process by evolving tools that both aid and monitor designer performance. The initial MIT efforts have been targeted at a level that overlaps the low end of CMU-DA and proceeds down to the actual chip layout issues. It remains to be seen just what results are produced by the MIT endeavor.

1.2.3.3 The MIMOLA Design System

The MIMOLA Design System [Zimmermann 79, Marwedel 79] appears to have more in common with CMU-DA than do the Software Compiler or the Design Procedure System. The data part and control part of a design are described at a behavioral level in the MIMOLA language. The system processes the description through several steps including syntax analysis, compilation, and allocation. A statistical analyzer provides information to a designer who must decide if (and how) the data part must be constrained in order to meet cost objectives. A restricted design may be reprocessed by MIMOLA. This iterative procedure may continue (with the designer performing an active role) until an optimum is reached.

MIMOLA does not attempt to deal with either the high level global optimizations or the data part synthesis that are integral parts of CMU-DA. It is primarily oriented toward automated operations on the control part of a design. The first pass through the system produces a design with the minimum number of control states. This approach often generates a very expensive hardware implementation. Designer restrictions to the data part may then be reprocessed to trade off control steps (and delay) for data part cost. MIMOLA does not appear to address any of the issues of this thesis. It assumes that the data part nodes can be implemented directly from available hardware modules. In cases where that is not true, the designer must intervene to design "super modules" that satisfy the description requirements.

1.3 The Problem

The functional logic level of a design is the starting point for this work. This level is represented as a path graph produced by a distributed data/memory allocator. No information about any physical hardware is expressed or implied by a design at this level. The operators reflect functions that are supported at the behavioral description (ISP) level. Functional logic operators may include any of the arithmetic modes (two's complement, one's complement, sign-magnitude, or unsigned) supported by ISP.

The goal of the research was to develop a design aid that solved the problem of translating the data part of a design from a functional logic level to a structural logic level with associated hardware information, and to use this design aid to investigate and better understand the design space of the structural logic level.

The objective of the translation to a structural logic level is the structural modification of a design to reflect correct implementation of all nodes with modules from a module set. The problem consists of identifying a set of transformations capable of operating on all path graph

node types. A methodology must then be devised to apply the transformations in an efficient manner.

Investigation of the design space associated with the structural logic level required that the transformations and their application be implemented as a programmed software system. It is possible to postulate an *absolute* design space that would be bound by the limits to which the parameters of the design could be changed. At this point in time, there is no known method to predict the absolute bounds. However, a *transformation* design space can be mapped. Once the software system existed, it was possible to vary the designs, the module sets, and the designer constraints to produce points in a design space. A design space may be drawn in one or more dimensions if changes in parameters (cost, delay, and power in this thesis) result from processing designs with different constraints and different sets of transformations.

This thesis proposes and implements a set of structural transformations. The ability of the transformations to produce a reasonable design space when compared to human designers is demonstrated, and those transformations are used to produce design spaces for several designs.

1.4 Approach

An attempt will be made in this section to state the underlying philosophies that guided development of the Logic Synthesis and Module Selection (LSMS) system implementation which is called SYNNER (SYNthesizing desigNER).

Initially ([Leive 77]) it was proposed to define a methodology for producing optimal designs at the LSMS level. The implementation envisioned at that time would have been fully automated. A functional level design would have entered the system and a structural level design with modules selected for each node would have been produced by the system. However, as the early software evolved, it became apparent that it was extremely useful to be able to apply transformations manually through keyboard interaction with the system. This observation led to the concept of a *design environment* where automatic processes would be a part of a larger capability to manipulate designs. Each of the capabilities available to the automatic LSMS operation is also available to the designer for manual application. In addition, a designer can control which transformations are applied during automatic processing. The designer has the tools to override decisions made by the automatic processes and to change module selections made either automatically or manually.

Every effort was made to allow easy extensions to both module information and synthesis

transformations. A designer should have the freedom and the capability to extend the information and the repertoire of algorithms that a design automation tool uses to make decisions. The extension capability should be external to the software. Programs should, in other words, be programmable. This principle was not fully achieved in the LSMS system. Instead, a reasonable compromise was achieved that reduced the number of hard coded transformations to a few that operate on fairly general aspects of the design structures. All other constructions required for LSMS are covered by the externally programmable capability that allows synthesis algorithms to be extended.

Transformations were defined to be as general as possible. At the LSMS design level, transformations used by designers often appear to be derived from a "bag of tricks". Rather than attempt to duplicate a special "bag of tricks" in software, the problem was viewed as one on which structure could be placed. A small number of very general transformations and a generalized synthesis procedure are adequate to produce good (not optimal) designs. There are special cases that are not efficiently dealt with, but it appears that structure has been place on the central issues of this level of design.

The design environment was built in a bottom up, evolutionary manner. Structure manipulation primitives were defined first. The primitive operations were then integrated into transformations. A controlling process was developed to act as a surrogate designer for automatic processing. The controlling process contains the capability to apply appropriate transformations, synthesize structures, and select the modules to implement the nodes of a design. This surrogate designer can apply the same structure manipulation transformations that are available to a human designer using the system in a manual mode.

The result of this approach is a design aid which may be used to further investigate the optimal automation of this level of design. The existing system can be extended somewhat through its external programming capability. Future research with this system could lead to the modification or redefinition of the surrogate designer.

1.5 Overview

The remainder of the thesis is organized as a progression starting with the details of the transformations and ending with the use of the system for design space explorations. Within that progression, there are two distinct groupings:

- Chapters 2 and 3 discuss the concepts involved in developing SYNNER as a software tool for use at the LSMS design level.

- Chapters 4 and 5 discuss calibration and applications of this tool.

Chapter 2 identifies and explains the automation of the basic structural transformations and the synthesis process. The transformations provide a raw capability for manipulating the structure of a design represented as a path graph. They are postulated to be a reasonable working set of transformations that are capable of handling most design situations. However, justification of that assumption will not be offered until Chapter 4. Chapter 3 describes the operation and implementation of the surrogate designer that controls automatic processing in SYNNER. The surrogate designer exercises the judgment necessary to apply the transformations much as a human designer would if the transformations were applied manually.

Chapter 4 presents the results of an experiment involving designers who manually processed descriptions at the LSMS level. Two descriptions (a change mechanism for use in vending machines and a truncated description of the PDP-8 minicomputer) and two module sets (TTL and the Sandia CMOS Cells [Sandia 78]) were processed by the designers. By comparing their results to the automated results for the same descriptions, it was possible to verify that SYNNER performs almost as well as a member of the human designer population. The experiment also provided information on transformations that designers actually use. The designers produced a few points near the optimal end of a design space. Chapter 5 builds from those few points by using the LSMS system to explore a much wider design space. Design space projections are plotted using data from processing three designs 64 times (each with different constraints) using two module sets for each design. The design space explorations provide a quantity of data that is reduced in order to arrive at a set of predictors that may be used to estimate the bounds of cost, delay, and power parameters for other designs.

Chapter 6 summarizes the results and identifies some areas for future research at the LSMS design level.

Chapter 2
Transformations for Logic Synthesis

"The art of progress is to preserve order amid change and to preserve change amid order".

-- Alfred North Whitehead

2.1 Introduction

Designs entering the LSMS step of CMU-DA are represented at a functional level which may contain operations that are not directly realizable from devices in a module set. In cases where there is not a match between members of the module set and operations in a design, some methodology must be applied to reduce the difference until the available modules can be used to implement the description.

The methodology used to implement the LSMS step of CMU-DA consists of transforming the structure of the functional level design until the resulting nodes only require operations that exist in a particular module set. The transformations used in the LSMS will be discussed in this chapter. They provide a capability for structural modification that may be used as a CAD tool with a designer deciding how they should be applied or a surrogate designer could be devised to automatically decide when and how to apply the transformations. The surrogate designer implemented in SYNNER will be discussed in Chapter 3.

2.2 Approach to Structure Transformation

In the early stages of this research consideration was given to transforming both the path graph and the module set. The path graph was to be modified such that the difference between the existing structure and the modules available to implement functions was reduced. The module set would then be transformed by synthesizing a construct (a super-module) that would again reduce the difference between the module set functions and the path graph structure. At some point the synthesized module construct and the path graph structure were supposed to converge. This approach was rejected primarily because convergence would be difficult to guarantee. The problem of devising meaningful measures of the difference between the graph structure and module constructs also made this approach appear impractical. Finally, the requirement for both graph structure and module set transformations would roughly double the effort required to implement this approach.

The method that was used only requires modifications to the structure of the path graph. The graph structure is transformed until it contains operations that can be implemented with existing modules.

The transformations are intimately associated with the path graph and micro-operation sequence, and a brief description of those data structures must be presented before the individual transformations can be discussed.

2.3 Data Part Representation

In 1977, Lou Hafer [Hafer 77] provided the key software tool that made this research and the rest of the CMU-DA implementation possible. That tool is a translator used to bridge the gap between the behavioral description produced by the ISP compiler [Barbacci 79] and the functional logic design level. The translator is called the Data/Memory (D/M) allocator for the distributed design style. This D/M allocator produces a functional logic level description of the data part of a design as a directed graph called a *data path graph* (or *path graph*). A control part representation is also produced by the D/M allocator and it will be described in Section 2.4.

Nodes of a path graph describe the variables, operators, switches, and data links. The data flow is described by directed edges. A path graph may describe any degree of complexity and may be non-planar, cyclic, and disconnected. Figures 2-1 and 2-2[2] show the basic node types that may be interconnected to form a path graph.

---

[2]Reproduced from [Hafer 77] by permission of the author.

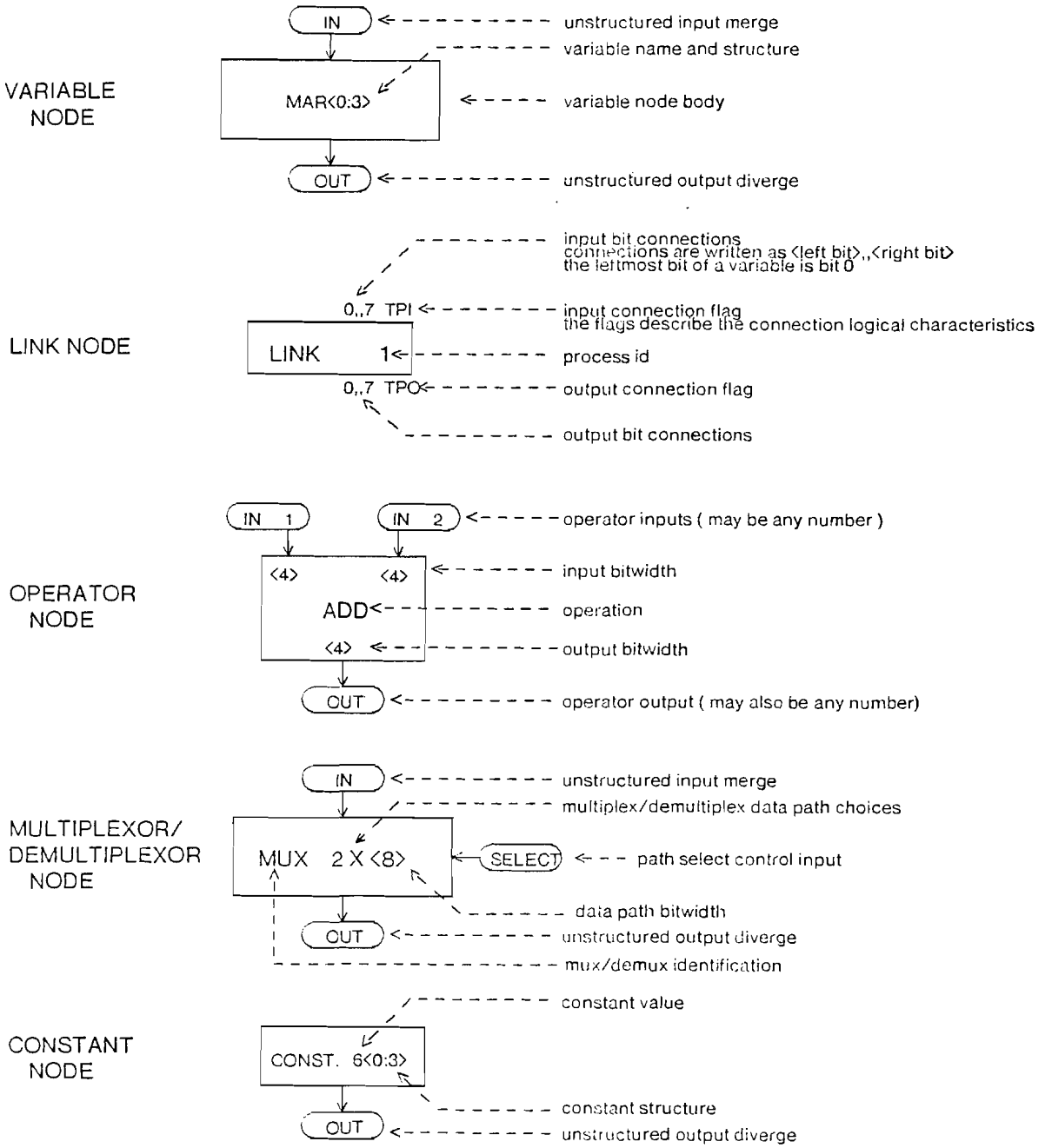
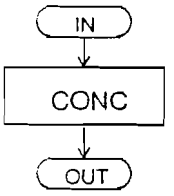
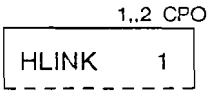


Figure 2-1: Basic Path Graph Nodes (a)

CONCATENATION NODE



HALF LINK NODE



used to specify one side of a connection for later use by the control allocator codes are identical to full links

CONNECTION FLAG CODE

C : complemented
T : true
P : parallel
S : serial
I : input
O : output

Figure 2-2: Basic Path Graph Nodes (b)

2.3.1 Variable Carriers (VC)

The Variable Carrier (VC) class of nodes includes registers, register arrays (memories), and constants. Registers and register arrays each have one data input and one or more outputs. Constants have no data inputs and one or more outputs.

Registers, register arrays, and constants all require a **bit range**. A bit range specifies a starting bit number and an ending bit number. The bit range is specified in the form:

```
<starting-bit:ending-bit>
```

Examples of registers with bit ranges are:

```
MAR<0:3>
PC<0:11>
```

In addition to the bit range, register arrays require a **word range** of the form:

```
[starting-word:ending-word]
```

A memory might be described as:

```
MP[0:4095]<0:7>
```

The following example shows the format of a typical VC Path Graph node in ASCII form:

```
REG     NODE ADDR.: #322        ASYTAB INDEX: #  30
```

The fields are interpreted as:

```
REG               => A register path graph node.
NODE ADDR.: #322  => The unique path graph node number.
ASYTAB INDEX:     => Allocator Symbol Table identifier.
#  30             => Unique symbol table node number.
```

The corresponding symbol table entry appears as follows:

```
#  30   REG             LAC<0:12>              READ  WRITE
        BITS:  13       NIBBLE SIZE:  13       LSHIFT  RSHIFT
        READ IN ICE'S:          PDP8
        WRITTEN IN ICE'S:       PDP8
```

The symbol table fields may be interpreted as follows:

```
#  30             => The unique symbol table node number.
LAC               => The carrier (register) name.
<0:12>            => Bit map showing a 13 bit register.
READ              => Register is read.
WRITE             => Register is written.
BITS:  13         => Register bit width.
NIBBLE SIZE: 13   => Smallest partition  read or written
                     is 13 bits.
```

```
LSHIFT            => Left shift mergered operator.
RSHIFT            => Right shift merged operator.
READ in ICE'S:    => Independent Control Environments
                     (ICE) where register is read.
PDP               => The name of the ICE where register
                     is read.
WRITTEN in ICE'S  => ICEs where register is written.
PDP               => The name of the ICE where register
                     is written.
```

2.3.2 Path Carriers (PC)

The Path Carrier class of nodes includes links and half-links. Path carriers do not store information, but direct its passage by specifying the direction of data flow, defining the logical sense of the inputs and outputs, and identifying bit mapping for the inputs and outputs. Links specify a complete connection between a predecessor node and a successor node. Links have exactly one input connection and one output connection. Half-links have an input connection, but no output destination. Half-links provide termination points for connections that will be used by the control allocator.

An example of a link Path Graph node is shown below:

```
LINK    NODE ADDR.: #154        ICE ID: #  0
SOURCE NODE: #   35  OUTPUT   1  BIT CONNECTIONS: 0:0
        CONNECTION FLAGS: TPO
DEST. NODE:  #   24  INPUT    1  BIT CONNECTIONS: 0:0
        CONNECTION FLAGS: TPI
```

The link node fields may be interpreted as:

```
LINK              => Link Path Graph node type.
NODE ADDR.: #154  => Unique Path Graph node number.
ICE ID: #  0      => Independent Control Environment
                     where the link appears.
SOURCE NODE: #35  => Path Graph node number that acts
                     as the source for the link.
OUTPUT 1          => Link connects to the first output
                     of the source node.
BIT CONNECTIONS:  => Identifier to indicate bit mapping
                     from the source.
0:0               => Connect to bit 0 (leftmost bit)
                     of the source.
CONNECTION FLAGS  => Identifier indicating source
                     connection attributes follow.
TPO               => True, Positive, Output source
                     connection attribute.
DEST. NODE: #24   => Path Graph node number that acts
```

```
                        as the destination for the link.
INPUT 1           =>    Link connects to the first input
                        of the destination node.
BIT CONNECTIONS:  =>    Identifier to indicate bit mapping
                        from the destination.
0:0               =>    Connect to bit 0 (leftmost bit)
                        of the destination.
CONNECTION FLAGS  =>    Identifier indicating destination
                        connection attributes follow.
TPI               =>    True, Positive, Input destination
                        connection attribute.
```

2.3.3 Variable Operators (VO)

The Variable Operator (VO) class of nodes is comprised of all logical, relational, and arithmetic data operators. These operators have one or more inputs (depending on their function) and they may have one or more outputs. The input and output bit widths are separately specified. Logical operators have an output bit width that is identical to the input bit widths. Arithmetic operators have outputs that are one bit larger than the input bit widths in order to capture carries. Relational operators have a single bit output that can be interpreted as a boolean.

An example of the ASCII form of a VO node is shown below:

```
OPER    NODE ADDR.: #337        ADD     OPTAB INDEX: #  35
```

These path graph node fields can be interpreted as:

```
OPER              =>  Operator type node.
NODE ADDR.:       =>  Node Address identifier.
#337              =>  The unique path graph node address.
ADD               =>  Unsigned addition operator.
OPTAB INDEX:      =>  Operator table index identifier.
# 35              =>  Node number of the corresponding
                      operator table entry.
```

The corresponding operator table entry is:

```
#  35   ARITH   ADD     BITWIDTH:   13   2 INPUT(S)   1 OUTPUT(S)
        INPUT BITWIDTH:    12   OUTPUT BITWIDTH:   13
```

The operator table fields can be interpreted as:

```
#  35             => Unique operator table node number.
ARITH             => Arithmetic operator class.
ADD               => Unsigned addition operator.
BITWIDTH:         => Identifier for operator bit width.
13                => This operator is 13 bits wide.
2 INPUT(S)        => The number of data inputs.
1 OUTPUT(S)       => The number of data outputs.
INPUT BITWIDTH:   => Identifier for data path input
                     bit width.
12                => 12 bit wide data path inputs.
OUTPUT BITWIDTH:  => Identifier for data path output
                     bit width.
13                => Output bit width includes carry.
```

2.3.4 Path Operators (PO)

The Path Operator (PO) class of nodes includes multiplexors and concatenations. Multiplexors may have two or more data inputs and one or more data outputs. Exactly one of the data inputs is selected and connected to the outputs at any time. The selection criteria is derived from the data part (terminating in a half-link) but the actual selection is the responsibility of the control allocator. Concatenations provide the mechanism for merging two or more data paths into a single data path. The bit width of a concatenation is the sum of the bit widths of all inputs.

An example of the ASCII representation for a PO path graph node is:

```
MUX     NODE ADDR.: #4     4 CHOICES OF   12 BITS
```

These path graph node fields can be interpreted as:

```
MUX               => Node operation identifier.
NODE ADDR.: #4    => Unique path graph node number.
4 CHOICES         => The number of selectable mux inputs.
OF 12 BITS        => The width of each selectable input.
```

The other PO node types have slightly different syntax since they do not have selects.

2.4 Control Part Representation

The control information for a design is provided in a form separate from the data path graph. The *micro-operation sequence* is a list of operations with sources and destinations. It is not unlike a high level programming language in that:

- It is a linearized form of the control flow for the design.
- It contains a mixture of data activation operations and sequence alteration operations.
- It implicitly specifies activation of the successor micro-operation sequence unless there is an explicit operation to alter control flow.
- It does not contain detailed information about the machine that it is controlling. Rather, it references the data part nodes that contain detailed information.

The overview presented here is designed only to provide enough of a description to allow a discussion of the impact of data part transformations on the micro-operation sequence. The functional level syntax for micro-operation sequence instructions produced by the D/M allocator is given in [Hafer 79]. Some syntax extensions required by structural level transformations are demonstrated in this chapter, but their complete definition is given in [Leive 80].

The general format for data micro-operations is:

<op>,<dest>,<source>;

Where:

<op>	::=	opcode
<dest>	::=	destination
<source>i	::=	$\langle source_i \rangle$, | $\langle source \rangle, \langle source_{i+1} \rangle$
$\langle source_1 \rangle$	::=	{first source}
$\langle source_{i+1} \rangle$	::=	{second, third, ... n^{th} source}
i	::=	1 | 2 | ... | n
n	::=	{any integer}

Variable carriers and variable operators enter micro-operations in slightly different manners that will be outlined separately.

2.4.1 VC Micro-Operations

Variable carriers occur in micro-operations as both sources and and destinations. An instruction to load a register (R1) with the contents of another register (R2) would take the form:

```
#210(MOVE),#1(R1),#2(R2):#6,;

Where:   #210     ::= Opcode for the ISP[3] MOVE instruction.
         (MOVE)   ::= Operator name.
         #1       ::= Node number of the destination register.
         (R1)     ::= Variable name of the destination register.
         #2       ::= Node number of the source register.
         (R2)     ::= Variable name of the source.
        :#6       ::= Node number of the link connecting R1 and R2.
```

2.4.2 VO Micro-Operations

Variable operators enter into the micro-operation sequence directly as operations or as sources for other operations. VOs never occur as destinations, only VCs may be destinations. Consider a micro-operation sequence that controls adding the contents of two registers (R1 and R2) and storing the result in a third register (R3):

```
#261(ADD2C):#4:#7,#1(R3),#2(R1):#5,#3(R2):#6;

Where:   #261     ::= Opcode for the ISP ADD2C instruction.
         (ADD2C)  ::= Operator name.
        :#4       ::= Node number of the operator.
        :#7       ::= Node number of the link connecting
                      ADD2C to R3.
         #1       ::= Node number of the destination register.
         (R3)     ::= Variable name of the destination register.
         #2       ::= Node number of the first source register.
         (R1)     ::= Variable name of the first source.
        :#5       ::= Node number of the link connecting
                      R3 to ADD2C.
         #3       ::= Node number of the second source register.
         (R2)     ::= Variable name of the second source.
        :#6       ::= Node number of the link connecting
                      R2 to ADD2C.
```

A path graph contains both the data part description of a design (at a functional logic level) and controlling information to specify the activation sequence of data part operations. The preceding discussions of the path graph and the micro-operations are not detailed, but they

---

[3]This Opcode is a member of ISPS codes considered to be archaic by ISPS maintainers but still in use for CMU-DA

should provide sufficient information to proceed to the descriptions of the structure transformations.

2.5 Partitioning Transformations

If certain types of nodes could be subdivided (partitioned) it would become possible to match the resulting nodes to the limitations of physical devices. For example, if the bit width of a register node was not an even multiple of the number of bits in a register module, partitioning might be desirable. Partitioning could divide the register node into one section that would be evenly covered by the bit width of a selected module and another section that could be used to select a module with a smaller bit width. In the PDP-8, the Link/Accumulator is a 13 bit register. When selecting modules from the TTL module set, candidates consist of four bit registers and one bit registers. Under certain constraints (i.e. cost minimization), the desired mixture would be to implement the Link/Accumulator with three packages of four bit registers and one package of a one bit register.

There are two categories of partition that may be treated separately: bit boundary partitioning and input boundary partitioning. Registers and operators generally have a predictable number of data inputs that is required by their function. The hardware modules that implement the behavior of registers and operators usually have the expected number of inputs. Therefore it is only necessary to be concerned with partitioning registers and operators along bit boundaries. Multiplexors have various numbers of inputs and it is necessary to partition them along input boundaries. Demultiplexors do not occur in a path graph and are not dealt with by the current implementation of SYNNER.

2.5.1 Bit Boundary Partitioning

The bit boundary partitioning of registers and operators appear to be similar enough that they can be treated together. There are certain instances where differences occur. These differences will be identified and alternate paths in the transformations will be discussed.

The bit boundary partitioning transformation essentially splits a node into two sections and requires a knowledge of the node type, the desired bit boundary for partitioning, and the initial bit width of a node. The first step is to incarnate a new node of the same type. The new node will be called the **right part** and the original node will be called the **left part**. If the original node had data inputs **n** bits wide, and the partitioning boundary **b** was indicated, the left part would become **(n - b)** bits wide and the right part would be **b** bits wide.

A concatenation node is incarnated for registers and arithmetic or logical operators to provide a connection point for output links that are different widths than either the left or right parts. The concatenation node is connected with links to both the left and right parts. All outputs links that are the identical bit width of the left part remain connected to it. All output links that are the identical bit width of the right part are moved so that they connect to it. All other output links are moved so that they connect to the concatenation. Relational operators always generate a single bit boolean output regardless of the input bit width. To "concatenate" the outputs of the left and right parts of a partitioned relational operator it is necessary to AND the resulting outputs and move all output links to the incarnated AND operator.

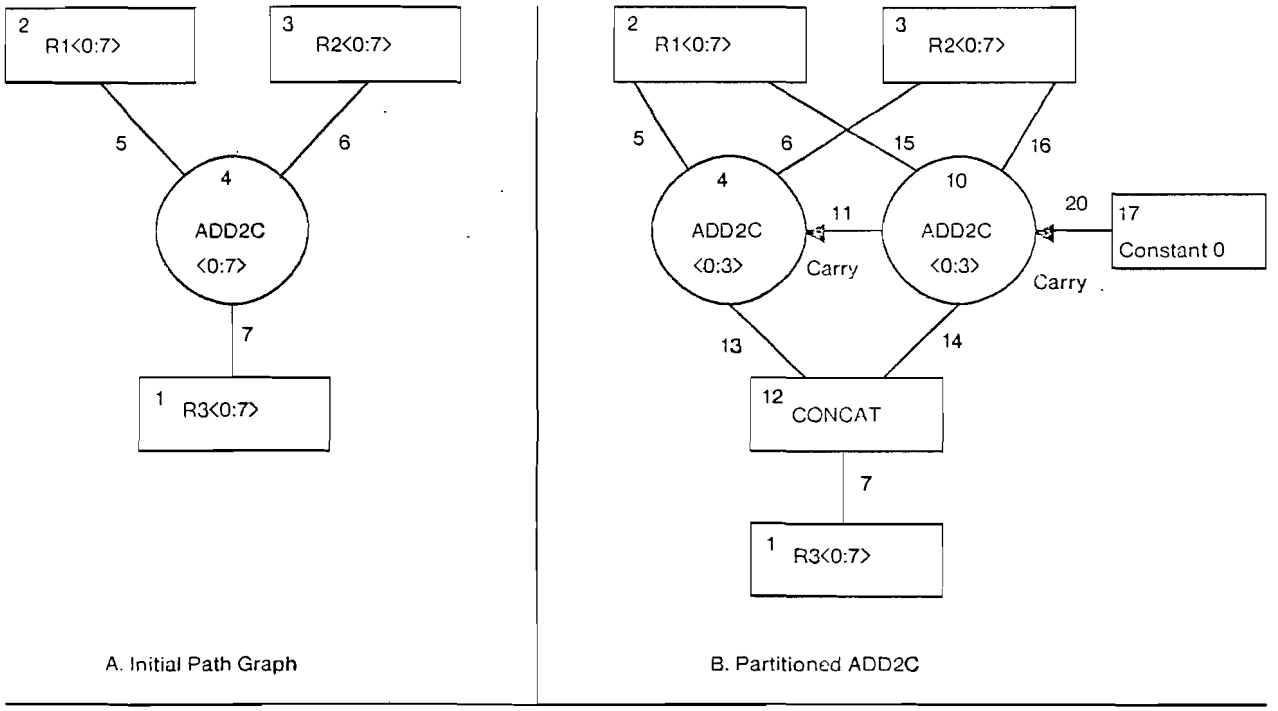
Input links are constructed to the right part from all nodes that connected to the inputs of the original node. The original links remain connected to the left part. Their bit width is adjusted to match the new width of the left part. In the case of arithmetic operators, a carry link is connected between the right and left parts. This is a single bit link with a special flag identifying its carry function. Registers nodes that are required to perform shift, rotate, increment, or decrement operations are connected with links to allow shifting, rotating, or carries.

Figure 2-3 shows the progressive complexity that occurs when splitting certain nodes of a path graph. In this example, the path graph represents the addition of the contents of two registers (R1 and R2) and storing the result in a third register (R3). The adder is first split into two parts (Figure 2-3b), then one of the source registers (R2) is split (Figure 2-3c). When the adder is split, an explicit carry (Link #11) is generated. A constant zero (Node #17) is also tied to the carry input of Node #10. A concatenation (Node #12) is generated to merge the data paths from the two adders. Links (with four bit mappings) are routed from R1 and R2 to the two adders. When R2 is split, there is relatively little change in this example. The register is divided into a left part and a right part. In this case there is no need for concatenation since the existing links (#6 and #16) match the input bit widths of the destination adders and have the correct mapping. If R2 had been split on another boundary (i.e. a left part of one bit and a right part of 6 bits), a concatenation would have been necessary. If R2 had included merged operators for shift, rotate, increment, or decrement, there would have been carry or shift links included by the partitioning transformation.

The original micro-operation sequence for this path graph is the same as the example in Section 2.4.2:

```
#261(ADD2C):#4:#7,#1(R3),#2(R1):#5,#3(R2):#6;
```

After the adder is split, the micro sequence takes the form:



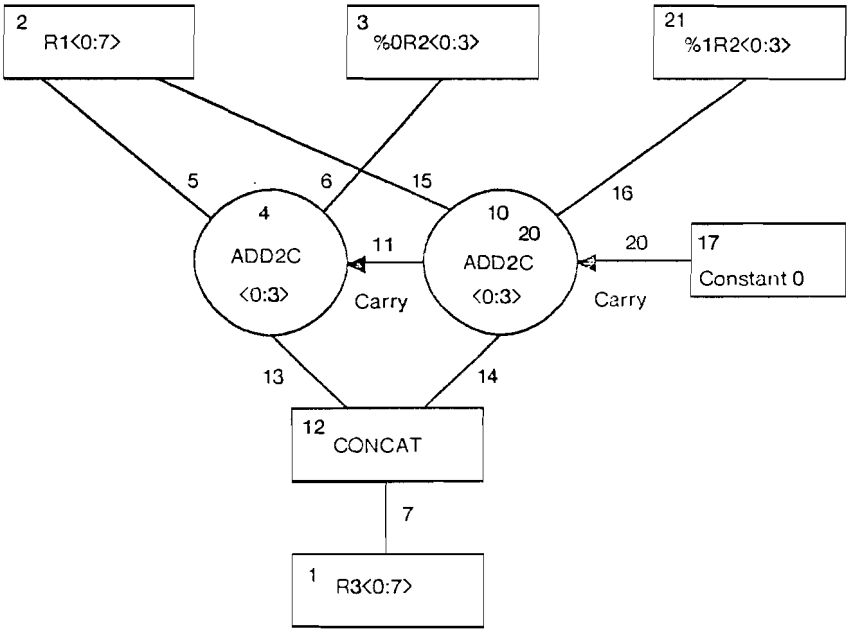


Figure 2-3: Bit Boundary Partitioning

```
#261(ADD2C):#4@#10:#13@#14,#1(R3),#2(R1):#5@#15,#3(R2):#6@#16;
```

The concatenated node numbers (#4@#10) identify the left and right parts of the split adder. The concatenated destination link numbers (#13@#14) identify the links joining the left part adder (#4) and the right part adder (#10) to R3. The concatenated source link numbers (#5@#15 and #6@#16) identify the link pairs from the sources to the right and left parts of the adder.

When the source register (R2) is split, the micro-operation sequence becomes:

```
#261(ADD2C):#4@#10:#13@#14,#1(R3),#2(R1):#5@#15,
            #3@#21(%0R2@%1R2):#6@#16;
```

Unique names are constructed for the right and left parts of R2 (%0R2 and %1R2). The ADD2C micro-operation sequence is changed to identify the node numbers of the two parts of R2.

2.5.2 Input Boundary Partitioning

Multiplexors are members of the class of nodes termed **path operators** (PO). They appear to require partitioning by the number of inputs rather than by the bit width of the input. The approach is similar to bit width partitioning, but the resulting structure is somewhat different. As in the case of bit width partition, the original node will be partitioned into a **left part** and **right part**. In this case, instead of a concatenation, there will be a joining multiplexor called the **bottom part**. If a **n** input node is partitioned at the i^{th} input, the left part will have (**i - 1**) inputs and the right part will have (**n - i**) inputs. The bottom part will have two inputs.

Figure 2-3 illustrates input boundary partitioning. Initially, there is a seven input multiplexor of arbitrary ("n" bit) bit width in a path graph. If partitioning were directed to occur at input five (5), the three multiplexor arrangement shown in the Figure 2-3b would result. The left part would retain the original node number (#1) but it would be reduced to four inputs. A right part node (Node #2) would be incarnated in the path graph and it would be linked to inputs 5, 6, and 7 of the original multiplexor. The inputs to Node #2 would be renumbered as 1, 2, and 3. A bottom part multiplexor (Node #3) would be generated to merge the right and left part multiplexors.

If a second partitioning were directed to split Node #2 at input three (Figure 2-3c), no right part multiplexor would be generated. Instead, the bottom part multiplexor (Node #3) would have its inputs increased to receive the single input (input 3). Note that this structure requires exactly the same components that would be required to implement the first partitioning.

If a third partitioning were directed divide Node #3 at the third input, the structure shown in Figure 2-3d would result. The process could be extended until only two input multiplexors would be required for implementing the original seven input multiplexor.

Viewing the progression of structures in Figure 2-3, it is clear that there are several considerations that must be weighed before directing a partitioning. The increasing depth of the structure adds delay to the design. In packaged module sets (such as TTL) cost evaluation must consider the package cost and the available spare (free) modules. It is possible that the original seven input multiplexor implemented with an eight input SN74151 might be both the cheapest and fastest implementation. Alternatively, if one or more spare SN74153 or SN74157 were available (from a previously mounted package), any of the other configurations might be the cheapest. The transformations do not specify modules (spares or otherwise), but they do establish conditions that favor certain selections. These types of considerations must be evaluated before initiating the input partitioning transformation. All evaluations are the responsibility of the designer if the system is used manually, or of the controlling software if the system is used in the automatic mode.

Multiplexors are not directly represented in the micro-operation list. Therefore, the micro-operations are not transformed during multiplexor input partitioning.

2.6 Combining Transformations

Two general classes of combining transformations have been identified. The *vertical join* transformation combines cascaded logical operators of a similar type. The *horizontal join* combines adjacent arithmetic or relational operators. of a similar type. These transformations appear to be quite useful in dealing with constructs that regularly occur in path graphs, but they could actually be viewed as special cases that could be included in a more generalized logic reduction capability.

2.6.1 Vertical Join Transformations

The vertical join transformation is used to merge cascaded logical operators of a similar type. An example of this situation would occur if two AND nodes (each with two inputs) were connected such that the output of one AND node (the **top part**) served as one of the inputs of the second AND node (the **bottom part**). The cascade of two operators can occur either during synthesis or directly from the functional logic level input.

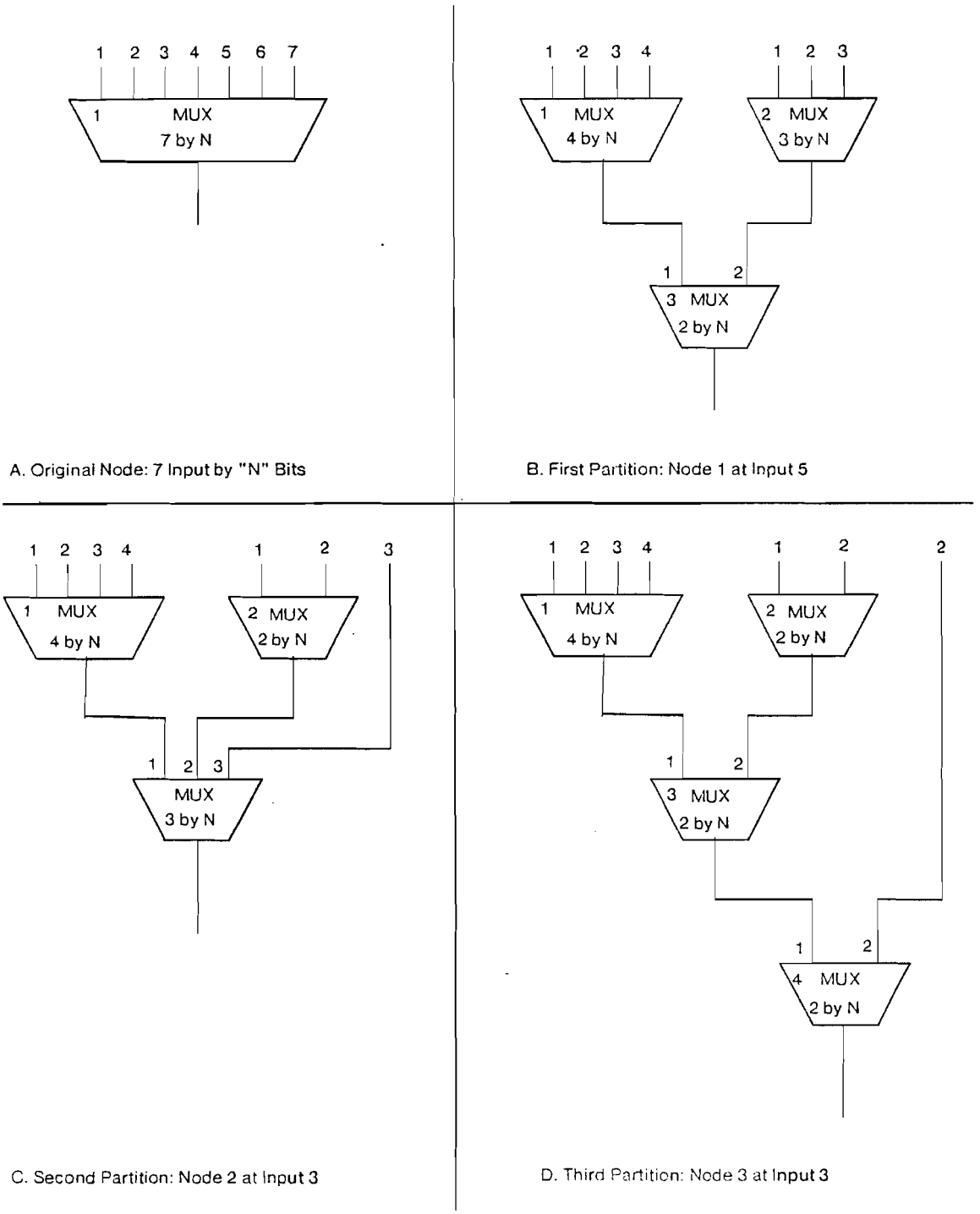


Figure 2-4: Input Boundary Partitioning

A pair of logical operators is defined to be similar if:

- Both operators are of identical types.
- The bottom part operator is a NAND and the top part operator is an AND.
- The bottom part operator is a NOR and the top part operator is an OR.

The vertical join transformation will not be performed unless the common link connecting the bottom part and the top part is the only output from the top part. The transformation moves all bottom part inputs that do not interconnect the two nodes to the top part. All outputs from the bottom part are moved to the top part. The link connecting the two nodes is deleted. The bottom part node is deleted.

The micro-operation sequence for cascaded operators is represented by two chained instructions. The transformation increases the number of inputs in the first micro-operation instruction, moves any destination information from the second instruction to the first instruction, and deletes the second instruction.

Figure 2-5a shows the example of a NAND node with two cascaded AND nodes. The vertical join transformation (applied to Nodes #11 and #14) will cause Node #14's inputs (and implicit inversion) to be assumed by Node #11. Node #11 will become a three input NAND as shown in Figure 2-5b. If the vertical join transformation were applied to Nodes #6 and #11 of Figure 2-3b, the single four input NAND node (#6) shown in Figure 2-3c would result. The original micro-operation sequence for the cascaded nodes is:

```
#235(AND):#6,,#1(D):#7,#2(C):#10;
#235(AND):#11,,#6(OPER):#12,#3(B):#13;
#237(NAND):#14,,#11(OPER):#15,#4(A):#16;
#210(MOVE),#5(E),#14(OPER):#17,;
```

After the first vertical join (applied to Nodes #11 and #14) the modified micro-operation sequence becomes:

```
#235(AND):#6,,#1(D):#7,#2(C):#10;
#237(NAND):#11,,#6(OPER):#12,#3(B):#13,#4(A):#16;
#210(MOVE),#5(E),#11(OPER):#17,;
```

The second application of vertical join (to Nodes #6 and #11) results in a micro-operation sequence that reflects the four input nature of the NAND node:

```
#237(NAND):#6,,#1(D):#7,#2(C):#10,#3(B):#13,#4(A):#16;
#210(MOVE),#5(E),#6(OPER):#17,;
```

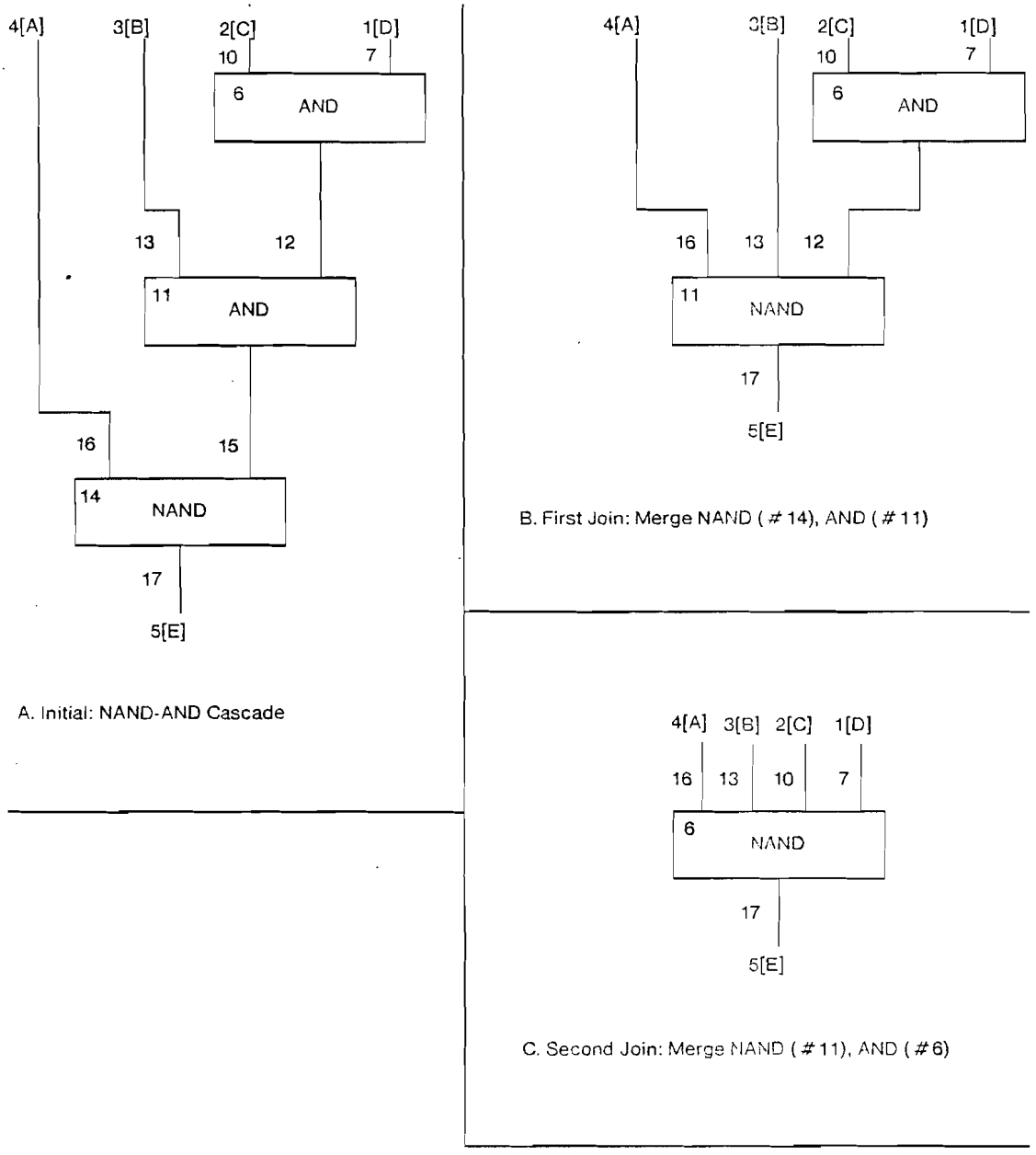


Figure 2-5: Combining AND Gates

2.6.2 Horizontal Join Transformations

Horizontal join transformations are used to combine arithmetic or relational operator nodes in order to eliminate duplicating hardware. The principle considerations for combining with a horizontal join are:

- The nodes must be in different control structures (they must never be required at the same time).
- The nodes must be of a type for which candidate modules exist after combination by a multifunction module (ALU) available from the database.
- There must be some benefit (such as reduction in cost) that requires consideration of a horizontal join.

If the conditions justify proceeding with a horizontal join, the **right part** characteristics and connections are mapped onto the **left part**:

- All **right part** output links are moved to the output of the **left part**. Any links that have the same destination and the same bit mapping as existing **left part** links are eliminated. Links that can be eliminated reduce the input requirements to other nodes (usually multiplexors) and further reduce the cost of the design.
- All **right part** input links are moved to the appropriate input of the **left part**. Input links are matched and redundant links are eliminated. Inputs to the **left part** are multiplexed if necessary.
- The Micro-operation Sequence Table is updated to reflect a change in node reference number for **right part** operations. No new micro-operations are created and no existing micro-operations are eliminated.
- The **right part** is deleted.

2.7 Equivalence Transformations

There are several ways to implement every logical. relational or arithmetic operation. It is possible, at one extreme, to implement every design using only NOR gates. Given today's MSI and LSI devices, this approach would be expensive because the low functionality of the devices would lead to a large number of packages and interconnections. At the other extreme, it would be possible to implement most designs using multifunction devices such as ALUs. Again, this approach would be expensive because the high functionality would be wasted in areas of the design where simple operations predominate. The large number of possibilities for implementing even the simplest functions poses a rather serious threat to including enough transformations in a program to implement general designs. An alternative

would be to select a usable subset of transformations to be included in a program. However, selecting a subset that would be complete enough to cover all designs might be difficult. It would also be difficult to include additional transformations because they would have to be coded, debugged, and inserted in the control structure of the program. Additionally, the transformations required by the same design are apt to change with different module sets. If the transformations could be altered or extended without too much difficulty, it would help insure that the transformations achieve "technological relevance" by making it easy to apply any new functions that are added to module sets.

In the LSMS system implementation, it was decided to abstract the equivalence transformation process by providing the knowledge of equivalences in an external database and programming the means of manipulating that knowledge. By abstracting the transformation process, the program would not need to know anything about the semantics of the equivalences, it would just need to understand how to apply a relatively small number of syntactic entities. The equivalences can exist in a format that is easy for designers to modify and is easy for the synthesis program to interpret. This approach appeared to be useful for the class of transformations that deal with specific operators.

2.7.1 Synthesis Equivalence Language

The form of representation for the external equivalences was a matter of some concern and consideration. The most attractive choice was some form of existing language definition such as ISPS or the Global Data Base (GDB) language. However, both of those languages are more general than were required for this research. The difficulties of constructing supporting software to translate ISPS or GDB descriptions into a form useful for this application precluded their use. A tree structured language named Synthesis Equivalence Language (SEL) was developed to satisfy both the requirements of designers and the requirements of SYNNER. A complete BNF of SEL is given in Appendix B. A brief description of the language features will be given here to provide a basis for discussing how these abstracted transformations are applied to a design.

The basic relationship defined by a SEL description is an *equivalence* stating that the right side of the description is equivalent to the left side. The right side of an equivalence consists of a *class* and a *classtype*. The left side consists of one or more reference line identifiers *lineid* and corresponding type equivalences *typeqv*. The basic syntax of an equivalence statement is:

```
CLASS:CLASSTYPE          *LINEID TYPEQV
```

An example of the equivalence for translating a two's complement relational equal to an unsigned equal would be:

```
RELAT:EQL2C              *1 EQL $1 $2
```

There are two inputs implicitly associated with the EQL2C and two inputs explicitly associated with the EQL. The explicit inputs are named "$1" and "$2". The "$" identifies the input as a source and the numbers indicate the position of the source on multi-input devices. A single output is implicit for EQL2C and is explicitly identified for the EQL as a termination on the line reference (*1).

SEL can specify most logical constructs through the use of features that allow multiple line equivalences, nonspecific repetition of inputs, and bit mapping. The following examples demonstrate the major SEL capabilities.

```
RELAT:LEQ2C     *1    SUB $1 $2
                *2    OR  *1<0> (NOR [*1<1>])
```

Where:

```
        RELAT    => Class (Relational).
        LEQ2C    => Classtype (two's complement
                    less-than-or-equal).
        *1       => Line 1 identifier.
        SUB      => Line 1 operator (unsigned subtract).
        $1       => First source input to SUB.
        $2       => Second source input to SUB.

        There is no explicit bit mapping in Line 1.  Input bit
        widths are derived from the LEQ2C operator.  The output
        bit width will be the input bit width + 1 to account
        for a carry out bit.

        The second line starts with the reference number:

        *2       => Line 2 identifier.
        OR       => Line 2 operator (Logical inclusive OR).
        *1<0>    => First source input to OR.
                    *1   => Reference to Line 1 (output of
                            of SUB as a source to the OR.
                    <0>  => Bit map, the leftmost bit (0)
                            is the single bit input.

        (        => Delimiter indicating the start of a nested
                    operation as the second source.
        NOR      => Operator.
        [*1<1>] => Repeated source reference.
                    Line 1 is the referenced output (from SUB).
                    Start with bit 1 of the output (leftmost bit
                    to the right of the carry) and connect
                    successive bits as inputs to NOR until
                    all Line 1 output bits are exhausted.
                    The number of inputs to the NOR is wholly
                    dependent on the size of the SUB operator.

        )        => Close nested source.
```

The following equivalence defines a different but equally valid form of LEQ2C.

```
RELAT:LEQ2C      *1     NOT (GTR2C $1 $2)
```

```
Where:
        RELAT    => Class (Relational).
        LEQ2C    => Classtype (2's complement less-than-or-equal).
        *1       => Line 1 identifier.
        NOT      => Operator.
        (        => Delimiter indicating the start of a nested
                    operation as the first (only) source.
        GTR2C    => Two's complement greater-than operator.
        $1       => First source to GTR2C.
        $2       => Second source to GTR2C.
        )        => Close nested operation delimiter.
```

The separate equivalences for LEQ2C demonstrated a number of SEL features including the basic prefix (operator-source-source) form, line references, nested operations as sources, and repeated bit sequences as sources. The following example shows a means of specifying an increment (INCR) operation and introduces the specification of constants in SEL.

```
ARITH:INCR       *1     ADD $1 #1
```

```
Where:
        ARITH    => Class (Arithmetic).
        INCR     => Classtype (unsigned increment).
        *1       => Line 1 identifier.
        ADD      => Line 1 operator (unsigned add).
        $1       => First source input to ADD.
        #1       => Constant of value 1 as the second
                    source input to ADD.
```

The final example demonstrates an equivalence for the two's complement subtract operation. This equivalence uses the complement and increment algorithm. A special qualifier "{CI}" is introduced to mark a source which is to be connected to an adder's carry input.

```
ARITH:SUB2C      *1     ADD $1 (NOT $2) #1<0>{CI}
```

```
Where:
        ARITH    => Class (Arithmetic).
        SUB2C    => Classtype (two's complement
                    subtract).
        *1       => Line 1 identifier.
        ADD      => Line 1 operator (unsigned addition).
        $1       => First source.
        (        => Delimiter indicating the beginning
                    of a nested source.
        NOT      => Nested operation.
```

```
$2      => First (only) source to the NOT operation
           is the second source to the SUB2C.
)       => End nested source delimiter.
#1      => Constant (value 1).
<0>     => Bit map forcing the constant to a bit
           width of one.
{CI}    => Qualifier forcing the constant to be
           connected to the carry input of an adder.
```

2.7.2 Template Generation

SEL descriptions provide the guidance for the translation process. They are used to build a trial structure called a *template* that is an intermediate form between a SEL equivalence and a path graph. Templates are used during design synthesis to evaluate how well an equivalence matches the module set and the constraints. When an appropriate template is identified, the transformation is completed by translating the template into path graph elements and micro-operation instructions. Both the evaluation and the translation of the template into path graph form are intimately tied to the design transformation decision process that will be discussed in Chapter 3. The template generation process will be discussed here although it could best be though of as only half of a transformation.

A template is a tree structured representation of the SEL equivalence. It is always associated with a path graph node (refered to as the *root* node) that requires transformation. The actual nodes in the tree could be termed *pseudo path graph* nodes. Nodes that have representations in the path graph (registers, operators, and constants) are identical to path graph nodes. Special nodes are generated to represent sources, nests, repetitions, and line references. A template does not contain explicit links, but the data flow is implicit within the tree structure itself.

The bit widths of the sources, registers, and operators must be determined during generation of a template. If the SEL equivalence specifies absolute bit mapping for a node or a group of nodes, that information predominates. If no information is provided in the SEL description, the default bit width of the root node predominates. Between those extremes, a portion of the bit width information may be derived from both the SEL description and the root node if both absolute and relative bit mapping is specified in the range information.

The sources of the SEL description must be correctly associated with the actual source links to the root node. This is accomplished by *threaded traces* to the sources of the root. There is no problem or complication when the template refers directly to a root node in the path graph. However, the template generation process may be applied recursively to

synthesize and evaluate equivalences for nodes that are members of a template. In that case, the threaded traces become quite important to provide a path to the outer sources. The line identifiers may be used as sources in multiple line SEL descriptions and they must be threaded in the same manner as sources.

2.8 Merged Operation Transformations

Register nodes in a path graph may specify associated operations that the register must be able to perform. These associated operations are called **merged operations**. The merged operations are:

- Clear
- Load
- Left Shift
- Right Shift
- Increment
- Decrement

Clear and load are attributes of certain register modules and do not require transformation. They become part of the module selection process. The remaining merged operations may require transformation to insure a correct implementation of all desired functions.

Consider a register node that requires both shifting and counting. In the TTL module set, there are shift registers and there are counters, but there is not a single module that can implement both functions. This situation requires that one of the merged operations be selected to be included with the register and the other operation be implemented in some other manner. If a counter were chosen to implement the node, the shift functions could be implemented with multiplexors by skewing an output from the register to multiplexors on the input of the register. This technique could be used if the shifting was by a constant number of bits. Another choice would be to insert a shift register into the data path. This option is most desirable when shifting is by a variable number of bits. If a shift register were chosen to implement the node, the count function could be implemented by inserting an appropriate function (increment or decrement) in to the path graph and synthesizing it with available adders or subtracters. In module sets that do not contain counters or shift registers, both functions would have to be performed externally to the original registers.

2.9 Inversion

A path graph produced by the Data/Memory Allocator identifies inversions implicitly as attributes of the links interconnecting the nodes. In order to operate on the path graph it is necessary to make the inversions explicit by incarnating NOT nodes in the data paths where inversions are specified.

The input to a link and the output from a link have separate specifications for the logical sense of the data at the connection point. The logical sense of the data can be either *true* or *complemented*. If the input and output specifications to a link are the same (both true or both complemented), there is no net inversion. If the input and output specifications are different (one is true and the other complemented), an inversion is required. In this case, a NOT node is incarnated in the path graph. The link is moved to connect from the predecessor node to the NOT node. The input and output specifications are both set to *true*. A new link is incarnated and connected between the NOT node and the successor node.

2.10 Summary

The transformations described in this chapter are a set of tools that appear to be useful for performing logic synthesis by modifying the structure of a design graph. However, automating the synthesis process requires that the application of these transformations be based on judgment and careful direction. The next chapter discusses a control structure that can act as a surrogate designer by exercising such judgment and direction.

Chapter 3
Automating Logic Synthesis

"If everybody contemplates the infinite instead of fixing the drains, many of us will die of cholera" -- John Rich

3.1 Overview of the Automated LSMS Process

The previous chapter outlined a set of transformations used to alter the structure of a design. The transformations are key to successful synthesis, but they only provide a bare capability. The larger context of achieving a transformed design with the hardware selected for implementation requires a supporting process that directs the application of the transformations. The supporting process that evolved for this implementation operates in three distinct phases:

- Phase I - Removes any existing module selections and explicitly identifies inversions in the path graph.
- Phase II - Identifies all modules that might be used to implement each node. This phase also subdivides the design into smaller sections (called windows) to facilitate analysis and synthesis.
- Phase III - The surrogate designer that exercises the judgment necessary to direct the actual synthesis and module selection.

The first two phases perform the operations to prepare a design for synthesis. The third phase is far more involved and consists of a number of operations that evaluate module choices, perform trial synthesis, apply transformations, and install synthesized constructs in a path graph.

3.1.1 Phase I - Unbind/Invert

The first phase of LSMS consists of a pass over the entire design to dissociate the nodes from any existing hardware information[4].

A path graph delivered by the Data/Memory allocator specifies inversion implicitly on the interconnecting links. During this first pass over the design, any required inversions are identified and explicit NOT nodes are incarnated in the path graph using the **inversion** transformation. The inversions could have been handled in other ways at other points in the processing. It is conceivable that the inversions could be left as information stored in the links until specific devices were considered for implementing a node. At that time, consideration could be given to devices having inverting outputs or inverting inputs. Incarnating explicit NOT nodes in the path graph during Phase I was chosen primarily as a mechanism for defering consideration of special cases such as inverting outputs and DeMorgan transformations until the final structure of the design was better determined.

3.1.2 Phase II - Candidate/Window

A second pass over the entire design is required to identify candidate modules. Candidate selection develops a list of all devices from the current module set that might be used for implementation of a node. Candidate selection is performed initially for all nodes in a design that require modules. It is also performed later in the processing for any nodes that are incarnated during synthesis. Consideration was given to incorporating candidate selection into Phase I to save one pass over the design. Because of the incarnation of NOT nodes which altered the structure of the path graph during Phase I, it actually requires less overhead to make a distinct second pass for candidate selection.

During this pass, a design is partitioned into entities called **windows**. A window is defined to include exactly one register and all non-register nodes that are in any path terminating at the register. Windowing identifies localized areas of a design that may be structurally modified without having to be concerned with the more global issues of parallelism resulting in shared variables.

---

[4] In SYNNER, the unbinding can be disabled if desired

3.1.3 Phase III - Evaluate/Transform/Bind

Phase III processes a design window-by-window rather than node-by-node. Phases I and II were procedures for getting the design into a correct form for synthesis. Phase III actually directs the transformations and performs synthesis.

The processing in Phase III strikes a balance between the data path requirements, the designer constraints, and the operations and data path characteristics of devices in the module set. Evaluation of individual path graph nodes determines if any available devices can support the node operation. If devices are available, the designer constraints and module data path parameters are used to determine the best candidate for implementing the node. Finally, a comparison of the node data path parameters and the best candidate's data path parameters is used to determine if transformations are to be applied. If no best candidate can be found to implement a path graph node, synthesis through application of equivalence axioms is attempted. Each attempt at synthesis results in a template tree that consists of one or more path graph nodes. A template tree is evaluated against designer's constraints and the module set. Synthesis is attempted and the resulting tree is evaluated for all available equivalence axioms of the appropriate type. The tree with the best evaluation is selected to be installed as a replacement for the original path graph node. The installed tree must be linked into the path graph and micro-operation instructions must be synthesized to specify a control sequence for the replacement construct.

The remainder of this chapter is devoted to a more detailed explanation of evaluation, transformation decisions, trial synthesis, and template tree installation, linkage, and micro-operation synthesis.

3.2 The Module Database

Device information is the major driving factor in processing designs with SYNNER. Throughout the rest of this chapter, device information will enter into evaluations, decisions concerning the application of transformations, and finally, the actual Path Graph node implementation. The device information is of such importance that a brief overview of the module database (MDB) subsystem will be given here. A more detailed account is included in Appendix A.

The module database is uses a hierarchical access method that starts with a very general **index**. The index contains pointers to several **design style set** indexes. A design style set is an accumulation of module sets that favor implementation of a certain design style. A design

style set index contains pointers to one or more **databooks**. A databook contains the detailed information on devices that comprise a module set.

Databook entries are cataloged by a module identifier such as: SN7400. Each module entry is comprised of a number of lines of information. Each line may contain a number of fields. The exact format for module information is specified by a data definition (DDF). A data definition is separate from the database. It defines fields within lines and the data type for each field. The external DDF makes it relatively easy to change the information format whenever necessary. Supporting software uses the DDF to operate on the database. A database editor (DBEDIT) is used to enter and maintain information. An access package (MDBLIB) is available for use by user programs. MDBLIB provides the means to access information from any level of the database.

The flexibility provided at all levels of the database make it a powerful tool. In this research, cost, delay, and power were the parameters of interest. Each module entry in each database contains values for those parameters. If a different parameter became interesting in the future, it would be quite easy to add it to the databases. TTL and the Sandia CMOS Cells were the two module sets used in this research. More module sets can be added to the database using the existing tools (i.e. no programs will have to be changed). Hypothetical devices could be added to existing module sets. Designs could then be processed by SYNNER to determine how the hypothetical devices impact the design space. In this manner, proposed additions to module sets could be evaluated before they are actually built.

Characteristics of devices mold the module selection choices made by human designers. The remainder of this chapter is devoted to the description of a *surrogate designer* that is also guided by device characteristics.

3.3 Implementation of a Surrogate Designer

The various operations described in this section are all analogous to operations a human designer would perform while trying to choose devices from a module set to implement the nodes of a design. Generally, a human designer would have a larger capacity to exercise judgment in the application of these types of operations, and he would not be constrained by the limitations inherent in a software implementation.

For the moment, the approach described here can be viewed as a proposed implementation of the decision process leading to structural modification for module selection. The approach remained a gamble throughout the development of SYNNER since it could only be verified

(through the experimental comparison to human designers discussed in Chapter 4) after SYNNER was completed.

3.3.1 Evaluation

The evaluation process must determine which module (if any) is best suited to implement a node. In order to make such a recommendation, candidate modules are evaluated according to a set of weighted constraints and the module with the highest rank is identified.

3.3.1.1 Constraints

Constraints are supplied by the designer. Constraints are specified with the following BNF:

```
conkey conid conop conweight<CR>

conkey     ::= CON | CONSTRAIN
conid      ::= COST | DELAY | POWER |
               {any element stored in the module database}
conop      ::= conuop | conbop limit
conuop     ::= MAX | MIN
conbop     ::= LSS | LEQ | EQL | GTR | GEQ
limit      ::= {any integer, real, string or boolean
                according to the data type of <conid>}
conweight  ::= {optional: a number supplied by the designer
                identifying how much emphasis to place on
                this constraint.  Constraints are normalized
                before application}
```

An example of a typical set of constraints would be:

```
CONSTRAIN COST MIN
CONSTRAIN POWER GTR 10
CONSTRAIN DELAY MAX .5
```

In this example, COST is constrained to be the minimum cost module that will implement each node. POWER is constrained to be greater than 10. The value (10) will have units specified for a particular module set. The units are milliwatts for TTL while CMOS Cells use units of microwatts. DELAY is constrained to be the maximum in each case. The designer has chosen a weight of 0.5 for the delay. Since neither of the other constraints specified a weight, normalization would rank the importance of the constraints as:

```
COST  => 0.0
POWER => 0.0
DELAY => 1.0
```

If no weights were entered for any constrained item, equal defaults (other than 0) will be applied prior to normalization. If the constraints had been:

```
CONSTRAIN COST MIN
CONSTRAIN POWER MIN
CONSTRAIN DELAY MAX
```

The normalized weights would be:

```
COST    => 0.33
POWER   => 0.33
DELAY   => 0.33
```

If no constraints are entered, a design is processed to minimize the cost.

The constraint processing implemented in SYNNER is extensible. Any item that is specified in a databook can be constrained. While cost, delay, and power were the constrained parameters used throughout this research, the system design is flexible to support additional investigations using other parameters. If, for example, an application were required to base module selection on the temperature range of individual devices, it would be necessary to add a temperature (call it TEMP) field to the databook entries for each module. A designer could then constrain TEMP in the same manner as cost, delay, and power have been constrained here.

3.3.1.2 Constraint Evaluation

Each candidate module for a path graph node is evaluated against all constraints. The list of candidates is ordered such that the "best" candidate module for implementing the node is recommended. The evaluation procedure will be described in this section.

For the following discussion, the distinction between modules and packages must be stressed. Modules are logically indivisible gates, operators, registers, and multiplexors. Packages are physically indivisible collections of modules. In the TTL module set, a SN7408 *module* is a single two input AND gate. A SN7408 *package* contains four of the AND gates. In the Sandia Cell CMOS module set, modules and packages are identical since all cells are comprised of only one logical entity.

The first step in evaluating any candidate is to determine the number of modules that are required to span the bit width of the path graph node. The following symbols will be used in this development:

```
NB    = Path graph node bit width.
MB    = Module bit width.
N     = The number of modules required to implement the
```

path graph node.

Then:

$N = \lceil NB/MB \rceil$

The number of packages is primarily a function of the number of modules per package. The number of spare modules is included to reduce the required number of packages. Spare modules can become available if only a portion of a package has been committed to implement a node. The number of packages required can be determined by:

P = Number of packages.
S = Number of spare modules.
C = Number of modules per candidate package.

Then:

$P = \lceil (N - S)/C \rceil$

A few points should be made about use of spare modules. While this practice tends to reduce the overall cost of a design, it also makes module selection somewhat dependent on the order in which a design is processed. Spares might be available if the design were processed in one order, but they might not be available if the design were processed in a different sequence. This is not believed to be a significant problem. Prior to using spares during evaluation, designs rarely had more than five percent spare modules after complete processing. Therefore, the sequencing problem could be expected to occur at most five percent of the time. It is interesting to note that the number of spare modules is still generally in the range of five percent even when spares are included during evaluation.

All constraints except **COST** are evaluated directly on the basis the number of packages and of information stored in the database. **COST** requires a bit of manipulation before evaluation. Blakeslee [Blakeslee 75] indicates that there is an overhead cost associated with each member of a packaged (i.e. TTL) module set. The overhead cost can be used to lump together estimated costs of manufacturing a complete system. For the packaged module sets, three dollars ($3.00) is added to the cost of the package to arrive at an implementation cost for mounting a package. For cell module sets, it is assumed that package cost is analogous to cell area since the real estate on a wafer of silicon is a limited commodity. The overhead cost is assumed to be a penalty incurred for interconnecting the cells. In the case of Sandia Cells, a value of three (3) times the cell area [Sandia 78] is used as the overhead-cost model. The cell overhead value does not include an estimate of I/O pad area. To formalize the overhead-cost calculation, define:

PC = package cost

OHC_p = overhead cost for package module sets
OHC_c = overhead cost for cell module sets

OHC_p = P * ($3. + PC)

OHC_c = P * (3. * PC)

The resulting overhead-cost is then evaluated according to the specified criteria.

After the number of packages and the overhead cost values are determined, a candidate is evaluated against all the constraints. There are seven constraint operators:

```
MIN => Minimum value in candidate list.
LSS => Less than some designer specified value.
LEQ => Less than or equal to some designer specified value.
EQL => Equal to some designer specified value.
GEQ => Greater than or equal to some designer specified value.
GTR => Greater than some designer specified value.
MAX => Maximum value in the candidate list.
```

Comparison against database values (or the computed overhead cost value when cost is constrained) results in a boolean (1 or 0) value indicating whether or not the condition was met. The comparison is performed twice on the candidate list in order to determine maximum or minimum values. In those cases, the first pass through the list serves to find the maximum or minimum database value. The second pass actually determines the boolean for each candidate.

At this point, each candidate has an associated list with one entry corresponding to each constraint. Each entry in the constraint correspondence list contains the boolean indicating if the candidate matches the constraint. The boolean corresponding to each constraint is multiplied by the designer specified (normalized) weight for each constraint. A candidate evaluation is the sum of the weighted booleans. Since the weights are normalized and the individual constraint evaluations are booleans, the candidate evaluation can take on any value in the range of 0.00 to 1.00.

The entire process can be summarized as follows: If that there are N criteria and J alternate candidates. An evaluation of criteria i is represented by e_i. If a normalized weight, w_i, is placed on each criteria i, then the evaluation of alternative j in the list of alternatives is:

$$E_j = \sum_{i=1}^{N} w_i e_i$$

The candidate in the list with the maximum E_j is recommended as the "best" module.

3.3.2 Synthesis

Synthesis is invoked whenever there is no "best" candidate module for a node. A synthesized node is replaced with one or more nodes (synthesis by expansion). If synthesis were applied to all nodes (even those with a "best" candidate), the transformation design space would have no bound in the direction of "worse" designs. That appears to be reasonable for an actual design space but it is not practical to allow an automated system to search for infinities. Because of limited run time and finite memory, the implementation of LSMS bounds the design space by only applying synthesis when necessary. This method has no impact on the optimum end of a design space. It only limits searches in the direction of "worse" designs. Fortunately, the optimum end of a design space is the most interesting region when implementation of designs is the goal.

Synthesis uses the equivalence transformations (refer to Section 2.7.1) to build template trees. The tree generation process was discussed in Section 2.7.2. In that section it was indicated that building the tree amounted to only half of a transformation since the control, evaluation, and installation of synthesized trees (which will be discussed here) is integral to the process.

3.3.2.1 Synthesis Control

Synthesis is initiated when there is no device in the module database that can be used to implement a Path Graph node. The list of axioms is searched until an equivalence for the Path Graph node function is found. If no equivalence is found, the node is considered to be unimplementable. It would then be up to the designer to insert an appropriate equivalence in the axiom database. In the more usual case, there would be several equivalences. Template trees will be synthesized and evaluated for *all* equivalences that match the Path Graph node function.

A synthesized template tree can be thought of as an independent Path Graph. Each node is subjected to the same candidate module search and node evaluation that occurs on the design's Path Graph. During tree node evaluation it is quite possible that no modules will be found to implement a node. In that case, synthesis is applied recursively to the node in the template tree. This mechanism allows template trees to be built up from successive equivalences. Consider a design being implemented with TTL. If a two's complement greater-than-or-equal (GEQ2C) operator were encountered, there is no TTL device that can directly implement the function. Synthesis would build a tree representing the equivalence:

```
RELAT:GEQ2C      *1 NOT (LSS2C $1 $2)
```

A search of the TTL database would find that the NOT operation could be implemented, but there is still no device that could directly implement LSS2C. Synthesis would be applied recursively and would build a subtree representing the equivalence:

```
RELAT:LSS2C      *1<0> SUB $1 $2
```

A **SUB** (unsigned subtract) operator is available (the SN74181 ALU) in the TTL database. Therefore, this tree would be completely implementable with TTL devices and no further recursive synthesis would be attempted. The recursively synthesized tree would be identical to a tree developed from an equivalence written as:

```
RELAT:GEQ2C      *1 NOT (SUB $1 $2)<0>
```

Recursive synthesis is limited to a depth of 25 to prevent stack overflow in the PDP-10 implementation. Generally this limit would be reached only if recursive tree synthesis was applied in very large loops (a fruitless search at best, and an error at worst). Recursive synthesis reaching a depth of 10 or less was sufficient for all designs encountered in this research. Another limit is placed on recursive synthesis to identify and eliminate tight loops. If either of the last two equivalences are identical to the current equivalence, subtree synthesis is suppressed.

At each tree or subtree, all possible equivalences are developed and evaluated. If there had been another equivalence for **LSS2C**, a second subtree would have been developed and evaluated in the example above. If there had been an additional equivalence for **GEQ2C**, a tree would have been synthesized and evaluated. After all equivalences are exhausted (at each level of recursive synthesis) to tree or subtree with the highest evaluation is selected to replace the Path Graph node. Details of tree evaluation will be discussed in the next section.

3.3.2.2 Template Tree Evaluation

Tree evaluation develops a summary indicator for the entire tree (or subtree if synthesis was recursively entered). Evaluation begins by performing candidate searches and evaluations on each node of the tree. Candidate evaluation for tree nodes is identical to candidate evaluation for Path Graph nodes (refer to Section 3.3.1.2). Suppose that there are **N** nodes in a template subtree that require modules for implementation. An evaluation of node **k** produces a "best" module candidate with an evaluation factor of E_k. The "best" candidate for a node is the one that has the maximum E_k. The summary evaluation t_n for the n^{th} template subtree is given by:

$$t_n = (1/N)(\prod_{k=1}^{N} E_k)$$

If there are **m** subtrees in an template tree,

the evaluation for the i^{th} template tree is given by:

$$T_i = \prod_{n=1}^{m} t_n$$

If there are i template trees, the final evaluation (T) is given by:

$$T = \text{MAXIMUM } (T_i)$$

A product model was chosen for tree evaluation because it is necessary to distinguish between single node and multiple node trees. If individual node evaluations were simply summed, the result could be a number larger than 1.0. Template selection is based on the maximum evaluation, so large values would be favored. If a sum were divided by the number of nodes in the template tree, the resulting value would be normalized to the range 0.0 to 1.0. Then it would not be possible to distinguish a single node tree from a multiple node tree although the latter could be more costly to implement. If the product of individual node evaluations is taken, the maximum value will be 1.0 (the node evaluations are all in the range of 0.0 to 1.0). If the product is then divided by the number of nodes (N), a multiple node tree will produce a lower value than a single node tree.

When an appropriate template tree is identified, installation of the replacement construct occurs in three stages: node installation, node linkage, and micro-operation synthesis.

3.3.2.3 Node Installation

The template tree is walked from the root node. If a tree node is a register, operator, multiplexor, or constant, the tree node is copied into the path graph data structure. All other types of nodes in the template tree do not directly generate corresponding path graph nodes. *Source* and *reference* tree nodes are used as information sources to generate links in the path graph.

3.3.2.4 Node Linkage

The linkage process interconnects the template tree nodes that were installed in the path graph. The actual processing methodology becomes rather involuted and will not be discussed here. However, a few specific problems that are addressed during linkage will be described. Template trees allow two types of inputs to register, multiplexor, or operator nodes:

- *sources* refer to connections at the input boundary of the template tree

- *references* are connections between nodes in the template tree.

In the simplest case sources must be matched to the correct input link to the root node. The destination of that link must be changed to point to the the newly installed path graph node that corresponds to the template tree node to which the source is attached. References require identification of both the source and destination nodes in the path graph. A link must then be installed between those nodes. Complications arise when template trees are nested. Sources of a nested template tree no longer refer to actual path graph links. Rather, they refer to sources in the template to which they are nested. The linkage process must follow the *threaded trace* described in Section 2.7.2. This procedure insures that the linkage parameters are derived from the true source and the nested destination node. After all source and internal connections have been made to the replacement path graph construct, all output links from the root node are moved to the last node of the construct.

3.3.2.5 Micro-Operation Synthesis

After installation and linkage, the template tree is accessed one final time to synthesize a micro-operation sequence for the structure. A root node is associated with zero or more micro-operations and the sequence must be modified at each of those points. A root node may be referenced by the micro-operation sequence as an operator, a source, or a destination. Operator micro-operations are replaced with a sequence that reflects the control flow of the template tree. The micro-operation sources to an operator become sources to the first replacement micro-operation. Intermediate micro-operations in the replacement sequence are chained unless a register is included in the template tree. The last replacement micro-operation will receive the destination information from the original micro-operation. If the root node acted as a source to micro-operations, the last replacement micro-operation is referenced as the new source. Only registers may be destinations in the micro-operation sequence. Registers may be installed in the path graph from a template tree. They could then serve as destinations for existing or synthesized micro-operations. Multiplexors in a template tree do not directly enter the micro-operation sequence, but they do cause a **select** micro-operation structure to be built.

After installation, linkage, and micro-operation synthesis are completed, the root node and the template are deleted.

3.3.3 Automatic Transformation

The *autotransform* process is the kernal controlling four critical decisions that result in logic synthesis and module selection:

- Synthesis
- Transformation
- Merged operator synthesis
- Binding (the association of selected module information to a node)

A design is processed window-by-window. Within a window, each node is subjected to a series of tests that direct it through the appropriate transformations. These tests are described in the following paragraphs.

3.3.3.1 Equivalence Synthesis

Synthesis is attempted only if no "best" candidate is identified for a node and proceeds as outlined in Section 3.3.2.

3.3.3.2 Partitioning and Combining Transformations

Each node is tested to determine if it might benefit from either partitioning or combining transformations. The test for partitioning uses the bit width of operators and registers or the number of inputs for multiplexors as the deciding factor. The bit width or number of inputs will be refered to as the *size* in the following discussion. The size of the "best" candidate module is compared to the size of the node. If the node size is an even multiple of the module size or if the module size is greater than the node size, partitioning is not required. If the module size is less than the node size and is not an even multiple of the node size, the partitioning transformation (Section 2.5) is activated to split the node.

The test for for applying the vertical join transformation requires that the nodes be operators of the same type that are in a cascaded configuration (the output of one node is an input to the other node). The top part may have only the output connecting it to the bottom part. Finally, there must be a candidate module that has enough inputs to support the combined node. If the top part has **n** inputs and the bottom part has **m** inputs, a candidate module must have at least (**n** + **m** - 1) inputs. If all of the conditions are met, the vertical join transformation (Section 2.6.1) will be activated.

All possible pairing of operators in the design is tested to determine if they may be merged.

In order to qualify, a pair of operators must be in independent control structures (i.e. the must never be activated simultaneously). There must also be a device in the module set that will perform the functions of both nodes. Finally, the nodes must share a data path. This restriction insures that the nodes are of similar (although not necessarily identical) size. If the conditions are satisfied, the horizontal join transformation (Section 2.6.2) will be activated.

3.3.3.3 Merged Operations

Merged operations constitute special cases that require individual responses. The lack of generality in merged operation transformations and a persistent lack of time resulted in incomplete implementation. As indicated in Section 2.8, these operations only impact registers that are flagged for additional functions. That section outlined the possible responses to the merged operations. All cases have been identified, but only the increment and decrement have been implemented. These operations are implemented by incarnating a functional level increment (decrement) node in the path graph. The incarnated node is routed between the output of the register and a multiplexor collecting the inputs to the register. The register is rewindowed and cycled through the autotransform process where the increment (decrement) is synthesized into a form that is consistent with the modules in the database.

3.3.3.4 Binding

If there is a "best" candidate module for a node, that module's information is associated with the node after all synthesis and transformations are completed. The association of information is referred to as *binding* the module to the node.

3.4 Conclusions

Automatic processing continues under control of the surrogate designer until the entire design has been completed. The automatic system is fast (Appendix D shows that a PDP-8 can be processed for TTL in under three minutes wall-clock[5] time or in less than nine CPU seconds) and complete (100% of the nodes have modules selected), but it is not yet clear how good a job it does. A design experiment was undertaken using human designers to process descriptions that could also be processed by SYNNER. The purpose of the experiment was to determine if a reasonable set of transformations had been identified and to provide a sample of designs to compare against the automatic system. The design experiment is described in the next chapter.

---

[5]On a DEC KL-10 CPU with approximately 20 other active jobs.

Chapter 4
Validation of Transformations

"MURPHY WAS AN OPTIMIST" -- T Shirt philosophy circa 1980

4.1 Introduction

The LSMS level of design has been explored in a manner that has stressed generality. An effort has been made to avoid automating a "bag of designer tricks". The result has been the definition of five basic structural transformations and a structural synthesis capability. The resulting system is able to completely process most designs that have been encountered, but there still are many areas of the process that are incomplete. The current implementation operates on very localized parts of a design and it has some shortcomings in its synthesis capabilities. Since it can completely process most designs, two questions must be answered before much confidence can be placed in its results:

- How closely does the LSMS system approximate the capabilities of good designers?
- Are there any additional transformations that would allow the system to produce better designs?

An experiment was conceived to answer those questions. Two descriptions of digital devices were chosen to be processed by a group of volunteer designers. Each designer was asked to process the two descriptions using modules from the TTL module set for one description and modules from the Sandia Standard Cell module set for the other description. Constraints were placed on the final cost and delay of each design. Designers were requested to account for every transformation that they used to arrive at their final implementation.

The results of the experiment provided a series of points near the optimal end of a cost/delay design space. Each designer described the transformations he used to arrive at those points.

The experimental data holds the answers to the calibration and transformation questions. The LSMS system was used to process all of the description/module set pairs that the designers produced. By comparing the automated designs to the experimenter's designs, it was possible to determine how closely the LSMS system approached manual methods. Using the audit trails the designers provided, it was possible to identify transformations that would improve the automated performance.

The calibration will show that the automated system produces designs that compare favorably with those done by humans. This result tends to support the choice of transformations proposed in Chapter 2.

4.2 Design of the Experiment

Before discussing the details of this experiment and its results, a brief background will be given which contrasts an earlier experimental approach to the current work.

4.2.1 Background

Thomas [Thomas 81] used extensive human experiments in order to validate the proposed Register Transfer Computer Aided Design (RTCAD) model for design automation. He conceived and ran two experiments to gather information about separate aspects of the design process:

1. The physical allocator level experiment was used to measure designer performance when implementing a specific functional description with a specific design style and a specific module set.

2. The module independent level experiment was used to measure designer performance in translating a behavioral hardware description to a functional logic level.

The current design experiment has similarities to the first of those experiments. However, there are some significant differences that make the current work unique. He had proposed a model of the design process and used the experimental *results* to justify that model without being concerned with the methods used to attain the results. The RTCAD model included global optimization, design style selection, and physical allocation. In the current experiment,

a model of the LSMS step of CMU-DA[6] has been proposed (in the form of a series of transformations and a methodology for applying them). The *results* of this design experiment were used to justify the LSMS model, but because the model identifies a method to attain similar results, the *methods* used by the designers were carefully monitored.

4.3 Description of the Experiment

The experiment was conceived to use six designers, two descriptions and two module sets. The descriptions are of a change mechanism for vending machine application and a truncated version of the PDP-8 minicomputer. The module sets are the common 7400 series TTL devices and the Sandia Laboratories Standard CMOS Cells for LSI implementation.

The six designers (referred to as designers "1" through "6") were CMU Electrical Engineering students. Two (designers 4 and 5) were PhD candidates, two (designers 1 an 6) were masters candidates and two (designers 2 and 3) were graduating seniors. All of the designers were familiar with digital circuit design principles. All had experience designing with TTL and two (4 and 5) had direct experience designing with the Sandia Standard Cells. One of the designers (1) was working on the control allocation problem for CMU-DA.

The descriptions were selected to provide different implementation problems. The first description is of a change mechanism which performs a large proportion of arithmetic and comparisons. The ISP and the functional logic diagram (drawn from the D/M allocator path graph) appear in Appendix C. The second description is of a minimal version of the PDP-8 minicomputer. All I/O and Operation group instructions were removed from the description leaving the fetch/execute cycles, addressing modes, interrupt service and full descriptions of the remaining six basic instructions. The ISP and the functional logic diagram for the truncated PDP-8 appear in Appendix C.

Each designer was asked to transform and select modules for the two descriptions. The experiment was conceived to be balanced by assigning three designers to each of two groups (the "A" group and the "B" group). The "A" group was asked to select modules for the change mechanism from the TTL module set and to select modules for the PDP-8 from the Sandia Standard Cell module set. The "B" group was to select modules for the change mechanism from the Sandia Standard Cells and select modules for the PDP-8 from the TTL module set. This assignment of designers, descriptions, and module sets is depicted in Figure

[6]The current name of the design system that is being used to refine and implement RTCAD concepts.

4-1. The purpose of this "cross assignment" was to eliminate learning curves that might cloud the results. Since each designer was to process two descriptions, he was faced with a totally new situation (module set and description) each time.

	TTL	CELLS
Change Mechanism	A (1, 2, 3)	B (4, 5, 6)
Small PDP8	B (4, 5, 6)	A (1, 2, 3)

Figure 4-1: Design Experiment Assignment

The experiment as conceived suffered somewhat in the actual execution. Only five off the six designers actually completed the experiment leaving ten instead of twelve processed designs. Another designer transposed the design/cell pair groups to which he had been assigned. He had been assigned in the "B" group but actually processed the designs as if he had been assigned to the "A" group. The cumulative result of these problems in the execution of the experiment left the unbalanced assignment shown in Figure 4-2.

	TTL	CELLS
Change Mechanism	A (1, 2, 3, 4)	B (5)
Small PDP8	B (5)	A (1, 2, 3, 4)

Figure 4-2: Actual Experimental Assignments

While the actual assignments are not as balanced as was desired, the results are just as useful in most ways as they would have been from the experiment as it was designed. One factor desired from the experiment was to determine if there were transformations not included in SYNNER that would be useful for structural modification and module selection. With 5/6 of the planned data available, there is an adequate sample for that purpose. The

other purpose of the experiment was to determine how closely SYNNER approximated designs processed by humans. The inadvertent increase in the sample size of the "A" group actually increases the confidence in the results for the change/TTL and PDP-8/Sandia designs. The corresponding reduction of the "B" group to a single data point for each design leaves no opportunity to draw statistical conclusions for those design/module set pairs.

Preliminary results [Parker 79] of the automated CMU-DA system have shown that the D/M allocator produced designs that were approximately 30% worse than good hand designs. Since the objective of the experiment was to evaluate aspects of the LSMS process, it was important to neutralize any effect of the allocator's additional 30% on the measured data. This was accomplished in two ways:

- Designer's received functional logic level descriptions identical to those that could be processed by SYNNER. The functional logic level descriptions contained the 'inappropriate design style' errors, but the errors were identical for both the human and automatic design processing.

- Designers were restricted in the degrees of freedom they had in transforming the designs to the class of transformations appropriate to the LSMS level. The restrictions were general and described the grain of the transformations that were restricted rather than suggesting or disallowing particular transformations. The statements of restricted classes of transformations are reproduced here from the instructions provided to the designers:

    - Permissible types of transformations are:

        - Any operations that pertain to a local area of the path graph and do not change the behavior of the design.

    Operations that are not permissible include:

        - Busing of data paths: Individual specifications of datapaths is a result of using the allocator for the distributed design style. Certain individual data paths could be combined in a bus. However, such an optimization would cause a change in design style.

        - Large parallel/serial transformations: Relocation of large groups of nodes to place them in line or in parallel with other operations is not permissible. Such relocation is a decision that must be made by an allocator.

4.4 Instructions to Designers

The designers were instructed to meet limits on the cost and delay of each design. In order to accomplish this, measures of the data part cost and control part delay were described. In addition, a thorough accounting of all transformations was requested. In order to clarify the exact set of requirements placed on the designers, portions of the instructions will be reproduced and discussed in the following paragraphs.

4.4.1 Data Part Cost

The data part cost of implementing a design is derived from the sums of the **direct cost** for each module and the **overhead cost** incurred for each module or package. In the case of packaged devices such as the members of the TTL module set, actual dollar cost is used as the direct cost. The overhead cost for TTL is the $3.00 per package identified by [Blakeslee 75]. For the Sandia Cells, dollar cost has no particular meaning. Instead, the module area is used as the direct cost. The overhead cost defined to be a penalty (in area) required for interconnect routing between cells. At the time the experiment was conducted the best value available was a factor of 1.2 times the cell area for routing overhead. The cell overhead factor was changed to 2.0 after the experiment was completed. The data will be reported first with the factor of 1.2 as it was gathered from the experiment, then it will be corrected to use the more accurate 2.0 factor. The computations of data part cost as given to the designers were:

TTL:

$$TTL.TOTAL.COST = (total.packages) * overhead + \sum package.cost$$

Where:
- total.packages = the total number of packages used.
- package.cost = the cost of each package used.
- overhead = $3.00

CELL:

$$CELL.TOTAL.COST = \sum cell.area * (1 + overhead)$$

Where:
- cell.area = silicon area (in square mils) for each cell.
- overhead = 1.2

4.4.2 Control Part Delay

The control part of D/M allocator produced designs is represented in the Micro-Operation Sequence Table which describes the activation sequence of the data part (Path Graph) nodes. Transformations on the structure of the data part may cause changes in the micro-operation sequence. The addition of controllable nodes in the data part would require that corresponding micro-operations be added to the control part. Deletion of controllable data part nodes would require the deletion of the corresponding micro-operations. Merging of controllable data part nodes into multifunction devices probably would not alter the micro-operation sequence.

The control part measurements requested for each description were the number of control words, the number of control bits, and the total delay for one execution of the longest path through the design. The number of control words is closely related to the number of micro-operations. However, some of the micro-operations do not contribute to the word count while others contribute more than one word. A table of countable micro-operations was provided to the designers to help compute the final control word count.

The design delay computation requires that the longest path through the micro-operation sequence be identified. The delay for each data part module associated with each micro-operation in the longest path must be added to the total. Multiplexors do not explicitly appear in the micro-operation sequence, but their delay must also be added to the total.

For the purpose of the experiment, the bit width of the control words was based on the assumption that each potentially controllable path graph node would require one bit in the word. The potentially controllable path graph nodes are:

```
Registers
Memories
Multiplexors
Demultiplexors
Operators
Temporary Registers
```

It was also assumed that the next-state ROM address would contribute LOG_2(control.words) to the bit width. This simplistic estimate was scrapped in the implementation of SYNNER (after the experiment was completed). The current approach estimates the bit width by counting the actual maximum control line requirement for selected modules. The initial analysis of the raw data from the experiment will be done using the method requested of the designers. Analysis of the data that is used in comparisons to SYNNER's performance will be done by recomputing the bit width using the actual control requirements of the selected modules.

The model of a controller described to the designers was a simple ROM based state machine that consisted of a STATE.REGISTER and a ROM. The next-state address was described to be part of the total bit width which was identical for both the STATE.REGISTER and the ROM. This machine is different from any of the controller models used by the Control Allocator, but it is sufficient to generate the upper bound estimates of the number of words and the required bit width. The upper bound estimates of controller words and bit widths are appropriate for the LSMS level since LSMS is primarily concerned with detailed data part transformations. The control part modifications dictated by data part transformations are made only to keep the control information relevant to the data path structure. While the data part is translated from one design level to another, the control information is updated at the same level of design that it entered LSMS.

4.4.3 Design Constraints

Constraints for cost and delay were placed on each description/module set pair. Designers were instructed to meet or exceed the constraints if possible, or make the best compromise between cost and delay if the constraints could not be met. The constraints were based on estimates derived by a trial run of the experiment (hand binding the designs) and using available facilities in SYNNER. At the time the constraints were derived, SYNNER was not capable of any automatic binding. It did have some operational transformations that could be applied directly by a designer.

The constraints placed on the description/module set pairs were:

```
Change.Mechanism/TTL

        Package + overhead cost ($3/package) less than $125.00
        Delay (longest Path of the design) less than
            5.5 microseconds

Change.Mechanism/Sandia

        Package + overhead (1.2 * cell area) less than 110,000
            Square Mils
        Delay (longest path of the design) less than 5 microseconds

PDP8/TTL

        Package + overhead cost ($3/package) less than $130.00
        Delay (longest path of the design) less than 4 microseconds

PDP8/Sandia
```

```
Package + overhead (1.2 * cell area) less than 105,000
    Square Mils
Delay (longest path of the design) less than 4 microseconds
```

These constraints were believed to be marginally attainable. They were selected to require significant transformations in order that they could be met.

4.4.4 Special Considerations

The following special considerations were identified to the designers:

- When using a TTL module set use only devices with totem pole drive. Do not use open collector or tristate output devices. Do not use Schottky devices.
- Do not bind the array memory (MP) in the PDP8 design.
- In the Sandia cell set, the width of devices is given in the data sheets. We want the resulting area used. The height of all standard cells is 14.4 mils.
- In Sandia Cell designs, use only cell input and output capacitance to determine if buffers are required. Do not attempt to account for wire or tunnel capacitance.

4.5 Analysis of Experiments

The data from the experiments will be presented three ways. Initially, the data as gathered from the designers will be presented and discussed. Certain problems and oversights occurred in the designer's execution of the experiment. Those problems will be discussed and the data will be extended to provide a consistent basis for analysis. Some of the accounting methods used by SYNNER are slightly different than the accounting methods requested of the designers. The final presentation will place the experimental data on the same accounting basis SYNNER uses for the purpose of comparing the hand and automated results.

4.5.1 Data Presentations

The data is presented in tabular form. The parameters that appear in the tables are defined below:

- **Total Bindable Nodes** - The final number Path Graph nodes that require modules for implementation. If Path Graph nodes occur as the result of partitioning a register or operator, only the original node (the *left-part*) will be added to the bindable node count.

- **Nodes Actually Bound** - The number of bindable nodes that have modules assigned.

- **Percent Binding** - Percentage of nodes actually bound.

- **Total Packages Used** - The number of packages used for the design. In the case of the TTL module set, this number is less than or equal to the number of modules actually used. In the case of Sandia Cells, the package and module count is identical.

- **Total Modules Used** - The number of individual modules used to implement the design. Modules are the discrete logical entities used to implement nodes of a Path Graph. In packaged module sets, individual modules may be replicated within individual packages.

- **Total Spare Modules** - The number of unused modules resulting from assigning packages to implement a design.

- **Module Utilization** - The percentage of total modules used to the total modules available.

- **Total Power** - The sum of the power required by all packages. This value is in milliwatts for TTL implementations and microwatts for cell implementations.

- **Control Words** - The total number of micro-controller words required by the design. The count of control words is derived directly from the Micro-Operation Sequence Table. The resulting number should be viewed as an upper limit since no attempt is made to model or estimate the performance of the Control Allocator.

- **Control Word Size** - The number of bits in each control word. This field presents a problem in the following data presentations. At the time the experiment was conducted, the designers were instructed to count one bit for each bindable node. The rational was that each operation has an entry in the Micro-Operation Sequence Table and it would be the task of the Control Allocator to convert that information to a number of control lines (zero or more) for each module. Since the time of the experiment, the module database has been upgraded to include the actual number of control lines for each module and SYNNER has been modified to make use of that information when accumulating this statistic. The raw data and consistent data presented in this field will be on the basis of the instructions to the designers. The data converted for comparison to SYNNER performance will contain the count derived from the actual module information. The reported number of bits should be viewed as an upper bound since it represents fully horizontal micro-code. All packing of control words is deferred to the Control Allocator.

- **Controller Address** - The number of bits required to address the control words: Log_2(**Control Words**) (rounded up).

- **Max Control Path** - A count of the number of control words in the longest path through the Micro-Operation Sequence. Loops are only transited once when measuring the longest path.

- **Max Path Delay** - The sum of the module delays in the maximum control path.

This number includes delays of multiplexors which are not explicitly mentioned in the Micro-Operation Sequence.

- **Raw Package Cost** - The sum of the individual package costs. This figure is in dollars for packaged module sets. It is in square mils for cell module sets.

- **Total Package Cost** - The raw package cost plus the overhead cost. Overhead for TTL is computed at $3.00 per package. Overhead for cells was specified as an additional 1.2 times the raw package area for the experiment. Later information (currently used by SYNNER) uses a cell overhead factor of an additional two times the cell area.

4.5.2 Initial Parameters

The descriptions have initial values for parameters not directly dependent on module information. There is an initial number of "Bindable Nodes" (nodes that require modules for implementation), "Control Words", "Control Bits" (identical to the bindable nodes by the accounting method requested), and "Longest Path". These initial parameters for each design are given below:

Small PDP-8:

```
Bindable Nodes: 17
Control Words:  89
Control Bits:   17
Longest Path:   38
```

Change Mechanism:

```
Bindable Nodes: 32
Control Words:  72
Control Bits:   32
Longest Path:   69
```

4.6 Change Mechanism Using TTL Modules

Designers 1, 2, 3, and 4 worked on the change mechanism using the TTL module set. The next subsection will be devoted to presentation and analysis of the experimental data. The more subjective process of determining and discussing the transformations used by the designers will complete this section.

4.6.1 Data Analysis

The designers were presented with drawings of the change mechanism as the D/M allocator produced it. They also had the path graph form of the description. The drawings did not explicitly denote inversions because inversions are attributes of the connecting links. The most common oversight throughout the experiment was missing an inversion. The reason for most of the discrepancies between **Bindable Nodes** and **Nodes Actually Bound** was a missed inversion. An inversion was considered to be properly handled if:

- A node which directly dealt with the inversion could be explicitly identified.
- An inversion originated from a node that was bound with an inverting output. If non-inverting links also originated from the node, it must also have a non-inverting output.

DESIGNER	1	2	3	4
Total Bindable Nodes:	21	31	30	29
Nodes Actually Bound:	20	25	23	25
Percent Binding:	95%	81%	77%	86%
Total Packages Used:	20	25	23	23
Total Modules Used:	31	37	39	45
Total Spare Modules:	7	4	5	8
Module Utilization:	82%	90%	89%	85%
Control Words:	67	67	68	69
Control Word Size:	20	25	23	25
Controller Address:	7	7	7	7
Max Control Path:	59	60	64	61
Max Path Delay (NS):	1107.5	1279.0	972.0	1061.0
Raw Package Cost:	$ 13.12	22.36	15.43	22.87
Total Package Cost:	$ 73.12	97.36	84.43	91.82

Table 4-1: Change/TTL Raw Data

In one case (Designer 1), an attempt was made to deal with the inversion but failed because of a misapplication of DeMorgan's theorem. Designer 1 dealt with the other three inversions using the criteria of selected modules which had inverting outputs. Designer 2 missed a link inversion and missed two of the nodes in the description. A questionable application of one package to three nodes makes it possible to speculate that the description would not perform as desired. The nodes were one bit carriers (registers) in the description. The attempted optimization used a four bit register (with common clock, clear, and preset). Unfortunately, the description gives no hint about how the individual registers are to be loaded, so this "optimization" was allowed but will not be one identified as useful for general module selection. Designer 3 missed three of the registers and all four of the inverting links. Designer 4 missed all inverting links.

Corrections were applied to each experiment such that the designs had all nodes bound with modules. The method used was to select the lowest cost option without changing any of the existing choices (i.e. not applying any transformations to the designs). Designer 1's correction required a one bit inversion. A spare inverter was available, so the cost was not changed. The module utilization was improved and the maximum path count and delay were corrected. To correct for the unbound nodes in Designer 2's description, an extra package was required. Several packages (inverters and registers) were required to correct the unbound nodes left by Designers 3 and 4. The resulting corrected data appears below:

DESIGNER	1	2	3	4
Total Bindable Nodes:	21	31	30	29
Nodes Actually Bound:	21	31	30	29
Percent Binding:	100%	100%	100%	100%
Total Packages Used:	20	27	25	24
Total Modules Used:	32	43	46	49
Total Spare Modules:	6	10	10	10
Module Utilization:	85%	81%	82%	83%
Control Words:	68	71	71	72
Control Word Size:	21	31	30	29
Controller Address:	7	7	7	7
Max Control Path:	60	64	67	64
Max Path Delay (NS):	1117.5	1333.0	1002.0	1091.0
Raw Package Cost:	$ 13.12	22.79	17.56	23.05
Total Package Cost:	$ 73.12	103.79	92.56	95.05

Table 4-2: Change/TTL - Consistent Data

In order to compare the designer's results to SYNNER's results, two changes were made to the data. First, an accounting of the power required by the selected modules (in milliwatts) was added to the table. Power was not one of the constraints requested of the designers, so its inclusion provides a random point for comparison. The second change was in the method of accounting for required control lines. The designers were instructed to count one bit required for each bindable node. The more correct approach is to count the actual control lines required. Although it could be argued that designers might have optimized differently had they known that actual control lines would be counted, it appears from the data that only SYNNER changed in relative proportion by this accounting method.

The data reported for SYNNER's work on the change mechanism was derived by requesting that the **cost** and **delay** be minimized. These parameters are the same ones the designers were asked to minimize.

DESIGNER	1	2	3	4	SYNNER
Total Bindable Nodes:	21	31	30	29	23
Nodes Actually Bound:	21	31	30	29	23
Percent Binding:	100%	100%	100%	100%	100%
Total Packages Used:	20	27	25	24	28
Total Modules Used:	32	43	46	49	50
Total Spare Modules:	6	10	10	10	5
Module Utilization:	85%	81%	82%	83%	90%
Total Power (MW):	3243	6649	5058	6958	8525
Control Words:	68	71	71	72	71
Control Word Size:	44	54	37	52	63
Controller Address:	7	7	7	7	7
Max Control Path:	60	64	67	64	68
Max Path Delay (NS):	1117.5	1333.0	1002.0	1091.0	1024.0
Raw Package Cost:	$ 13.12	22.79	17.56	23.05	28.07
Total Package Cost:	$ 73.12	103.79	92.56	95.05	112.07

Table 4-3: Change/TTL - SYNNER Accounting Basis

The mean and standard deviation were computed for each of the measured categories. Computations were based on the population of four designers. The data was from the final table (corrected and placed on the same accounting basis that SYNNER uses). The results of these computations are shown in Table 4-3. The table gives the mean ($\overline{x}$), the mean plus one standard deviation ($\overline{x} + s_x$), and the mean plus two standard deviations ($\overline{x} + 2*s_x$) for the designer values. SYNNER's performance in each measurable parameter is given in the last column of the table. It should be noted that SYNNER's performance in all measured categories except **Control Word Size** is within two standard deviations of the mean. This is the range within which 95% of the population of designers would fall if the population was normally distributed. One standard deviation from the mean is the range within which 68% of the population of designers would fall. This range is more selective that the two standard deviation limit. Only SYNNER's **Max Path Delay** was within one standard deviation of the designers' mean. Since SYNNER is only marginally able to attain the two standard deviation limit, it is clear that the system does not approach optimal designs.

	$\bar{x}$	$\bar{x} + s_x$	$\bar{x} + 2*s_x$	SYNNER
Nodes Actually Bound:	27.75	32.50	37.25	23
Total Packages Used:	24	26.94	29.88	28
Total Modules Used:	42.5	49.92	57.34	50
Total Spare Modules:	9	11	13	5
Total Power (MW):	5477	7183.19	8889.38	8525
Control Words:	70.5	72.23	73.96	71
Control Word Size:	46.75	54.55	62.35	63
Controller Address:	7	7	7	7
Max Control Path:	63.75	66.62	69.49	68
Max Path Delay (NS):	1135.88	1276.27	1416.66	1024.00
Raw Package Cost:	$ 19.13	23.87	28.61	28.07
Total Package Cost:	$ 91.13	104.07	117.01	112.07

Table 4-4: Change/TTL - Statistics

In order to justify the assumption that the designers represented a normal distribution, a χ^2 goodness of fit test was applied for the two categories (**Max Path Delay** and **Total Package Cost**) that were to be constrained during the experiment. The computation was performed using a method described by [Bevington 69] that develops the χ^2 statistic using the normal distribution then tests that hypothesis to determine to what degree it was true. The χ^2 for **Max Path Delay** was 3.61 which corresponds to a probability (or confidence level) of between 0.80 and 0.90 that such a value would be occur in the parent population that was normally distributed. Since a probability of approximately 0.5 would be expected for totally random data, the test indicates a good fit. The χ^2 for **Total Package Cost** was 3.74 which also corresponds to a probability of between 0.80 and 0.90 that the χ^2 value would be exceeded in a normally distributed parent population. Given the very small number data points, the χ^2 statistics indicate that the assumption of a normal distribution is valid.

In view of the experimental objective (minimize total package cost and maximum path delay), SYNNER's performance is remarkably good in this case. In fact, the **Maximum Path Delay** from SYNNER is the second best of all the designers. The **Total Package Cost** from SYNNER is the worst of all the designers but still well within the expected population of designers with similar capabilities. The **Control Word Size** is a parameter that is not considered during module selection.

4.6.2 Designer Transformations

The designers universally applied some form of the **Horizontal Join** transformation to both the arithmetic operators and to the relational operators. Joining the arithmetic operators into SN74181 ALUs not only eliminates nodes and packages directly, but it also has the side effect of reducing the number of inputs to the multiplexor. Three of the designers joined all arithmetic operators into a single ALU. This approach requires that one of the inputs to the ALU have a four way multiplexor to select one of the three constants or node # 6 as the input. One of the designers (2) took the approach that operators with the same inputs should be joined. This approach resulted in three ALUs, but it eliminated the requirement for multiplexors on the inputs. Node # 16 was reduced to a three input multiplexor. Designer 2 was the only one to merge the decrement operator (node # 134) with the SUM register (node # 15). The merge transformation resulted in the selection of SN74193 counters to implement node # 15. All other designers used SN74174 registers without any merged operation capability.

The most common join of relationals resulted in grouping comparators (SN7485) into two groups according to shared inputs. The shared inputs were then multiplexed. One of the designers used the special capabilities of SN7423 (expandable dual 4-input positive-nor gates with strobe) and SN7425 (dual 4-input positive-nor gates with strobe) to synthesize comparisons with constants having one or less bits true (0, 1, and 2). Use of the SN7425 appears to be acceptable, but the SN7423 appears to require an open collector drive for the expandable inputs.

SYNNER contains a horizontal join transformation (refer to Section 2.6.2) that merges arithmetic and relational operators. This transformation is the primary reason that SYNNER's performance fell within the population defined by the designers. SYNNER does not have the capability to recognize special cases of relational operations such as comparisons to zero. It would be worthwhile to enhance the Synthesis Equivalence Language to allow specifying transformations for such special cases. Another problem in SYNNERs performance occurred as a result of the extremely local view take of the selection process. The greater than or equal (GEQ) operators do not have a corresponding TTL database module. The SN7485 implements greater-than (GTR) and equal (EQL) with separate outputs. SYNNER recognizes that these outputs may be ORed to form the required GEQ operation. However, the evaluation finds that the penalty for mounting a package of OR gates exceeds the cost of implementing the operation another way (by testing for a carry from a subtract operation). SYNNER does check the spare module list during evaluation, but if no spare OR gates exist at the time of evaluation, the subtract operation is specified. A more global view of requirements in the

design might identify the pending requirement for packages that would leave spares useful for implementing operations.

4.7 Small PDP-8 Using Sandia Cells

Designers 1, 2, 3, and 4 selected modules for the small PDP-8 description using the Sandia Laboratories Standard CMOS Cell module set. The data will be presented and discussed as it was for the change mechanism.

4.7.1 Data Analysis

The first table shows the raw data as it was extracted from the design experiments.

DESIGNER	1	2	3	4
Total Bindable Nodes:	15	15	26	19
Nodes Actually Bound:	14	13	19	18
Percent Binding:	93%	87%	73%	95%
Total Packages Used:	226	264	796	193
Total Modules Used:	226	264	796	193
Total Spare Modules:	0	0	0	0
Module Utilization:	100%	100%	100%	100%
Control Words:	75	89	79	76
Control Word Size:	14	13	19	18
Controller Address:	7	7	7	7
Max Control Path:	30	37	31	24
Max Path Delay (NS):	950.0	1360.0	1200.0	1225.0
Raw Package Cost:	25911.77	30633.12	85340.64	30651.35
Total Package Cost:	57005.89	67392.86	187749.41	67432.20

Table 4-5: Small PDP-8/Sandia Cells - Raw Data

As with the change mechanism, certain inversions were universally missed by the designers. In addition, one of the designers missed the one bit LINK register and another designer used a two bit compare where a twelve bit compare was required. All but one of the designers dealt with the more obvious inversions.

Corrections were applied to the raw data to cause all nodes to be bound. As with the change mechanism, modules were added without any additional transformations to the basic design. The parameters from the resulting designs are shown in the table below.

DESIGNER	1	2	3	4
Total Bindable Nodes:	15	15	26	21
Nodes Actually Bound:	15	15	26	21
Percent Binding:	100%	100%	100%	100%
Total Packages Used:	227	266	844	209
Total Modules Used:	227	266	844	209
Total Spare Modules:	0	0	0	0
Module Utilization:	100%	100%	100%	100%
Control Words:	77	91	87	80
Control Word Size:	15	15	26	21
Controller Address:	7	7	7	7
Max Control Path:	31	38	38	25
Max Path Delay (NS):	990.0	1475.0	1350.0	1250.0
Raw Package Cost:	25967.93	30840.48	88260.96	32416.69
Total Package Cost:	57005.89	67392.86	187749.41	71316.72

Table 4-6: Small PDP-8/Sandia Cells - Consistent Data

It is obvious at this point that the **Total Package Cost** from Designer 3 is rather wildly different from the more closely grouped results of the other designers. There appears to be two reasons for Designer 3's large cost. First, in the case of two multiplexors (the two biggest ones in the design) he got concerned about the drive capability of the select drivers from the controller. He then chose to put buffer devices on the inputs of the multiplexors. In fact, the addition of those buffers should have been deferred to the control allocator where they could be put on the output of the control lines. The instructions to designers did not make any recommendations on how buffers should be applied. Therefore, Designer 3's decision was valid within the context of the experiment but it worked against his attempt to meet the cost constraint. However, the second problem was the major contributing factor to the large area. Designer 3 developed a four bit adder as a basic building block. He then used the adder to implement Node #44 (ADD2C, 13 bits), Node #71 (INCR, 13 bits), and Node #27 (ADD2C, 12 bits). All other designers combined nodes #44 and #71 and built add functions to exactly 13 bits rather than the 16 bits used by Designer 3. The other designers dealt with Node #27 as an incrementer or by designing Node #10 (PC, 12 bits) as a counter. Designer 3's decision to use a general adder as a basic building block is certainly a valid possibility, but once again, it worked against his attempt to meet the cost constraints.

The final transformation on design data puts the **Max Path Delay** on the same accounting

basis as SYNNER uses. As before, the **Total Power** parameter was added to the table.

DESIGNER	1	2	3	4	SYNNER
Total Bindable Nodes:	15	15	26	21	29
Nodes Actually Bound:	15	15	26	21	27
Percent Binding:	100%	100%	100%	100%	93%
Total Packages Used:	227	266	844	209	326
Total Modules Used:	227	266	844	209	326
Total Spare Modules:	0	0	0	0	0
Module Utilization:	100%	100%	100%	100%	100%
Total Power (μW):	17820	22725	52900	20840	29170
Control Words:	76	90	86	80	98
Control Word Size:	23	14	23	14	22
Controller Address:	7	7	7	7	7
Max Control Path:	31	38	38	25	39
Max Path Delay (NS):	990.0	1475.0	1350.0	1250.0	1750.0
Raw Package Cost:	25967.93	30840.48	88260.96	32416.69	40641.84
Total Package Cost:	77903.79	92521.44	264762.88	97250.07	121925.52

Table 4-7: Small PDP-8/Sandia Cells - Synner Accounting Basis

It should be noted that SYNNER missed two nodes during automatic processing. These two nodes were rotates. SYNNER has no capability to deal with rotates that appear in a design as discrete nodes. It can deal with rotates that are merged operations of registers. The discrete rotates in the small PDP-8 could be implemented by twisting the link connections to the multiplexor. If that had been done, no additional modules would be required in the design. Since the multiplexor already existed, no additional control bits or words would be required. Therefore, no correction was applied to SYNNER's results.

The designers were instructed to not make any module selection for the memory in the PDP-8 description. SYNNER makes a selection of a dummy memory that satisfies control allocator requirements. SYNNER'S **Total Bindable Nodes** and **Nodes Actually Bound** fields were each reduced by one to place the results on the same basis as the designers. The dummy memory does not add any cost, but it does add two control lines to the total. Therefore, SYNNER's **Control Word Size** was reduced by two from the reported value.

Designer 3's rather surprisingly large total area presents a problem for analysis and comparison. While it represents a valid data point, it does not appear to be well grouped with the other designers. In this case, the χ^2 goodness of fit test was applied first to the results of all four designers, then it was applied using only the three apparently well grouped designers. The χ^2 statistic for the **Max Path Delay** parameter and two degrees of freedom (four designers) was 3.69 which corresponds to a probability of 0.80-0.90 that a normal distribution

is appropriate. Eliminating Designer 3 from this computation gives a χ^2 with one degree of freedom (three designers) as 2.64 which corresponds to a slightly better 0.85-0.90 probability that the data is normally distributed. For the **Max Path Delay** parameter it appears that Designer 2, not Designer 3, was the cause of the lower probability in the first case. When the χ^2 statistic was computed for the **Total Package Cost** parameter, it was found that for four designers (two degrees of freedom) the χ^2 was 2.51 which corresponds to a 0.70-0.75 probability of being normally distributed. When Designer 3 was eliminated, the χ^2 was 2.53 for one degree of freedom. This value corresponds to an 0.85-0.90 probability of being normally distributed. Considering the low number of data points, the difference in probabilities is not particularly great and either set of data could be used for comparison to SYNNER'S results. The summary tables below do make the comparison both ways. Table 4-7 includes the data from all four designers. The mean ($\overline{x}$) of the designer parameters is given in the first column. The mean plus one standard deviation (which is the range within which 68% of the population would be expected) is given in the second column. The third column is the mean plus two standard deviations ($\overline{x} + 2 * s_x$). This is the upper limit of the range in which 95% of the members of this population should fall. The last column is SYNNER'S results.

	$\overline{x}$	$\overline{x} + s_x$	$\overline{x} + 2*s_x$	SYNNER
Nodes Actually Bound:	19.25	24.57	29.71	29.0
Total Packages Used:	386.50	692.43	998.36	326.0
Total Power (μW):	28571.25	44915.75	61260.25	29170.0
Control Words:	83.00	89.22	95.44	98.0
Control Word Size:	18.75	23.94	29.13	22
Controller Address:	7	7	7	7
Max Control Path:	35.5	38.82	42.13	39
Max Path Delay (NS):	1266.25	1472.14	1678.03	1750.0
Raw Package Cost:	44371.52	73759.62	103147.72	40641.84
Total Package Cost:	133109.55	221263.91	309418.27	121925.52

Table 4-8: Small PDP-8/Sandia Cells - Statistics/Designers 1, 2, 3, and 4

Of the significant parameters, SYNNER is within one standard deviation for **Total Power** and **Total Package Cost**. SYNNER is outside the sample population for **Control Words** and **Max Path Delay**. SYNNER does not have a particularly good reduction capability. Therefore, inverting links that are inserted as NOT nodes never get merged with devices that have inverting outputs. NOT nodes cause countable micro-operations to be inserted into the Micro-Operation Sequence. These micro-operations definitely add to the total **Control Word** count. Another problem was caused by SYNNER's synthesis of the EQL operator (Node #51). In conjunction with the change mechanism analysis it was mentioned that SYNNER does not have the capability to identify special cases of relational operators. The EQL

operator in this design makes a comparison to a constant of zero. Three of the four designers took advantage of that by NORing the twelve bits from Node #11 to generated the required boolean. SYNNER used an Exclusive OR then NORed the twelve outputs. In addition to the extra packages and extra micro-operation caused by the XOR, SYNNER synthesized the NOR as a series of two and three input OR gates followed by a NOT which is not a particularly good way to synthesize the NOR. The result was a chain of eight countable micro-operations (all in the maximum control path). Three of the four designers only contributed two micro-operations at this point. The other designer contributed three micro-operations from the EQL. Two problems contributed to the excess in **Max Path Delay**. The use of explicit NOT gates mentioned earlier add a minimal increment into the delay. However, a far worse contribution to the delay was caused by the handling of Node #27 (an adder used for incrementing) and Node #10 (the PC which carried a merged operation of INCREMENT). SYNNER is unable to identify the special condition of the adder used as an increment, so a 12 bit adder was assigned to the node. The PC (with its merged INCREMENT operation) was assigned counter modules. Three of the four designers handled the problem either by assigning counters to PC and eliminating the adder or by constructing special incrementers to replace the adder then using a cheaper (and faster) register for PC. The fourth designer used an adder, but he also used the non-counting registers for PC.

The second comparison (in the table below) gives the mean, standard deviation, $\overline{x} + 2 * s_x$, and SYNNER's results as before. The statistics were developed using data from Designers 1, 2, and 4.

	$\overline{x}$	$\overline{x} + s_x$	$\overline{x} + 2*s_x$	SYNNER
Nodes Actually Bound:	17.00	20.46	23.92	29.0
Total Packages Used:	234.0	263.14	292.28	326.0
Total Power (μW):	20461.67	22935.96	25410.25	29170.0
Control Words:	82.0	89.21	96.42	98.0
Control Word Size:	16.33	18.64	20.59	22
Controller Address:	7	7	7	7
Max Control Path:	31.33	37.84	44.35	39
Max Path Delay (NS):	1238.33	1481.04	1723.75	1750
Raw Package Cost:	29741.70	33103.56	36465.42	40641.84
Total Package Cost:	89225.10	99310.69	109396.28	121925.52

Table 4-9: Small PDP-8/Sandia Cells - Statistics/Designers 1, 2, and 4

In this case, SYNNER is outside the population in all interesting categories. The most important parameters (**Max Path Delay** and **Total Package Cost**) are both outside the desired two standard deviations of the mean. The major contributing factor seems to be the adder used to increment the PC. If that 12 bit adder were eliminated from SYNNER's design, the results would be as shown in the following table:

	$\bar{x}$	$\bar{x} + s_x$	$\bar{x} + 2*s_x$	SYNNER
Nodes Actually Bound:	17.00	20.46	23.92	28.0
Total Packages Used:	234.0	263.14	292.28	293.0
Total Power (μW):	20461.67	22935.96	25410.25	25690.0
Control Words:	82.0	89.21	96.42	98.0
Control Word Size:	16.33	18.64	20.59	22
Controller Address:	7	7	7	7
Max Control Path:	31.33	37.84	44.35	39
Max Path Delay (NS):	1238.33	1481.04	1723.75	1520
Raw Package Cost:	29741.70	33103.56	36465.42	33790.32
Total Package Cost:	89225.10	99310.69	109396.28	101370.96

Table 4-10: Small PDP-8/Sandia Cells - Statistics/SYNNER - Without Adder

From this change, the important parameters are now comfortably within the desired two standard deviations of the mean. Most of the parameters are just slightly above one standard deviation from the mean. Certain parameters still remain outside desired range. The control and maximum path parameters would be further reduced if the handling of inverters were improved and if the special case comparisons were dealt with correctly.

4.7.2 Designer Transformations

The major transformations used by the designers have already been mentioned during the analysis of data and will be summarized here. One highly specialized (and interesting) transformation has not been mentioned and will be discussed.

Three of the four designers used a **horizontal join** to eliminate one of the three adder/incrementers required in the original design. This resulted in a significant reduction of cell area. Designer 1 fabricated a counter from cells for the PC (Node #10) in order to eliminate the adder (Node #27) that was only used as an incrementer. Designers 2 and 4 each fabricated an incrementer from cells to implement the adder (Node #27). The incrementers were less costly than general adders.

Three of the four designers identified the sole relational operator as a special case (compare equal to zero) and utilized a minimum logical form of testing for any bit true.

Designer 2 took special interest in reducing the multiplexors. To this end, he developed an interesting "switchable link" that had the characteristic of inverting the data in one position or passing the data unchanged in the other position. In fact, the switchable link is a multiplexor with one inverting input and one non-inverting input. The contents of the accumulator (AC,

Node #14) was originally routed five places: three as inverted data and two as uninverted data. The data ended up at the main multiplexor inputs (Nodes #15 and #20) in three cases. The inverting link eliminated two of those inputs. Joining the adder and incrementer into a single adder resulted in reducing the most costly multiplexor (Node #15) to four inputs. A four input multiplexor is the largest that exists in the Sandia Cell module set, so it was possible to implement this node with modules one level deep rather than with the more costly cascaded multiplexor design used by SYNNER.

The general classes of transformations used by the designers as a group are similar to those available to SYNNER. The major areas that caused SYNNER problems were identification and handling of special cases. Hopefully, structuring and applying special case analysis to SYNNER would be a fruitful area for future research.

4.8 Change Mechanism Using Sandia Cells

At the outset of this chapter it was mentioned that one planned portion of the balanced design experiment failed to materialize. Only one designer from the "B" group completed the descriptions as requested. This event leaves no chance of statistical comparisons, but it is worthwhile to make a relative comparison of Designer 5's results against SYNNER's results. This section will compare the results for the Change Mechanism using Sandia Cells. The next section will compare the results of the Small PDP-8 using TTL modules.

The following table gives the comparison of measured parameters. The data has been placed on the same accounting basis as SYNNER uses. The changes made to the raw data

are the same as those discussed in Section 4.6.

DESIGNER	5	SYNNER
Total Bindable Nodes:	34	32
Total Bound:	34	32
Percent Binding:	100%	100%
Total Packages Used:	352	194
Total Modules Used:	352	194
Total Spare Modules:	0	0
Module Utilization:	100%	100%
Total Power (μW):	20260.00	15270.00
Control Words:	71	93
Control Word Size:	32	36
Controller Address:	7	7
Max Control Path:	63	90
Max Path Delay (NS):	4875.00	2845.00
Raw Area:	31613.56	26292.24
Total Area:	94840.68	78876.72

Table 4-11: Change Mechanism/Sandia Cells - SYNNER Accounting Basis

This is a rather disconcerting result. SYNNER's results are far better than Designer 5's. It was difficult to accept this at face value because Designer 5 is one of only two designers in the experiment with direct experience using Sandia Cells. However, a careful analysis of his design show two areas that caused Designer 5 to have large values for the **Total Area** and the **Max Path Delay**. The first problem was that the arithmetic operators were synthesized as they occurred in the original Path Graph. While SYNNER merged six operators into one, Designer 5 chose to leave them as six separate operators. This had the side effect of requiring a much larger multiplexor than SYNNER's implementation. The multiplexor contributed to the delay as well as to the total area. The second problem concerns synthesis of the multiplexors. Designer 5 chose to encode the multiplexor select lines. Unlike the TTL module set, Sandia Cells do not contain any multiplexors with built in decoders. Therefore, decoders must be fabricated if it is assumed that the control allocator encodes the multiplexor selects. SYNNER makes the assumption that the control allocator can handle unencoded multiplexors by assigning extra bits to the control word. Obviously there is an interesting (and unexplored) trade-off here. Designer 5's choice of encoded multiplexors might cause the *total* design (including the controller) to be lower by reducing the bit widths of the control words. However, SYNNER's choice to leave the multiplexors unencoded definitely reduces the data part cost and delay since the gates required for decoding contribute to both the cost and delay.

Designer 5 did the best job of anyone (or any program) in handling buffering to compensate

for CMOS delays that increase with capacitive loading. SYNNER does not consider the buffering problem. SYNNER's design is as general as possible and the various loading problems differ significantly with each logic family. It was decided that loading problems could best be identified and handled by programs that translate SYNNER's output to forms suitable for partitioning, layout, and routing for each logic family. For the purpose of comparing Designer 5's results to SYNNER's, the contributions of buffers to the area, delay, and power were removed.

4.9 Small PDP-8 Using TTL Modules

Designer 5 was alone in completing the Small PDP-8 using TTL modules. The comparison of his results to SYNNER's results are presented in the table below.

DESIGNER	5	SYNNER
Total Bindable Nodes:	17	17
Total Bound:	17	19
Percent Binding:	100%	89%
Total Packages Used:	51	53
Total Modules Used:	114	111
Total Spare Modules:	4	5
Module Utilization:	97%	96%
Total Power (MW):	7657.00	9234.00
Control Words:	95	113
Control Word Size:	35	35
Controller Address:	7	7
Max Control Path:	41	49
Max Path Delay (NS):	587.00	617.00
Raw Package Cost:	$ 29.80	44.46
Total Cost:	$ 182.80	203.46

Table 4-12: Small PDP-8/TTL Modules - SYNNER Accounting Basis

It can be seen that the results are relatively close together. As with the other Small PDP-8 design, SYNNER could not handle the rotate operators. This accounts for the two unbound nodes. These nodes would not add any cost or delay to the design.

The relative similarity in the measured parameters belies the major differences in the designs. The most significant differences were:

- Designer 5 eliminated the adder (Node # 27) used to increment the PC, then he implemented the PC (Node # 10) with a counter. As mentioned earlier, SYNNER was unable to recognize that the adder performed the same function as the merged increment operator in the PC. SYNNER implemented the adder and also assigned a counter to the PC.

- Designer 5 eliminated the increment(Node #71) and implemented the L and AC registers (Nodes #17 and #14) as counters. SYNNER merged the increment with the adjacent adder (Node #44).

4.10 Conclusions

The purpose of the design experiment was to determine how well SYNNER's performance compared to a group of relatively good designers and to identify additional transformations that could further enhance SYNNER's performance. The results from the experiment show that SYNNER comes quite near to the population of designers. In the Change Mechanism design using TTL modules, SYNNER was within two standard deviations of the mean for both the cost and delay. In the experiment using the Small PDP-8 and Sandia Standard Cells, SYNNER was well within two standard deviations of the mean for total area when all four designers were considered. While SYNNER's delay result was outside the 95% group, it was only 5% above the limit. When the worst design was eliminated from that experiment, SYNNER did not quite attain the limit for either the total area or the delay, but it was still quite close. One transformation was identified (elimination of a redundant adder) which SYNNER missed. Modifying SYNNER's results to reflect the effect of that transformation showed that it would have been within two standard deviations of the mean for both the total area and the delay if the adder had been eliminated. It should be noted that the two standard deviation limit is quite broad (which means that designs in that range can be far from optimal). A narrower statistical range of one standard deviation from the mean was included in the tables to show the upper bound of the range in which 68% of the designer population would fall. SYNNER rarely attained that range.

The two other combinations of the experiment (Change Mechanism with Sandia Cells and Small PDP-8 with TTL) had only one designer. There was no opportunity to draw statistical conclusions from these results, but a comparison of SYNNER's results to the lone designer's results provided interesting contrasts. In the Change Mechanism design, SYNNER appears to have performed better than the designer (primarily by identifying the possibility of merging arithmetic operators). In the Small PDP-8 experiment, the designer attained a lower total cost and delay but SYNNER was within 12% for cost and 6% for delay.

A number of specific transformations were identified that could enhance SYNNER's performance. The interesting thing about the transformations is that they are primarily related to special cases. Identifying an adder used as an incrementer and merging it into a counter would have proven the most beneficial in this set of designs. Identifying a comparison to zero would have probably been the next most important. From these results it would appear that

the next step for future work in this area would be developing a methodology for taking advantage of special opportunities in designs.

The identification of transformations that could enhance the capability of the LSMS system is quite useful. However, the major result of this chapter is that the experimental calibration has shown SYNNER's performance using the existing set of transformations to be quite comparable to the performance of human designers.

Five designers and a program have identified a few points in the design space of two descriptions with two module sets. These points (with one exception) have been near the optimum area of the experimental design space. From the onset of this research there have been questions regarding how much latitude the structural design level would have to position a description in the design space. There have also been questions regarding the shape of the design space. The next chapter attempts to answer these questions.

Chapter 5
Design Space Exploration

"When it is dark enough you can see the stars."

-- Ralph Waldo Emerson (1803-1882)

5.1 Introduction

In Chapter 4 a few points were obtained from the designers and from SYNNER that could be plotted to start defining a design space. For example, the cost of all change mechanism designs using TTL modules could be plotted against the delays for that design. The same costs could then be plotted against the power. If enough different points were available, they would start to identify a characteristic shape of the design space. The high cost of engaging 50 or 100 designers to generate the points has precluded obtaining actual design space information. With SYNNER it is now possible to obtain a large number of points for various descriptions. A single description/module set pair can be used to generate various points in the design space by varying the constraints, the weights placed on the constraints, and the transformations used while processing a description. The resulting points identify the *transformation design space.* The transformation design space will be a subset of the *absolute design space* for a given description/module set pair. There is no known method of determining the bounds of the absolute design space. The transformation design space is limited by the transformations available to synthesize a design. Chapter 4 showed that while SYNNER's set of transformations is not complete, it is complete enough to approximate the work of human designers. The transformation design space that SYNNER is able to explore has some interesting properties which when generalized indicate that the absolute design space may have a rather surprising shape.

Design space projections will be discussed for three constrainable parameters: cost, delay, and power. The design spaces were generated for each of the descriptions discussed in Chapter 4 and for a larger description of the PDP-8. A set of *predictors* will be derived from

the design space data. The predictors allow bounds to be placed on the minimum, mean, and maximum expected values of a design using information available before processing by SYNNER. They are useful for hand design processes as well as higher levels of automated design systems. Application of the predictors will be demonstrated using the description of the Manchester Mark-1.

There has been speculation [Lawson 78, Leive 77] that some type of summary information could be extracted solely from a module set and used to guide the design style selection and D/M allocation steps of CMU-DA. It will be demonstrated that module set summary information alone is not well correlated to processed designs. The predictors developed in this chapter are functions of *both* the module set and the designs.

5.2 The Descriptions

Three descriptions will be used for design space exploration: the Change Mechanism, the Small PDP-8, and a *real* PDP-8 processor description. The first two descriptions could be called "toy" descriptions since their size was purposely curtailed to convenience the designers but they each represent general classes of digital design problems. The existence of data points generated by the designers make these descriptions attractive choices for initial design space exploration. The larger PDP-8 description is included because its greater detail provides a richer set of opportunities to apply constraint tradeoffs. The larger PDP-8 also serves as a demonstration that SYNNER is able to process designs of reasonable size and complexity.

The initial parameters for the first two designs were presented in Chapter 4 and are reproduced below for convenience. The initial parameters for the Full PDP-8 are also presented here.

Change Mechanism:

```
Bindable Nodes: 32
Control Words:  72
Longest Path:   69
```

Small PDP-8:

```
Bindable Nodes: 17
Control Words:  89
Longest Path:   38
```

```
Full PDP-8:

Bindable Nodes: 55
Control Words: 251
Longest Path:   58
```

5.3 Module Sets

The module set used for processing a design has a large impact on the resulting design space. Not only are the cost, delay, and power parameters directly summed from values for individual modules, but the existence or absence of functions in a module set will direct the application of transformations or synthesis. A number of interesting statistics can be derived from the module set itself. These statistics may be viewed as being derived from a design comprised of exactly one node. If each device in the database is assigned to that node and the resulting overhead cost, power, and delay are measured, a number of data points occur. If the data points are paired three ways (cost-power, cost-delay, delay-power) three projections of the module set space can be plotted. The module set parameters and projections will be used for comparison to parameters and projections derived from the designs.

The module sets used for this research were the SN74XX TTL module set and the Sandia CMOS Cell module set. These module sets represent differences in packaging, power requirements, and cost. TTL is packaged for insertion into etched boards. Sandia Cells are designed to be placed (through photomask techniques) on silicon for LSI circuit implementation. The characteristics of these module sets will be discussed in the following sections.

5.3.1 TTL Module Set

The SN74XX module set was chosen for use in a module database because of its wide use in implementing designs. The module database is a subset of the full SN74XX module set marketed by Texas Instruments and other manufacturers.

Generally, devices were included in the database if they had totem-pole outputs. Open collector and tristate devices were not used for this research. One Shottky TTL device was included because a similar function was not available with the standard devices.

Thirty-one (31) TTL modules are included in this database. They range in complexity from the single gate SN7400 NAND to the SN74181 ALU. A list of devices included in the database

is included in Appendix E. The module set space projections for those devices are shown in Figures 5-1a, 5-1b, and 5-1c.

A number of statistics that may be derived from the module set data. The maximum-to-minimum ratios for each axis of the module set space are the first relationships to be discussed.

Package Cost	12.19:1
Overhead-Cost	1.57:1
Delay	33.33:1
Power	45.50:1

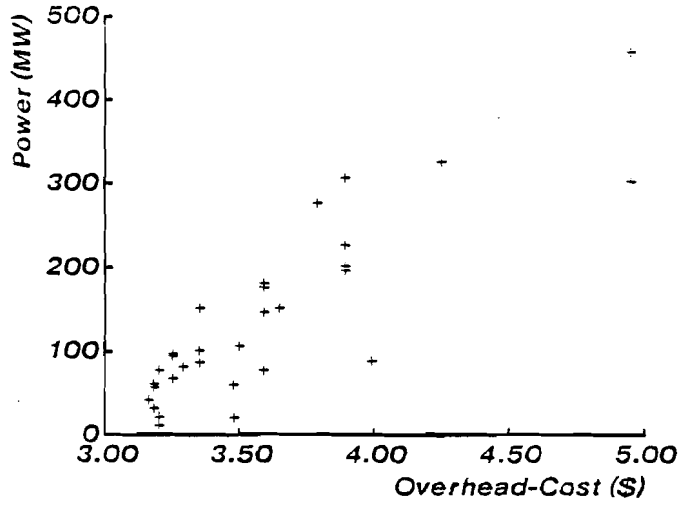
The package cost ranges from $0.16 to $1.95. The overhead-cost is derived by adding the $3.00 overhead to each package. This puts overhead-cost on the same basis as the **Total Package Cost** reported by SYNNER for all processed designs. The overhead figure dominates the numbers and reduces the maximum-to-minimum ratio to 1.57:1. The delay ratio is large because of the presence of the Shottky device (eliminating the Shottky device gives a ratio of 11.11:1). The power ratio is large because the SN74181 ALU requires 455 milliwatts while the (single module per package) SN7430 requires only 10 milliwatts.

If a straight line is fitted to the paired points for cost, delay, and power using linear regression, the goodness of fit of the line can be estimated by computing the correlation factor (R^2). The results of such computations are:

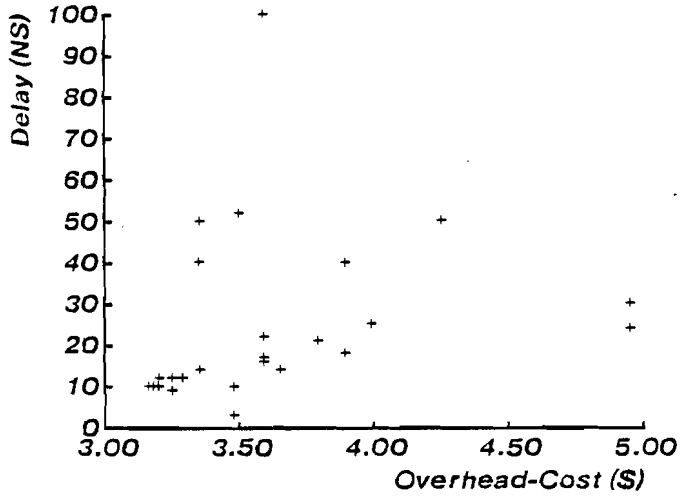
Overhead-Cost:Power (CP)	$R^2 = 0.746$
Overhead-Cost:Delay (CD)	$R^2 = 0.096$
Delay:Power (DP)	$R^2 = 0.163$

It is easy to see that the Overhead-Cost and Power are highly correlated while delay is not well correlated with either cost or power. This is not particularly surprising because the delay at this level is a primarily a function of gate delays. Power is primarily a function of the number of active devices which is related to the chip area. Overhead-cost is the sum of a constant for all packages and a device cost set by several marketing agencies. It would be expected that the final package cost would be some relatively constant percentage higher than the manufacturing cost. Manufacturing cost would be expected to be related to complexity (and size) of the device. These are the same factors that increase power requirements.

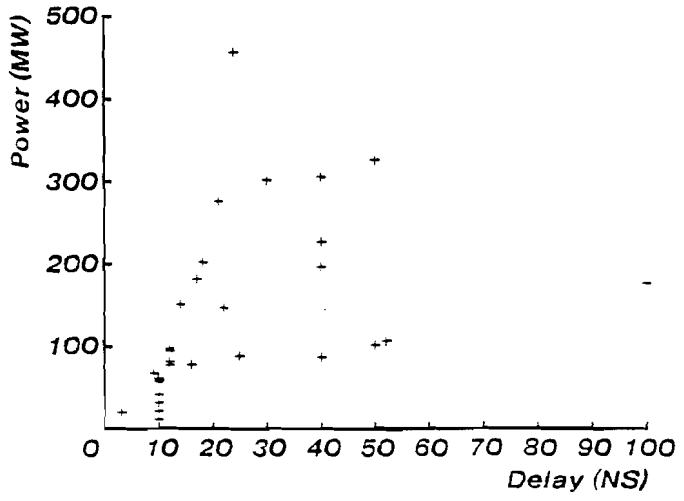
A wholy different set of relationships occur in the Sandia CMOS Cell module set discussed in the following section.



A. TTL Databook: Cost vs Power



B. TTL Databook: Cost vs Delay



C. TTL Databook: Delay vs Power

Figure 5-1: TTL Module Set Design Space Projections

5.3.2 Sandia CMOS Cell Module Set

The Sandia CMOS Cell module set was defined by Sandia Laboratories as a series of functional building blocks that could be positioned and routed with computer aids. These devices are representative of cell technologies under investigation by several organizations. The public domain access of Sandia information and an existing CMU contacts with Sandia Laboratories were the reasons for choosing these devices for this research.

Twenty seven (27) devices (listed in Appendix E) from the Sandia Cell library were included in this module database. Special driver and pad termination devices were not included. Certain special logic functions that could not be categorized were not included.

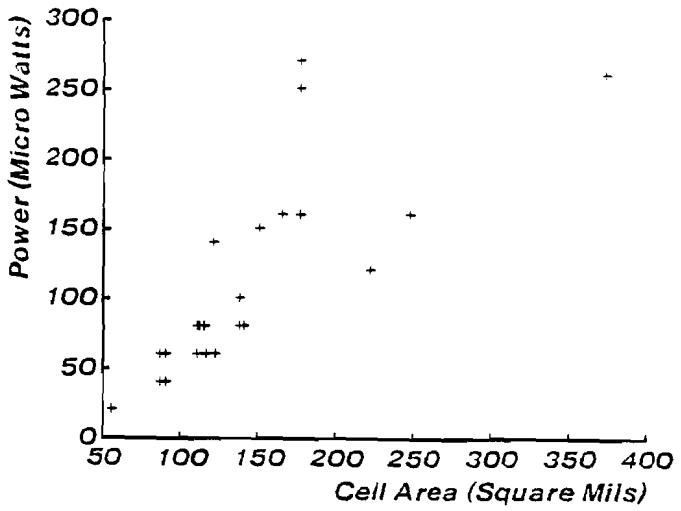
Two devices (ADD1 and ADD4) which are not Sandia devices were included in the module database. These entries incorporate the parameters of one bit and four bit adders as if they were built up from the other cells. Since SYNNER does not have the capability to correctly synthesize multiple output devices, these devices were defined as database entries. The database statistics were derived without using these fabricated devices. The data from all cell designs have been corrected to reflect the actual number of Sandia Cells required when either of these adders were used.

The maximum to minimum ratios for the Cell database are:

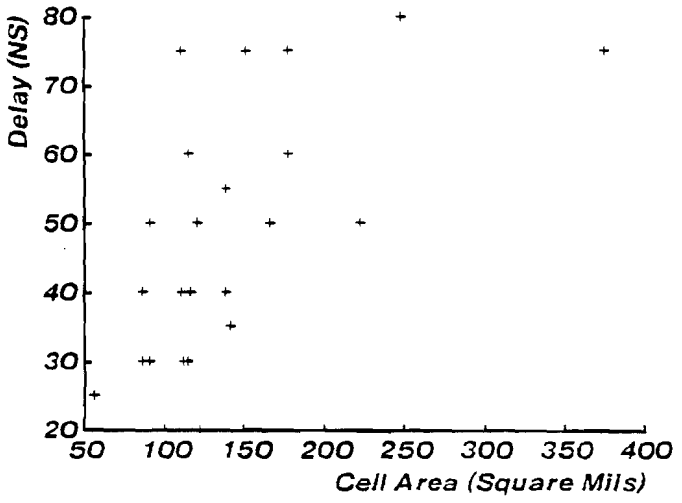
Overhead-Area	6.67:1
Delay	4.00:1
Power	13.50:1

The surprising information here is the lack of similarity between the ratios for Overhead-Area and Power. Power is not a figure included in the device descriptions published by Sandia Laboratories. Since the devices are CMOS, the power requirements are virtually negligible. In order to satisfy SYNNER's requirement for power information, Sandia personnel provided a value of 10 microwatts/transistor. That figure was given as a very rough estimate and was used without too much confidence that it reflects the actual device requirements. It does, however, provide a figure that is useful for relative comparisons. It appears that the area of individual transistors within the cells varies quite widely. Otherwise the maximum/minimum ratios for area and power would be far closer together.

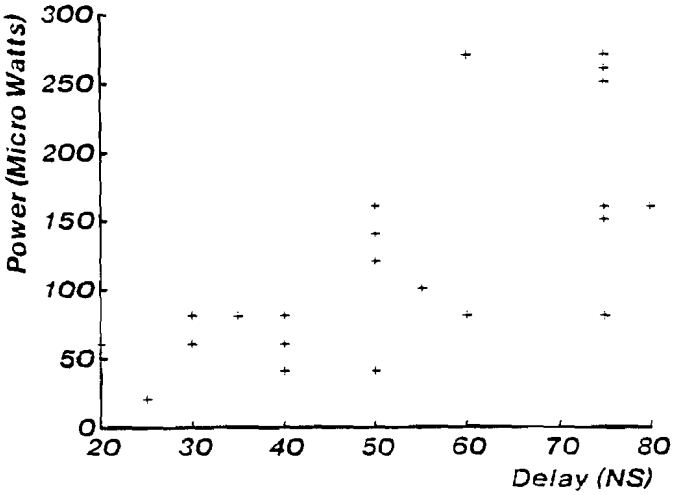
The correlation factors (R^2) for pairs of parameters also indicates that power and area are somewhat independent.



A. Cell Databook: Area vs Power



B. Cell Databook: Area vs Delay



C. Cell Databook: Delay vs Power

Figure 5-2: Sandia Cell Database Design Space Projections

Overhead-Area:Power (AP) R^2 = 0.571

Overhead-Area:Delay (AD) R^2 = 0.417

Delay:Power (DP) R^2 = 0.541

The pairs including delay are far better correlated than were the TTL module set pairs. This can be attributed to the lower functionality of the Sandia Cell module set. While TTL spans the range from SSI to MSI, Sandia Cells are strictly SSI devices. The lower functionality implies that there are less gate delays in the cell devices than occur in the TTL devices.

The module set parameters will be compared to similar parameters derived from the designs in the following sections and their usefulness as predictors will be explored.

5.4 Design Space Plots

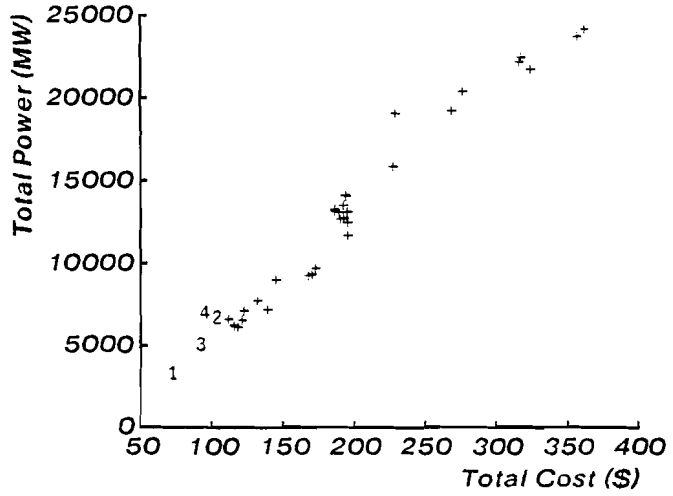
The plots of design space projections will be presented for the three designs and two module sets. These plots are grouped together at this point to give a view of the design space before deriving and analyzing the parameters associated with the plots.

The plots were extracted from 64 separate SYNNER runs for each design/module set pair. The constraints and transformations were varied for each run. Attempts were made to drive the the points toward all "corners" of the design space. Certain mixtures of constraints and transformations resulted in the same value as other mixtures so there are not necessarily 64 distinct points in each plot. The plus symbol (+) is used to denote SYNNER produced values. The plots for the Change Mechanism and the Small PDP-8 include the points generated by the designers for the experiment discussed in Chapter 4. Those points are identified by the designer number (1, 2, 3, 4, or 5) to distinguish them from the " + " points.

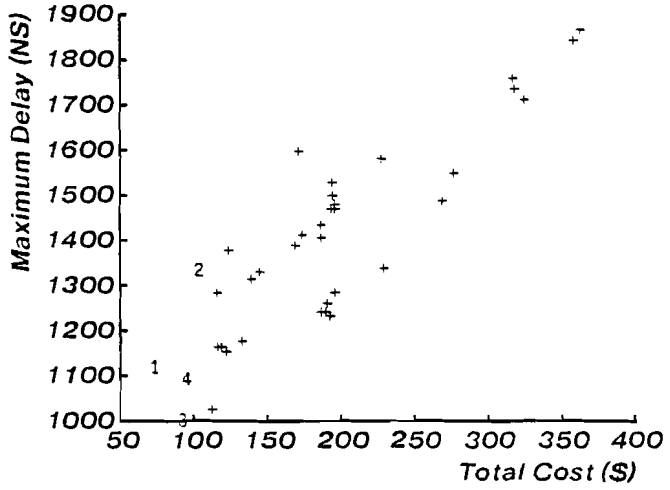
The plots will be presented in the order:

```
Figure 5-3a: Change Mechanism/TTL  - Overhead Cost vs. Power
Figure 5-3b: Change Mechanism/TTL  - Overhead Cost vs. Delay
Figure 5-3c: Change Mechanism/TTL  - Delay vs. Power

Figure 5-4a: Small PDP-8/TTL       - Overhead Cost vs. Power
Figure 5-4b: Small PDP-8/TTL       - Overhead Cost vs. Delay
Figure 5-4c: Small PDP-8/TTL       - Delay vs. Power

Figure 5-5a: Full  PDP-8/TTL       - Overhead Cost vs. Power
Figure 5-5b: Full  PDP-8/TTL       - Overhead Cost vs. Delay
Figure 5-5c: Full  PDP-8/TTL       - Delay vs. Power

Figure 5-6a: Change Mechanism/Cell - Overhead Cost vs. Power
Figure 5-6b: Change Mechanism/Cell - Overhead Cost vs. Delay
Figure 5-6c: Change Mechanism/Cell - Delay vs. Power

Figure 5-7a: Small PDP-8/Cell      - Overhead Cost vs. Power
Figure 5-7b: Small PDP-8/Cell      - Overhead Cost vs. Delay
Figure 5-7c: Small PDP-8/Cell      - Delay vs. Power

Figure 5-8a: Full  PDP-8/Cell      - Overhead Cost vs. Power
Figure 5-8b: Full  PDP-8/Cell      - Overhead Cost vs. Delay
Figure 5-8c: Full  PDP-8/Cell      - Delay vs. Power
```

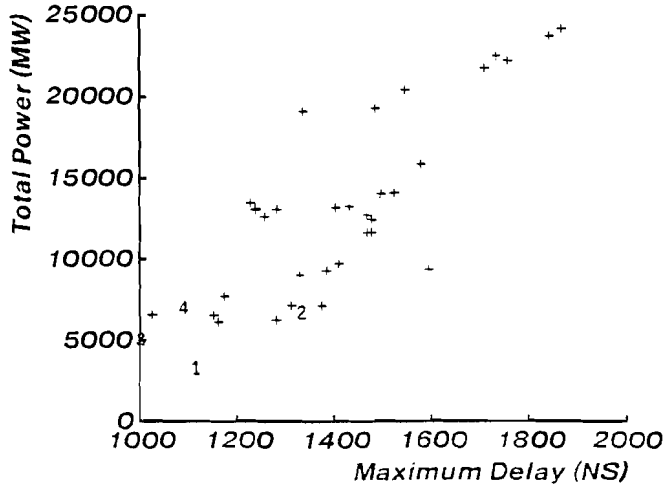
An interesting result from the projections is that none of the shapes approach a hyperbolic bound. The termination of point generation in the upper right area of each plot is due to SYNNER's termination of searching progressively worse design spaces. Clearly, the "worse" designs do not have a bound. Cost, power, and delay can all be made greater without limit if more devices are added (such as pairs of back-to-back NOT gates) which do not change the behavior of a design. SYNNER contains two limiting features which serve to keep synthesis searches in the realm of reasonable execution time. The first feature checks for and eliminates tight synthesis loops by determining if the same construct is being attempted recursively. The second feature puts an arbitrary upper limit of 25 on the depth of recursion in the synthesis process. This eliminates large loops or exceptionally long (possibly fruitless) searches. If designs were allowed to be made progressively worse, it appears that the points would have a roughly parabolic bound.



A. Change/TTL - Cost vs Power

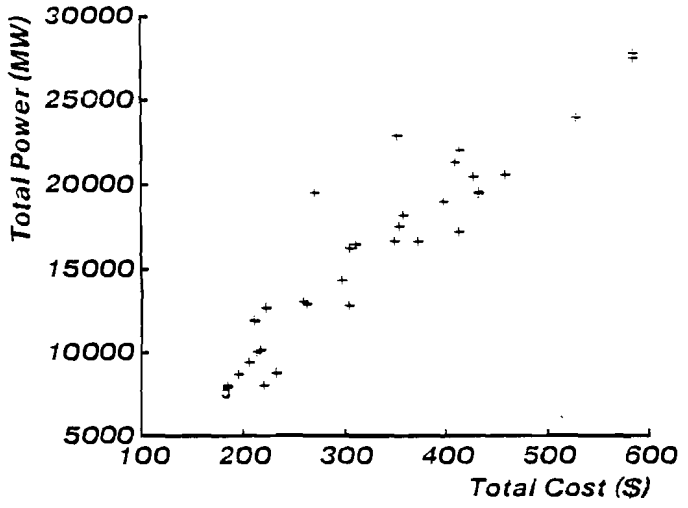


B. Change/TTL - Cost vs Delay

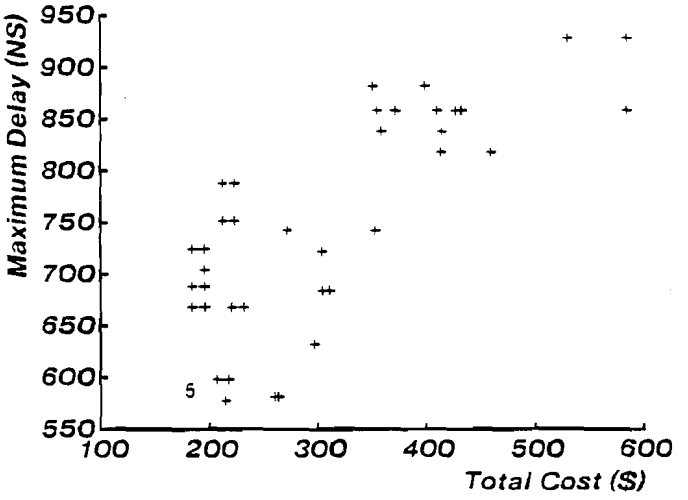


C. Change/TTL - Delay vs Power

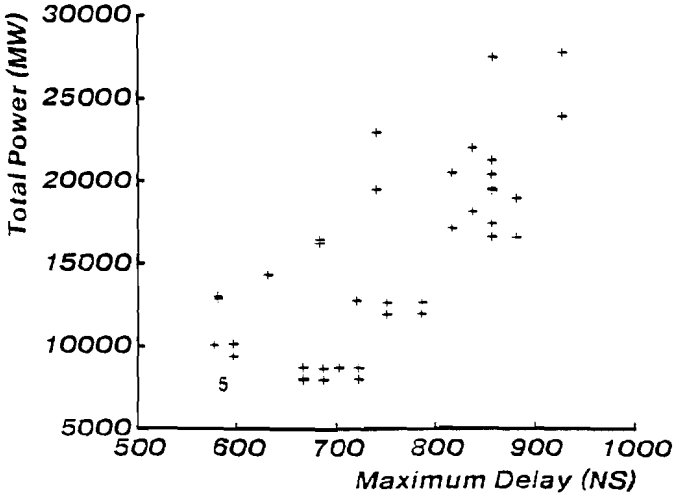
Figure 5-3: Change Mechanism/TTL Design Space Projections



A. Small PDP-8/TTL - Cost vs Power

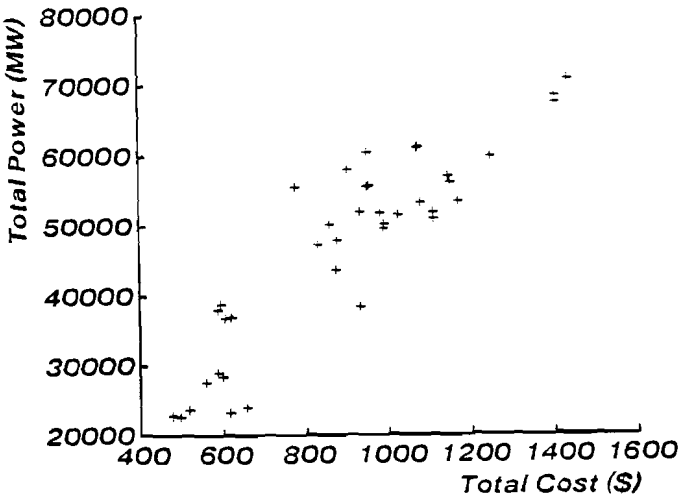


B. Small PDP-8/TTL - Cost vs Delay

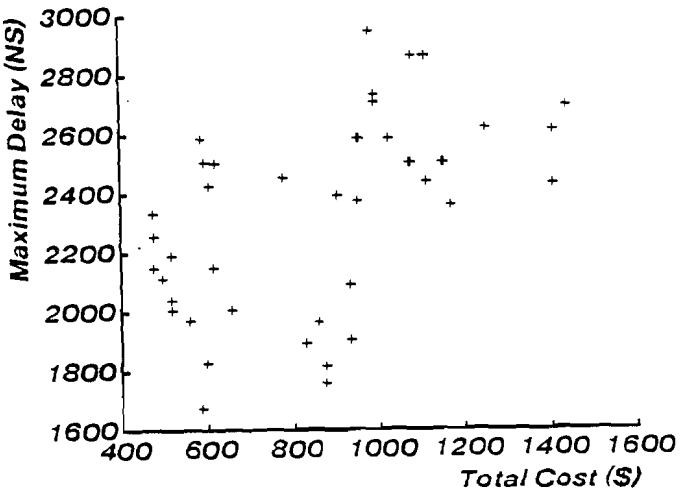


C. Small PDP-8/TTL - Delay vs Power

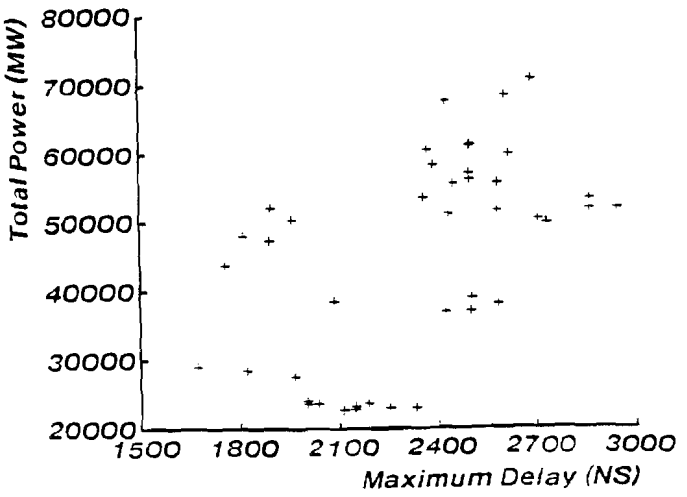
Figure 5-4: Small PDP-8/TTL Design Space Projections



A. PDP-8/TTL - Cost vs Power

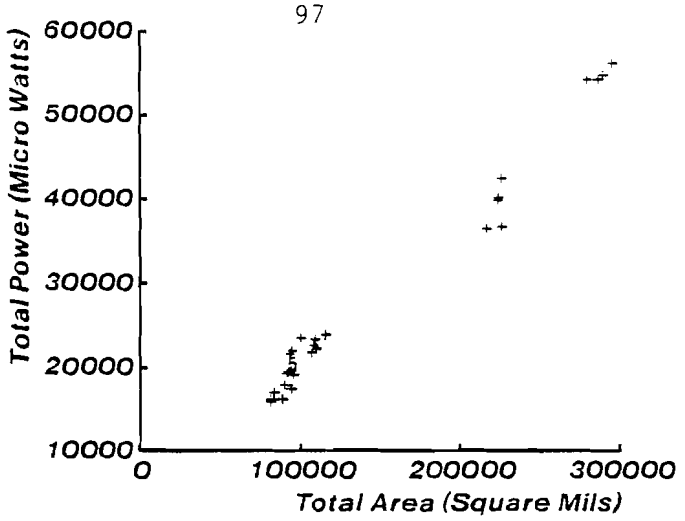


B. PDP-8/TTL - Cost vs Delay

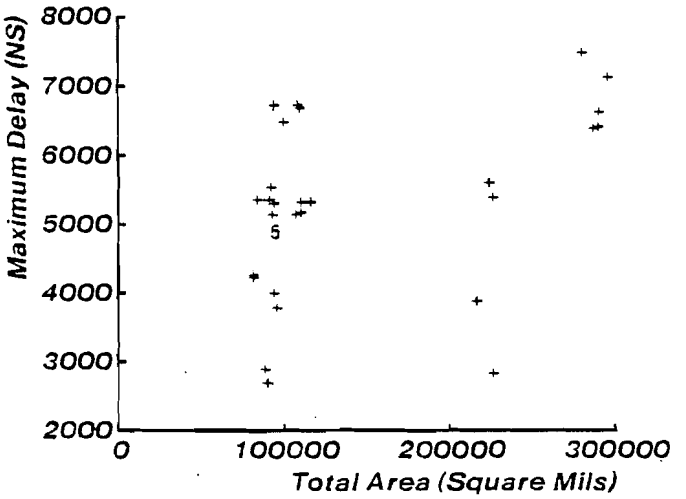


C. PDP-8/TTL - Delay vs Power

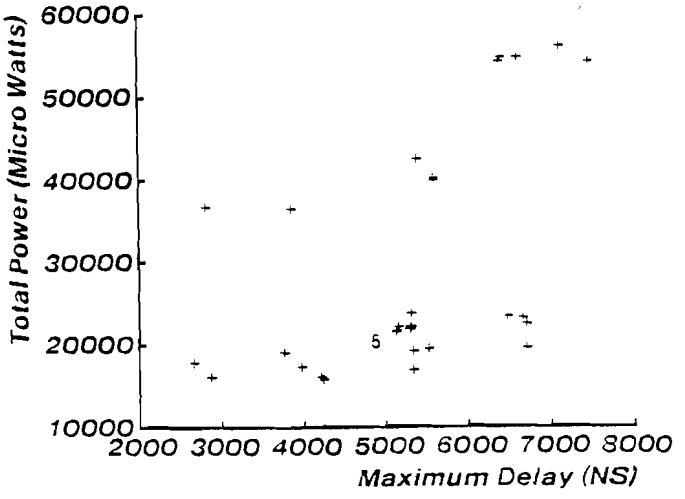
Figure 5-5: Full PDP-8/TTL Design Space Projections



A. Change/Cell - Area vs Power

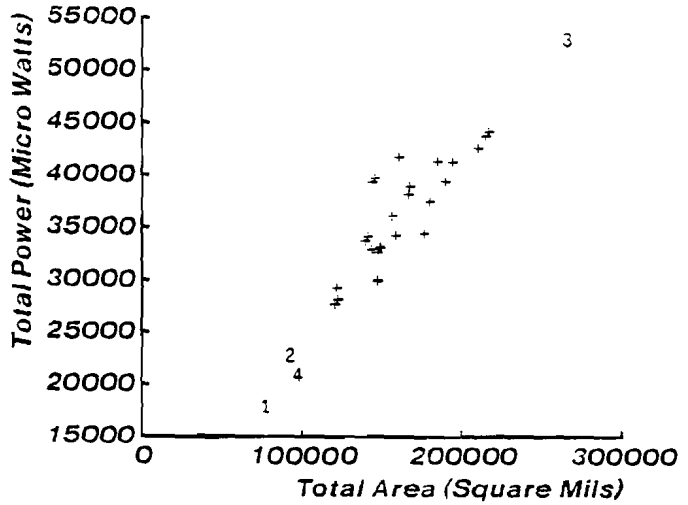


B. Change/Cell - Area vs Delay

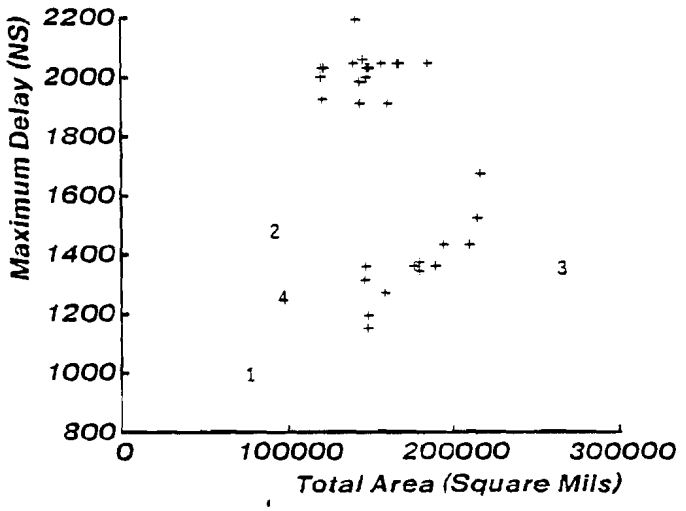


C. Change/Cell - Delay vs Power

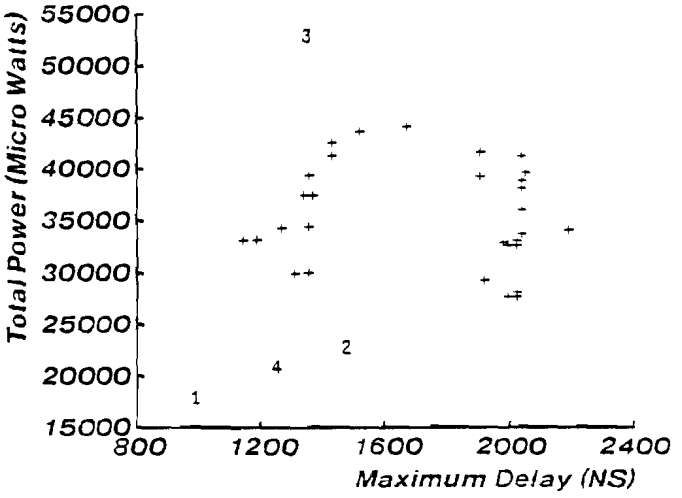
Figure 5-6: Change Mechanism/Cell Design Space Projections



A. Small PDP-8/Cell - Area vs Power

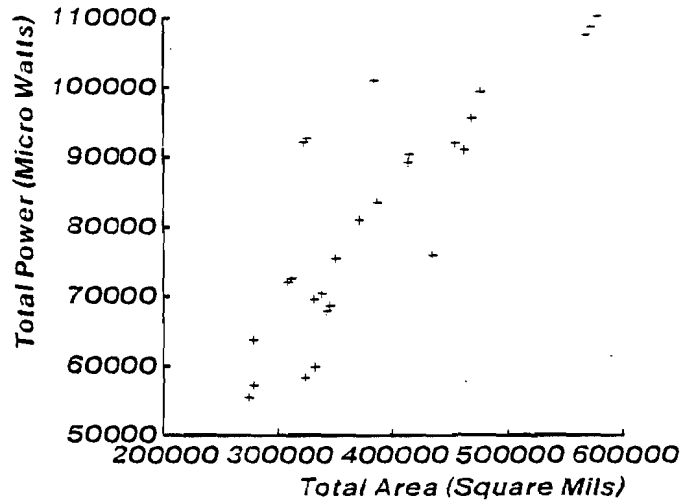


B. Small PDP-8/Cell - Area vs Delay

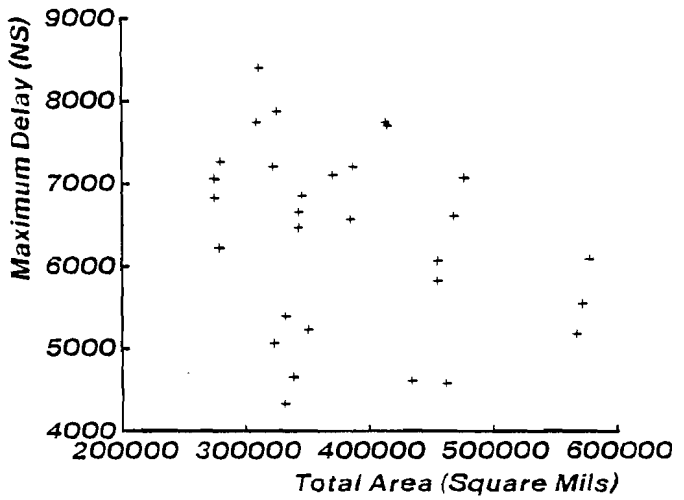


C. Small PDP-8/Cell - Delay vs Power

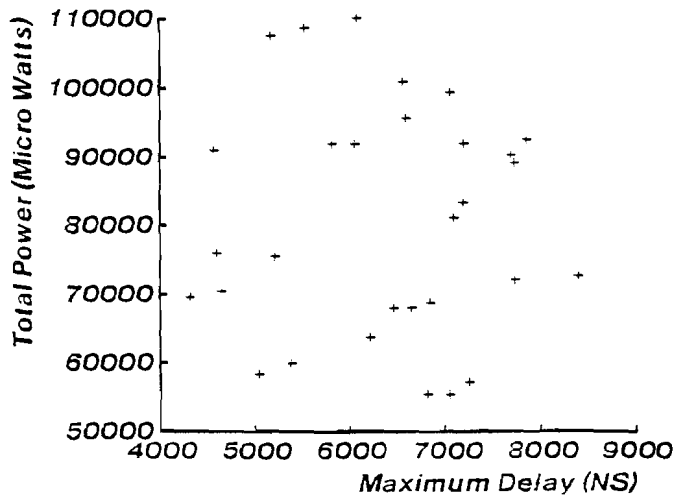
Figure 5-7: Small PDP-8/Cell Design Space Projections



A. PDP-8/Cell - Area vs Power



B. PDP-8/Cell - Area vs Delay



C. PDP-8/Cell - Delay vs Power

Figure 5-8: Full PDP-8/Cell Design Space Projections

5.5 Design Space Shape Analysis

The general result of a parabolic bound to the design space projections is interesting and it naturally leads to questions regarding prediction of the shape or bounds of a design space before a design is processed. If a set of predictors could be found to estimate the maximum, mean, and minimum expected for the cost, delay, and power of a design, it would enable higher levels of the design system (such as the Design Style Selector and D/M allocators) to maneuver designs into the most favorable state for processing by the LSMS level. If the spread of design space could be estimated, it might be possible to determine which parameters (cost, delay, or power) may be independently constrained. Certain useful predictors can be extracted from the design space studies. There are several relationships that appear less promising but also deserve discussion.

5.5.1 Correlation Comparisons

The correlation factors measure the independence of pairs of constrainable parameters (cost-power, cost-delay, and delay-power). The correlations (R^2) for data measured from the three TTL designs are shown in Table 5-1. The mean and standard deviation for each column is included in the table. The correlation factors for the TTL module set are reproduced in the last row. If a pair of parameters are found to be independent, it would indicate that a final position in the design space would respond to separate constraints on the parameters. If a pair of parameters are correlated, it would indicate that only one of the parameters need be constrained since the other would directly follow. The column headers for all tables relating statistics of paired points are:

TTL Designs:

- CP - Overhead-Cost/Power
- CD - Overhead-Cost/(Maximum Path) Delay
- PD - Power/(Maximum Path) Delay

Sandia Cell Designs

- AP - Overhead-Area/Power
- AD - Overhead-Area/(Maximum Path) Delay
- PD - Power/(Maximum Path) Delay

The mean and standard deviation were computed for each column and are shown in the fourth and fifth row.

	CP	CD	DP
Change Mechanism:	0.969	0.766	0.679
Small PDP-8:	0.880	0.670	0.582
Full PDP-8:	0.876	0.403	0.427
Mean	0.908	0.613	0.563
Standard Deviation	0.053	0.188	0.127
TTL Module Set:	0.746	0.096	0.163

Table 5-1: TTL Designs - Correlation (R^2) Factors

The uniformity of the correlation factors for the CP column is possibly the most interesting feature in Table 5-1. The CP correlation factor for the TTL module set taken alone is somewhat lower than for any of the designs but it is high enough to indicate a reasonable correlation of cost and power. Modules selected for these designs are more highly correlated in cost and power than randomly selected modules would be. This trend is continued for the cell designs (shown in Table 5-2).

	AP	AD	DP
Change Mechanism:	0.975	0.154	0.263
Small PDP-8:	0.740	0.179	0.007
Full PDP-8:	0.743	0.090	0.000
Mean	0.819	0.141	0.090
Standard Deviation	0.135	0.046	0.150
Cell Module Set:	0.571	0.417	0.541

Table 5-2: Cell Designs - Correlation (R^2) Factors

Cost and power are measurements are summed for all packages used in a design. The maximum path delay measurement does not reflect the entire design. It is the sum of the delays associated with the longest possible execution path through the micro-operation sequence for the design. It is interesting that both data pairs which include the maximum path

delay measurement (CD and DP) show relatively high correlation factors. The TTL module set R^2 factors indicate that these pairs are almost totally independent. The corresponding (AD and DP) columns for the cell designs (Table 5-2) show a reverse trend. The correlation factors for the designs are actually lower than for the module set. Because the delay measurements are a function of the control structure of a design, correlation factors including delay would be expected to be rather random. However, the opposite trends for the two module sets begs for an explanation since the same designs are involved.

The difference in the module sets is related to the complexity of logic functions represented. TTL ranges from SSI through MSI functions. The delay through the MSI modules is strictly a function of their logic design (gate delays and feedback paths) while the power (and apparently the cost) increase with the size of the device. Therefore, the delay would not be expected to be well correlated with cost or power. In the Sandia Cell module set, there are only SSI devices. Most of the devices only have single gate delays. The most complex devices are the counter registers which involve several gate delays and feedback. However, there are no modules that approach the complexity of the large multiplexors or ALUs of the TTL module set. Since the majority of devices in the Sandia Cell module set have a low number of gate delays, a higher correlation would be expected between area, power, and delay. Compare the module set space plots of delay versus power for TTL (Figure 5-1c) and Sandia Cells (Figure 5-2c). The delay versus power data points of the TTL module set are rather randomly distributed. The same data points for the Sandia Cell module set indicate that the delays fall into distinct lines at five nanosecond intervals with the majority being grouped at 30, 40, 50, and 75 nanoseconds. The problem then appears to be related to the distribution in the module sets. The result would be that the module set correlation factors (R^2) are simply not good predictors for any relationships involving delay although they might be applied to cost and power which are both related to module count.

Correlations extracted from the module sets do not seem to correspond to correlations extracted from the design space for either cost-delay or delay-power parameter pairs. The correlation correspondence for the cost-power parameter pair is not especially crisp either. It appears to be more accurate to draw conclusions from the information extracted from the processed designs. This differs from the earlier speculation that summarized module set information would be sufficient to characterized a design. Cost and power are so highly correlated for designs processed with TTL (0.908 mean for the three designs) that either parameter may be constrained and the other parameter will follow that constraint. Pairing either cost or power with delay results in moderate correlation. This indicates that constraints placed on delay are moderately dependent on constrained cost or power. Designs processed using Sandia Cells also show rather high correlation between cost and power (0.819 mean for

the three designs). Delay is almost totally independent of either cost or power when Sandia Cells are used.

A different view of the data will be obtained in the next section by analyzing the ratios of the maximum and minimum points for each parameter. These maximum-to-minimum ratios are interesting in their own right, but they are also necessary for developing the maximum and minimum predictors in Section 5.5.4.

5.5.2 Maximum to Minimum Ratios

If the shape of each design space projection is taken to be roughly parabolic, maximum-minimum ratios would not occur because the design space would not have an upper bound. SYNNER closes the search space as a practical feature which causes the design space plots appear elliptic rather than parabolic. The ratios of maximum-to-minimum points for each axis (cost, delay, and power) is a measure of the range in which SYNNER is allowed to manipulate designs.

Figures 5-3 and 5-4 summarize the maximum-to-minimum ratios for designs processed with the TTL and Sandia Cell module sets respectively. The mean and standard deviation are computed for the cost, delay, and power columns and are displayed as the fourth and fifth rows in the table. The last row in each table reproduces the module set maximum-to-minimum ratios developed in Section 5.3 for comparison to values extracted from the designs.

	Cost	Delay	Power
Change Mechanism:	3.23	1.82	3.99
Small PDP-8:	3.17	1.61	3.52
Full PDP-8:	3.02	1.76	3.13
Mean	3.14	1.73	3.55
Standard Deviation	0.11	0.11	0.43
TTL Module Set:	1.57	33.33	45.50

Table 5-3: TTL Designs - Maximum/Minimum Ratios (MMR)

The mean of the cost maximum-to-minimum ratios is 3.14 to 1 for the three designs processed with the TTL module set. The clustering of these numbers is quite good (the

standard deviation is seen to be 0.11). Therefore, it appears that the mean value is a fairly accurate representation of the cost maximum-to-minimum ratios for any of these designs. Delay and power are also well grouped. The mean value of the delay maximum-to-minimum ratios is 1.73 to 1 which is significantly different from the cost ratio. This indicates that synthesis of control steps in the longest control path is not proportional to synthesis of data part nodes for TTL. That is not particularly surprising because MSI modules such as comparators, adders, or ALUs which directly implement data part nodes (without any additional control steps) are often more expensive than the mean cost of the module set. However, the SSI module that may be used (under different constraints) to implement complex functions are often less expensive than the module set mean cost and generally require the addition of one control step for each synthesized node. The maximum-to-minimum ratio for power is 3.55 to 1 which is the largest ratios of the three parameters. It is similar to the ratio for cost. This reinforces the conclusion that cost and power are highly correlated. The module set maximum-to-minimum ratios shown in the last row (and discussed in Section 5.3.1) are not similar to the ratios for the designs. The cost ratio is damped by the $3.00 overhead included before computation. The actual package costs (excluding the $3.00 overhead) have a ratio of 12.19 to 1. Delay and power for the module set each have large maximum-to-minimum ratios but these occur because of unique points at the extreme range. The SN74181 ALU requires 455 milliwatts of power. The next closest device (the SN74191) uses 305 milliwatts. Figure 5-1c shows the distributions of delay and power for the TTL module set.

	Area	Delay	Power
Change Mechanism:	3.63	2.80	3.56
Small PDP-8:	1.79	1.91	1.60
Full PDP-8:	2.10	1.94	1.99
Mean	2.51	2.22	2.37
Standard Deviation	0.99	0.51	1.05
Cell Module Set:	6.67	4.00	13.50

Table 5-4: Cell Designs - Maximum/Minimum Ratios (MMR)

Table 5-4 shows the maximum-to-minimum measurements for the Sandia Cell module set. The results differ from the TTL case. All of the mean ratios are in the neighborhood of 2.4 to 1. SYNNER appears to have less range in the manipulation of cell designs than it had for TTL designs. The standard deviations indicate that the results are less uniform for cells than they

were for TTL. In the case of the change mechanism, the maximum-to-minimum ratio for area was larger than any cost ratio for TTL designs. This can only be a reflection on the exceptionally high proportion of arithmetic operators in that rather limited design and it shows that SYNNER is able to exercise a wide range of options in synthesizing arithmetic operators. Designs that are more homogeneous (such as the large PDP-8) approach the mean value of SYNNER's capability.

The mean maximum-to-minimum ratio for delay in cell designs is similar to the mean ratios for cost and power. This would seem to be reasonable since there is an absence of MSI devices in the Sandia Cell module set. Complex functions must synthesized from the existing SSI devices, and those devices usually add one control step per synthesized path graph node.

As in the case of the TTL module set, the maximum-to-minimum values for the Sandia Cell module set do not seem to reflect any of the trends shown for actual designs. This result is important because it indicates that summarized module set information may not reflect the trends found in processed designs.

The three designs used in this research have produced well grouped maximum-to-minimum ratios for cost, delay, and power. Section 5.5.4 will make use of these ratios and the expressions developed in the next section to derive numeric predictors for the maximum and minimum expected cost, delay, and power of arbitrary designs.

5.5.3 Predictor Development

In this section expressions will be derived for predictors of estimated bounds on cost, delay, and power for arbitrary designs. Numeric values will be evaluated from the expressions in Section 5.5.4. Predictions for arbitrary designs can be made by multiplying the numeric predictors by parameters than can be extracted from designs at the register transfer or functional level descriptions.

Cost (or area) and power values are summed over an entire design. They are functions of the number of packages used in a design. Unfortunately, the number of packages is not available until after a design has been processed at the LSMS level. However, cost and power can be related to the number of bindable nodes and the bit width of the major data path. Both of those parameters have values in the earliest forms of description. Delay is related to the initial number of control words in the longest control path.

Bindable nodes are path graph (or schematic) entities that will require assignment of

modules to implement the design. Registers and operators are examples of bindable nodes. Constants and links are examples of nodes that do not require modules for implementation. The major data path bit width is not always so clear. For example, should this parameter be twelve (12) bits in the PDP-8, or should it be thirteen (13) bits? A twelve bit data path agrees with the memory and the Accumulator bit widths but a thirteen bit data path agrees with some of the operators and the combined Link/Accumulator bit widths. The approach taken here was to count the data path links of each bit width in the design. The number of links was multiplied by its bit width and the maximum value was selected. The PDP-8s both have 12 bit data paths by this process.

It will be helpful to define a few symbols that can be used in the derivation of expressions for the predictors. The attributes that can be extracted from descriptions will be called:

N Initial number of bindable nodes.
B Major data path bit width.
L Initial number of control words in the longest control path.

The mean cost, delay, and power values extracted from a number of processed designs will be named:

x_c Mean Measured Cost.
x_d Mean Measured Delay.
x_p Mean Measured Power.

The total cost and power of a design are proportional to the number of packages used to implement the design. The total number of packages required is proportional to the number of path graph nodes that require modules and to the bit width of those nodes. The major data path bit width of a design is used as an estimate of node bit width. If the mean measured cost and power parameters (x_c and x_p) are normalized by the product of the initial number of bindable nodes (N) and the major data path bit width, the result is:

$x_{cn} = x_c/(B * N)$ Normalized Design Cost.

$x_{pn} = x_p/(B * N)$ Normalized Design Power.

Delay is proportional to the number of control words in the maximum path. The initial number of control words in the maximum path of a design is the estimate used to normalize the mean design delay:

$x_{dn} = x_d/L$ Normalized Design Delay.

These normalized parameters have units that depend on the module set used to implement the design:

For TTL:

x_{cn} Dollars per (bindable-node * major-data-path-bit-width)
x_{dn} Nanoseconds per maximum-control-path-control-word
x_{pn} Milliwatts per (bindable-node * major-data-path-bit-width)

For Cells:

x_{cn} Square-Mils per (bindable-node * major-data-path-bit-width)
x_{dn} Nanoseconds per maximum-control-path-control-word
x_{pn} Microwatts per (bindable-node * major-data-path-bit-width)

The general symbol used to refer to any of the preceding normalized parameters is:

x_{nn} Any of: x_{cn}, x_{dn}, or x_{pn}

The general names for cost, delay, and power predictors will be:

X_c Cost predictor.
X_d Delay predictor.
X_p Power predictor.

X_n Any of: X_c, X_d, or X_p.

Predictors for the mean values (center of the design space) can be computed by taking the means of the normalized parameters. If the symbol $X_n^{=}$ is designated to stand for the mean predictor for any of cost, delay, or power, then:

$$X_n^{=} = (1/n) \Sigma x_{nn}$$

A mean predictor can be considered to be the average value of the maximum and the minimum predictor:

$$X_n^{=} = (X_n^{+} + X_n^{-})/2 \qquad (5.1)$$

The maximum-to-minimum ratio can be defined as:

$$MMR = X_n^{+}/X_n^{-}$$

From those relationships, the expressions for computing the minimum and maximum predictors can be isolated:

$$X_n^{-} = (2 * X_n^{=})/(MMR + 1) \qquad (5.2)$$

$$X_n^{+} = MMR * X_n^{-} \qquad (5.3)$$

The measured maximum-to-minimum ratios (MMRs) will be used to develop X_n^{-} and X_n^{+}.

Those expressions for $X_n^=$, X_n^-, and X_n^+ are reasonable, but not exact predictors. It is possible to extend the range systematically by including the standard deviations with the various parameters. If all standard deviations are included, the bound area is so large that it does not provide much information. After several attempts get a better prediction of the actual data points, the predictors as derived appear to be as good or better than any of modified predictors. Numeric predictors will be derived from the equations as they stand.

5.5.4 Numeric Predictors

The equations developed in the last section will be applied to measurements taken from the three designs. Separate predictors will be developed for the TTL and Sandia Cell module sets.

Table 5-5 lists the N, B, and L factors. These values were derived directly from functional level path graphs produced by the distributed D/M allocator. The first row for each design contains the three parameters. The second row for each design (labeled "Factors") shows the computations for the N * B factor and reproduces the L factor.

	Bindable Nodes (N)	Major Data Bits (B)	Max Control Path Steps (L)
Change Mechanism:	32	5	69
Factors	32 * 5 = 160		69
Small PDP-8:	17	12	38
Factors	17 * 12 = 204		38
Full PDP-8:	55	12	58
Factors	55 * 12 = 660		58

Table 5-5: Designs Normalizing Factors

Table 5-6 summarizes the mean value of the cost, delay, and power measurements for the three designs processed using TTL modules. The values were computed by taking the mean of each parameter from the data produced by the 64 SYNNER runs that were used to plot the design space projections of Figures 5-3, 5-4, and 5-5.

	Cost($) x_c	Delay(ns) x_d	Power(mW) x_p
Change Mechanism:	190.15	1399.75	12109.09
Small PDP-8:	317.47	757.50	15364.28
Full PDP-8:	854.06	2310.19	43911.22

Table 5-6: TTL Designs - Measured Mean Values

The mean measured values are normalized by the appropriate (B * N) or (L) factors and the results are tabulated in Table 5-7. For example, dividing the change mechanism cost ($190.15) by the change mechanism B * N factor gives the value (1.19) shown in Table 5-7. The mean of the normalized values is then determined. Those values are the respective $X_n^=$ predictors (Equation (5.1)) for TTL designs.

	Cost x_{cn}	Delay x_{dn}	Power x_{pn}
Change Mechanism:	1.19	20.29	75.68
Small PDP-8:	1.56	19.93	75.32
Full PDP-8:	1.29	39.83	66.53
Mean (Normalized)	1.35	26.68	72.51
Standard Deviation	0.19	11.39	5.18

Table 5-7: TTL Designs - Mean Predictors ($X_n^=$)

The development for cell implemented designs is totally analogous to the preceding development for TTL designs. Table 5-8 summarizes the mean values of the measured

parameters.

	Area x_c	Delay x_d	Power x_p
Change Mechanism:	150037.70	5053.98	28460.00
Small PDP-8:	158798.00	1731.17	35045.63
Full PDP-8:	382366.70	6355.39	78724.06

Table 5-8: Cell Designs - Mean Measured Values

Table 5-9 gives the results of dividing the entries in Table 5-8 by the appropriate design factors from Table 5-5. The mean of the columns results in the Sandia Cell $X_n^=$ predictors.

	Area x_{cn}	Delay x_{dn}	Power x_{pn}
Change Mechanism:	937.74	73.25	177.88
Small PDP-8:	778.42	45.56	171.79
Full PDP-8:	579.34	109.58	119.28
Mean (Normalized)	765.17	76.13	156.32
Standard Deviation	179.57	32.11	32.22

Table 5-9: Cell Designs - Mean Predictors ($X_n^=$)

Tables 5-10 and 5-11 summarize the minimum, mean, and maximum predictors determined by applying Equations (5.2) and (5.3). For example, the TTL maximum-to-minimum ratio for delay (from Table 5-3) is 1.73. The $X_d^=$ value from Table 5-7 is 26.68. Applying equation (5.2) using those values, the X_d^- entry for Table 5-10 is found to be:

$X_d^- = (2 * 26.68)/(1.73 + 1) = 19.55$

This value is the minimum predictor for delay in TTL designs and is entered in the first row and

second column in the table.

	Cost X_c	Delay X_d	Power X_p
X_n^- (Minimum):	0.65	19.55	31.87
$X_n^=$ (Mean):	1.35	26.68	72.51
X_n^+ (Maximum):	2.05	33.81	113.15

Table 5-10: TTL Designs - Predictors

	Area X_c	Delay X_d	Power X_p
X_n^- (Minimum):	435.99	47.29	92.77
$X_n^=$ (Mean):	765.17	76.13	156.32
X_n^+ (Maximum):	1094.35	104.97	219.87

Table 5-11: Cell Designs - Predictors

The predictors have the following units:

For TTL:

X_c Dollars per (bindable-node * major-data-path-bit-width)
X_d Nanoseconds per maximum-control-path-control-word
X_p Milliwatts per (bindable-node * major-data-path-bit-width)

For Cells:

X_c Square-Mils per (bindable-node * major-data-path-bit-width)
X_d Nanoseconds per maximum-control-path-control-word
X_p Microwatts per (bindable-node * major-data-path-bit-width)

The method of using these predictors will be demonstrated in the next section where the predicted and measured design space projections are described.

5.6 Predicting a Fourth Design

In order to get some indication of how well the predictors work, they were applied to the Manchester Mark-1[7]. From the viewpoint of this research, the most interesting aspect of the Mark-1 is its 32 bit major data path bit width. The designs used to derive the predictors all had major data path bit widths of 12 bits or less. If the predictions seem reasonable for a device with data paths three times the size of the other designs it would indicate that the predictors are fairly general. Other considerations such as the ratio of registers to operators in a design could make the predictions even more accurate.

The first step in applying the predictors is the tabulation of the N, B, and L values for the Mark-1.

	Bindable Nodes (N)	Major Data Bits (B)	Max Control Path Steps (L)
Change Mechanism:	16	32	16
Factors	16 * 32 =	512	16

Table 5-12: Mark-1 Normalizing Factors

The predictions are computed using the following relationships:

P_c Cost (area) prediction.
P_d Delay prediction.
P_p Power prediction.

$X_n^?$ Any of X_n^-, $X_n^=$, or X_n^+

$P_n^?$ Any of P_n^-, $P_n^=$, or P_n^+

$P_c^? = X_c^? * B * N$

$P_d^? = X_d^? * L$

$P_p^? = X_p^? * B * N$

For example, to predict the maximum expected power for a cell implementation of the Mark-1:

$X_p^+ = 219.87$ (from Table 5-11)

[7] the ISP description is included in Appendix C

B * N = 512 (from Table 5-12)

P_p^+ = 219.87 * 512 = 112573.44 microwatts

That predicted maximum power value for the Mark-1 implemented with Sandia Cells compares favorably with the measured maximum of 108600.00 microwatts.

Tables 5-13 and 5-14 summarize the resulting predictions. Numbers shown in parenthesis are the values measured from processing the Mark-1 with 64 different sets of constraints.

	Cost	Delay	Power
Minimum:	332.80 (375.12)	312.73 (348.00)	16317.44 (15952.00)
Mean:	691.20 (746.76)	426.88 (500.38)	37125.12 (37214.84)
Maximum:	1049.60 (1179.36)	541.03 (624.00)	57932.80 (60700.00)

Table 5-13: Mark-1 - TTL Predictions

	Area	Delay	Power
Minimum:	223226.88 (286938.70)	756.64 (690.00)	47498.24 (50820.00)
Mean:	391767.04 (409513.50)	1218.08 (1246.25)	80035.84 (76374.69)
Maximum:	560307.20 (606871.40)	1679.52 (1660.00)	112573.44 (108600.00)

Table 5-14: Mark-1 - Cell Predictions

Figures 5-9 and 5-10 show plots of the measured design space projections and the computed predictions. Predictions are shown as dashed lines.

The results appear to be quite reasonable. Many of the predictions are within 5% of the measured values[8]. All but the minimum cost for the cell design are within 15% of the measured values. The minimum cost for the cell design is approximately 22% from the measured minimum.

---

[8]Absolute percent difference - it has no bearing on whether the point is inside or outside the prediction box

The plots show that the Mark-1 with TTL modules tended to have clusters of design points. Since several of the points fall outside the box defined by the predictions, two things should be considered:

- The majority of points falling outside of the predictions are toward the "worse" (slower, more expensive) end of the design space.
- It is easy to make the predictions more liberal by including some factor such as the standard deviations.

The predictions for the Mark-1 look better in the plots for cell implementations than in the plots for TTL implementations. There was less tendency toward clustering with the cells.

The major result from this demonstration is that the predictors appear to give a useful set of bounds for the expected design space using items that can be measured from a description at a very early stage of the design process. In the larger CMU-DA context, it can quickly be decided whether to continue processing if the designer's constraints fall within the bounds or to refer the design back to higher levels if the target has been missed.

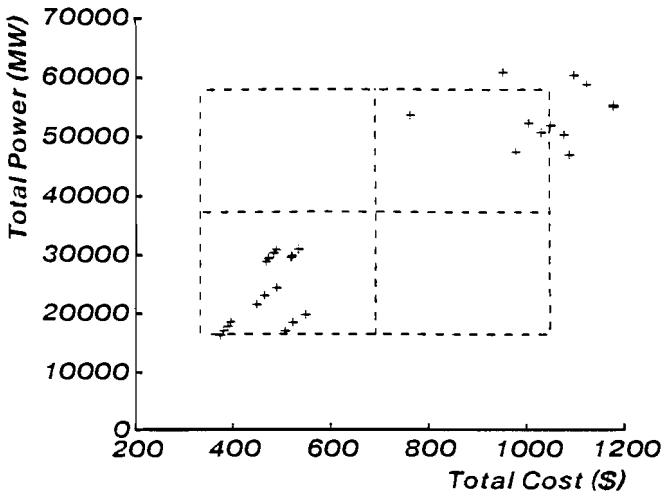
Since the design space information is available for a fourth design, it will be used to refine the predictors.

5.6.1 The Best Predictors

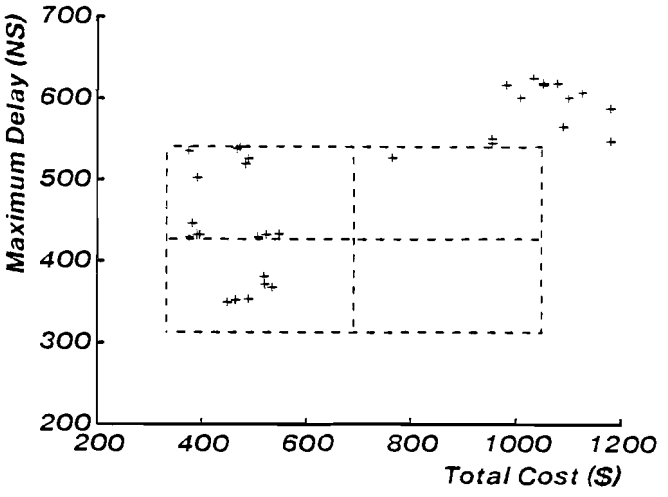
The previous section showed how the Mark-1 design space could be predicted using the values derived from three designs. The Mark-1 information can be included with that of the other designs to refine the predictors. The resulting values are given in Table 5-15 for TTL designs and in Table 5-16 for cell designs.

	X_c Cost X_c	X_d Delay X_d	X_p Power X_p
X_n^- (Minimum):	0.67	20.24	30.75
$X_n^=$ (Mean):	1.38	27.83	72.56
X_n^+ (Maximum):	2.09	35.42	114.37

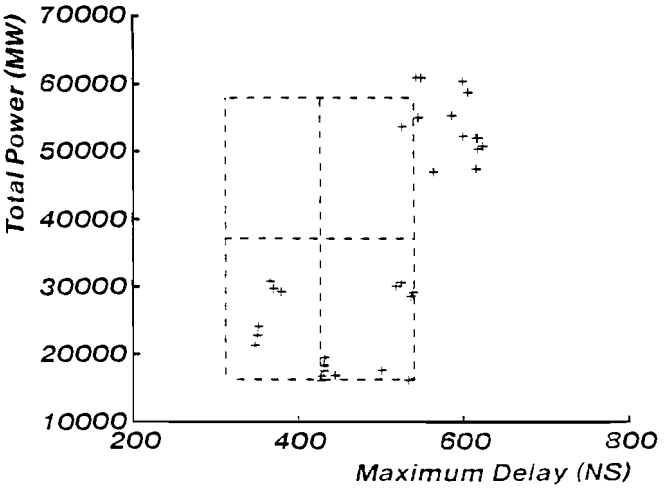
Table 5-15: TTL Designs - Final Predictors



A. Mark1/TTL - Cost vs Power

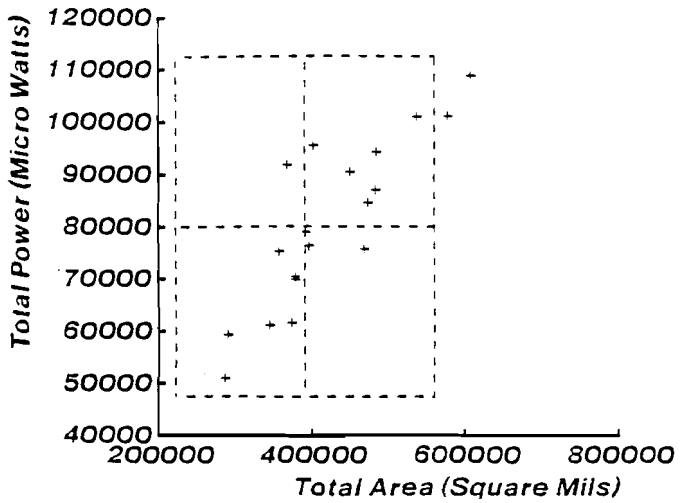


B. Mark1/TTL - Cost vs Delay

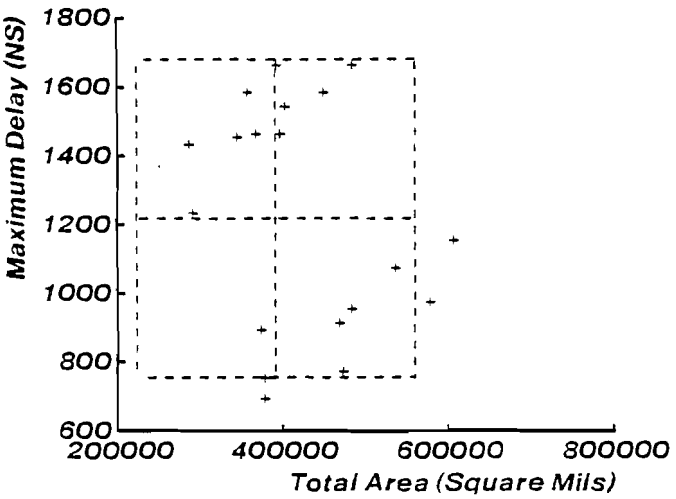


C. Mark1/TTL - Delay vs Power

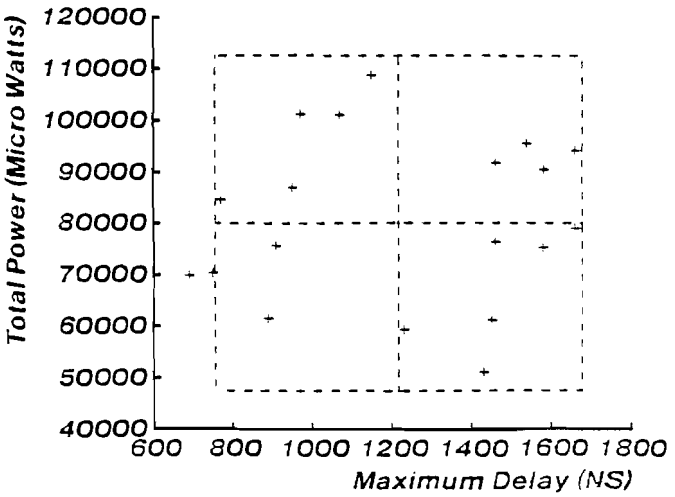
Figure 5-9: Mark-1/TTL Design Space and Predictions



A. Mark1/Cell - Cost vs Power



B. Mark1/Cell - Cost vs Delay



C. Mark1/Cell - Delay vs Power

Figure 5-10: Mark-1/Cell Design Space and Predictions

	X_c Area	X_d Delay	X_p Power
	X_c	X_d	X_p
X_n^- (Minimum):	453.86	46.83	93.09
$X_n^=$ (Mean):	773.83	76.57	154.53
X_n^+ (Maximum):	1093.80	106.31	215.97

Table 5-16: Cell Designs - Final Predictors

Comparing these values with the numbers in Tables 5-10 and 5-11 it can be seen that slight modifications occurred. This refinement process could (and should) be extended by including measurements for more designs as the information becomes available.

Within CMU-DA, it is expected that the predictors would be most useful at the Design Style Selection level. In the "master plan" it is envisioned that Design Style Selection will be tasked with deciding on the appropriate D/M allocator and the appropriate module set for implementation. The implication is that there would be several D/M allocators available and more than the two module sets used here. For example, the RCA CMOS module set might be available in addition to TTL for packaged design implementations. There might be several choices of cell module sets for LSI implementation. The various cell module sets would reflect differences in technology might allow (for example) trading speed for power.

At the D/M allocator level, the predictors could be used to assist processing decisions. In evaluating choices between potential constructs, a D/M allocator could estimate the expected impact of the choices on the final design.

At the LSMS level, the predictors should have application in applying global strategies to guide logic synthesis and module selection.

Outside the context of CMU-DA, these predictors could be applied to the very early system level block diagrams in hand designs to estimate the range of cost, delay, and power that might be expected in a detailed design.

5.7 Conclusions

The LSMS step of CMU-DA is the first point where actual hardware information is associated with a design. Until now, the size and shape of design spaces could not be estimated or measured in dollars, nanoseconds, and milliwatts. The advent of SYNNER as a tool for assigning actual hardware and directing that selection based on designer constraints has made it practical to map some transformation design spaces. Transformation design spaces using real module sets have been shown to have parabolic bounds rather than the expected hyperbolic bounds. Using the data from the design space studies, a set of predictors have been derived that make it possible to estimate the achievable bounds of the cost, delay, and power parameters *before* a design is processed at the LSMS level.

The capability to manipulate designs by varying constraints has just started to be exploited. However, from that start several important results have been realized:

- The predictors are specific to each module set and can be included as summary information with the module set databases. Unlike earlier speculation, the predictors are also a function of processed designs and are not simply summarized module set information.
- For the first time within CMU-DA, information has been extracted from one level of design that can serve as guidance for earlier levels of design. The predictors can be applied at the Design Style Selection level, the Data/Memory allocator level, and the LSMS level (to aid global synthesis strategy).
- The predictors could be used outside the context of CMU-DA as estimators to guide the implementation of hand designs.

Chapter 6
Results and Conclusions

"Man-machine identity is achieved not by attributing human attributes to the machine, but by attributing mechanical limitations to man."

-- Mortimer Taube

6.1 Results

The research reported in this thesis is a first excursion into the automation of the Logic Synthesis and Module Selection level of digital design. The excursion began by identifying and automating certain transformations (Chapter 2) that appeared to be necessary to manipulate the structure of a design. The transformations have application as computer aided design tools for use by a human designer. However, the development of a surrogate designer (Chapter 3) to exercise the judgment in applying the transformations brought the LSMS level into true design automation. Calibration of the automated system (Chapter 4) against human designers quantified the abilities of SYNNER and gave confidence that it is very close to producing designs indistinguishable from those of the population of relatively good human designers. The calibration occurred toward the "optimal" end of a design space, the end approaching the intersection of axes. Given the confidence that SYNNER was reasonably good toward the optimal end of the design space, the next step (Chapter 5) was to apply its speed and capability for varying constraints to a number of descriptions in order to "map" the design space projections for cost, delay, and power. The design space projections are interesting since they imply a parabolic shape rather than the expected hyperbolic shape. The wealth of data accumulated could only be subjected to a rather superficial analysis due to time and size constraints here. However, the information led to the development of a set of predictors which can be used with hand design methods or higher levels of a design automation system to estimate the general bounds of design space projections. Predictions

are simply end points and define a bounding box in the design space projections. It is very possible for future research to extend the concept to include parabolic or elliptic bounds on the shape of the estimated design space.

6.2 Contributions

When this work began it was not really clear that there was a particularly large problem within the module selection area. It was even less clear that there were any significant tradeoffs that would lead to sizable design space projections. Both of those early doubts have been proven to be unfounded. One of the contributions of this work has been the identification and structuring of the LSMS design level.

Earlier work in module selection ([Barbacci 73], [Rege 74]) used module sets that were defined with more consistent control and data interfaces than are found in the highly used commercial module sets. The decision to try to deal with actual module sets rather than "nicely" defined module sets was an attempt to satisfy a part of the goal to keep CMU-DA technologically relevant (which implies that it should be technology independent). Implementation of the module set database concept with the capability of module independent transformations provided a demonstration that it is possible to deal with different technologies (at least at the SSI/MSI level). Whether this particular implementation is judged as good, bad, or indifferent is of little consequence. What is important is the existence of at least one demonstration that the problem can be organized and managed with software aids. The fact of existence hopefully will provide one additional motivation for undertaking additional research and development in this area.

The existence of a software tool with the capability to process designs with different module sets and different constraints provided an opportunity to explore design spaces. For at least the past decade, the concept of trading cost for speed has been cast as a projection of a design space. Data points in design spaces have generally been provided by computer manufacturers in the form of computer families (the most notable of these are the IBM 360/370 and DEC PDP-11 families). The sheer cost of developing, marketing, and supporting computer families has left such design spaces sparcely populated. The expense of developing alternative designs has mitigated any desire to get an actual look at design space projections. Even the small sample of human designs generated for the experiment of Chapter 4 took between two and four weeks to complete (obviously on a low priority/part time basis). SYNNER generated cost, delay, and power points for 64 sets of constraints on three

designs using two module sets (384 points) within an eight hour (wall clock) period[9]. This capability made it possible to actually map design space projections. These projections represent SYNNER's structural manipulation capabilities and do not as yet imply too much about the absolute bounds of the design space. However, with ranges exceeding 300% for cost and power with TTL, these design spaces provide a statement about the shape of a useful set of tradeoffs. The surprising parabolic bound to these design spaces appears in hindsight to be more reasonable (for real module sets) than the expected hyperbolic bound. This discovery is significant since it changes the way in which design spaces at this level will be envisioned. However, better quantification of bounds and attempts to predict the absolute limits of such bounds remain for future work.

The approach to implementation was unique within the CMU-DA context. The software system was designed to be configurable from a variety of external means including module databases, equivalence axioms, command files, and direct designer input. The resulting flexibility provides a cost effective means of exploring this level of design by making it possible to perturb variables (module sets, axioms, etc.) and observe the resulting impact on a design. This capability brings an air of engineering expermentalism that offers a useful balance to the more formal investigations of CMU-DA.

6.3 Future Research

This research can be viewed as a rather long but narrow penetration of the LSMS design level. It has traveled a long distance but has passed many tempting opportunities to explore interesting issues that became evident along the way. A few of those issues will be outlined here in the hope that they will provide further ideas to people that are interested in continuing these explorations.

SYNNER was developed to deal with the only D/M allocator that is currently available in CMU-DA. This allocator is for the distributed design style. SSI/MSI level modules are appropriate for distributed designs, but they are not necessarily appropriate for other design styles. No work has yet been done on LSMS issues for LSI modules. In many ways it would appear that LSMS would be simplified (or at least the opportunities for trading off constraints would be reduced) if the modules were more capable. However, the difference in range of parameters for TTL and Sandia Cells indicates that this may not be true. While SYNNER's techniques might be usefully applied to the design of LSI modules, the actual application of LSI modules to large designs may require development of a completely different approach.

---

[9] Without ever leaving for coffee

Within the confines of the distributed design style and SSI/MSI modules, there are a large number of issues that beg for investigation. SYNNER takes an extremely localized approach to synthesis and selection. The nodes of a design are viewed one at a time as if they were on a strip of movie film that is run past a small window restricting the view to a single frame Constraints are applied to the modules or aggregate of modules used to implement single nodes. A more global view the design would lead to more optimal results. By taking a wider view of the locality around a specific node, opportunities for reductions could be identified. For example, register nodes may have merged operators that require a counter to be selected. If an adder existed in the locality of the register, it might be beneficial to multiplex its inputs to perform an increment rather than assign a counter. Yet a wider view might be able to analyze the interactions of such localities and develop strategies for optimizing the total design.

If more global strategies are developed it may become possible make designer constraints reflect total design goals. As it stands, SYNNER can be told (for example) to minimize the cost or it can be given a specific cost as an absolute constraint. Either form deals with a list of candidate modules for implementing a single node. It is desirable to be able to set goals for the entire design (a 500ns, $300, TTL PDP-8). The localized approach in SYNNER does not allow specification of goals because there is no strategy for dealing with them.

SYNNER explores various structures to implement specified behaviors. Other approaches are possible and might be worth investigating. It would be possible to hold the structure constant and synthesize "super modules" from existing devises. The super modules would implement the behavior required by a specific node, but it could also be stored in the module database for future use. A mixed approach might be employed to iterate between structural modification and super module synthesis. It would be a difficult task to determine if the constructs were converging, but that might be an interesting application of some artificial intelligence techniques such as Means-Ends analysis [Ernst 69].

The first implementation of CMU-DA was structured as a feed-forward model. Designs simply progress from a predecessor step of the system to a successor step. From the outset it has been acknowledged that some form of feedback between the partitions of the system would be necessary to produce optimal designs. Identifying and dealing with the issues localized to each partition of CMU-DA have received the major attention to this point. It would now be both appropriate and interesting to investigate just what kind of information could be returned from the LSMS level to a higher level (D/M allocator, Design Style Selector, or Global Optimization) in order to arrive at a better overall result.

Appendices

Appendix A
The Module Database System

The storage and retrieval of module and module set information was one of the earliest concerns of this research [Leive 77]. An investigation of database systems available in the CMU environment led to the inescapable conclusion that development of a limited database facility specifically tailored to the needs of LSMS would be worthwhile. The early attempts to expeditiously develop a minimum facility resulted in the decision to devise a hierarchy of ASCII files that could be manipulated with the existing text editors. The access mechanism and the data record formats were embedded in an early version of SYNNER. However, as other steps of CMU-DA (particularly Control Allocation) were implemented, it became apparent that a more flexible and generalized database mechanism would be needed to serve an expanded community of users. The first step in generalizing the database system was the development of an editor (DBEDIT [Leive 79, Weiss 79]) for the database. The editor was designed to process a very general form of a record oriented database. An access mechanism (MDBLIB [Leive 79]) was also developed to remove most of the chores of information retrieval from individual programs. The editor and the access mechanism use the same externally defined data record format. Therefore, it is quite easy to extend the information content of any of the databases as new requirements evolve.

A.1 Module Database

The Module Information DataBase access structure of [Leive 77] is shown in Figure A-1. The hierarchical structuring of the information follows the top-down philosophy of the design synthesis task by providing either summary or detailed information to the design programs. The design style selector would only have access to certain types of summary information while the allocators followed by the LSMS system and Control Allocator could access progressively more detailed design information.

The implemented database follows the structure of Figure A-1 closely although some details differ. The module database (MDB) is really distributed across several ASCII files that allow

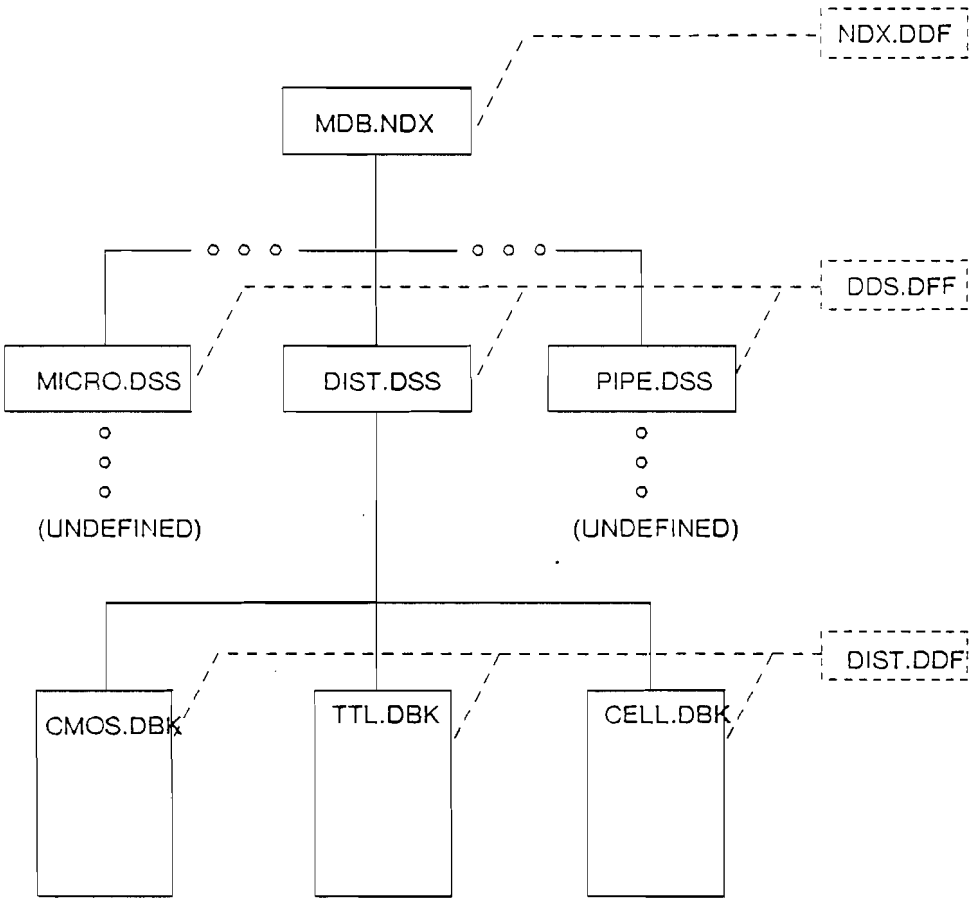


Figure A-1: Module Database Organization and Access

ease of access, maintenance, and transportability. The files are linked together by filename pointers as indicated in Figure A-1. The *.DBK files are the "databooks" which contain specific module information. The *.NDX and *.DSS files provide the access paths to the detailed information. They also will contain the summary information extracted from each more detailed level of the database. The *.DDF files are the data definition files which specify the format of the data and access files.

The database is organized in a hierarchy of **accesslevels**. The most inclusive accesslevel is an index containing pointers to an intermediate index. The intermediate index contains pointers to the lowest accesslevel, the module databases. Individual modules within the module databases may contain lists of information that may be of any length. The lists within specific modules are said to be **nested**. Nested lists may also contain other nested list to any depth. Such nested lists are referred to as **nestinglevels**.

The root node of the MDB is a file named MDB.NDX, the iNDeX file. MDB.NDX defines access to the Design Style Sets. The form of MDB.NDX is:

```
NDX.DDF[N750DA99]
 0 DIST       DIST.DSS   N750DA99
 0 MICRO      NONE.DSS   N750DA99
```

The reference to NDX.DDF is a pointer to a file containing the data format definition. The leading '0' on a line is a reference line number. Both of these items will be explained in more detail later in this discussion. The first character string identifies the Design Style by name. The second field defines the **<filename.extension>** of the access path to the Design Style Set Index. The third field is the account number where the file defined by the second field resides. Note that only the Distributed Design Style Set is defined (the file NONE.DSS is empty).

The Distributed Design Style Set index (DIST.DSS) points to the module sets that are included in that design style set:

```
DSS.DDF[N750DA99]
 0 TTL        TTL.DBK     N750DA99    3.00        DOLLARS:PLUS
 1 TTL                    MW          NS

 0 CELL       SANDIA.DBK  N750DA99    2.00        SQMILS:MUL
 1 CELL                   UW          NS

 0 CMOS       CMOS.DBK    N750DA99    3.00        DOLLARS:PLUS
 1 CMOS                   UW          NS
```

The format is similar to that of the NDX file, however the individual module sets are named.

Four pieces of module set summary information appear after the accessing information. The number (3.00 for TTL and CMOS and 2.00 for the Sandia Cells) is the overhead factor. The attribute:value pair following the overhead factor provides both the units and the method of applying the overhead. The units are dollars [Blakeslee 75] for TTL and CMOS and square mils for Sandia Cells. In the case of the Sandia Cells, the number represents the silicon area overhead 'cost' for routing of each cell. "PLUS" indicates an additive overhead model is used and "MUL" indicates a multiplicative overhead model is used. The parameters on line "1" are the units for the power and delay data book entries. The summary information will be expanded to summarize databook information for use by the allocators and the style selector.

The end result of these access paths is a data book (*.DBK) file that contains the actual parameters specifying the member modules. The following example from the TTL.DBK is presented to indicate the form of module information:

```
 0 SN7404     NOT          72.12        TRUE
 1 SN7404     VO           1
    0 OPN.1       LOGIC:NOT

 2 SN7404     1            1            0

 3 SN7404     1            1
 4 SN7404     6            0.18         1            20
 5 SN7404     NOM:10.00    NOM:10.00
 6 SN7404     TI           1
14 SN7404     0            0
           1
    0 NOT         0             0             1             20            0
```

There is an identifying line number and a module identifier (SN7404) attached to each line. The individual fields on each line are a mixture of string, integer, real and boolean operand types. Each element of MDB discussed in previous sections has a different format, but the LSMS system and the editor have been designed to operate on each of them without specific knowledge of the format. This is accomplished by specifying a Data DeFinition (*.DDF) file that utilizes a simple syntax to specify the field name, the line number, the field number, and the data type. The DDF file for DIST.DSS is shown:

```
FILENAME    0  0  S  ! FILENAME.EXT
PPN         0  1  S  ! FILE RESIDENT PPN
OVERHEAD    0  2  R  ! MOD MOUNT OVERHEAD
```

The first parameter is the FIELD name which is a string of up to 10 characters. It is used by the information access routines in both the editor and the the database access routines to locate a specific field. The second parameter is the line number on which the data will reside

in the database. The third parameter is the positional number of the field in the specified line. The final parameter specifies the data type and must be a member of the set: B, I, S, R, AVB, AVI, AVS, AVR, P, O. These stand for: Boolean, Integer, String, Real, Attribute-Value:Boolean, Attribute-Value:Integer, Attribute-Value:String, Attribute-Value:Real, Pointer, and Octal Integer. The comment at the end is provided as an explanation of the field and is required when building DDF files. The comment is used by the editor as an extended help facility during field insertion.

The DDF files correspond quite closely in both spirit and implementation to the *schemas* of larger database systems [Wiederhold 77]. True schemas generally contain more information such as valid ranges for each data type. Schemas may be tailored such that different users have different 'views' of the same data. All of the attributes of schemas are interesting and useful, but they exceed module database requirements.

A.2 DataBook Data Definition

The data definition (DDF) for the databook portion of the database is discussed in this section. It defines the position, field name, and data type of each entry in a databook. The fields that are defined contain enough information for data part module selection and controller synthesis.

Seven data types are supported by the Module DataBase:

1. Data type **string** allows strings that may be up to 10 characters long. They must start with an alphabetic character. All other characters may consist of the set: '.', 'A':'Z', 'a':'z', '0':'9'. Lower case alphabetic characters are set to upper case. No embedded spaces are allowed. All characters not in the specified set act as string delimiters. The DDF symbol for character string is 'S'.

2. Data type **integer** consists of the set of all legal (PASCAL) [Jensen 74] integers. Leading '+' or '-' signs are recognized, but only the minus sign is restored on output. Integers are represented in base 10. The DDF symbol for integer is 'I'.

3. Data type **octal** allows octal integers. The DDF symbol for octal is 'O'.

4. Data type **real** allows all legal (PASCAL) real numbers. The DDF symbol for **real** is 'R'.

5. Data type **boolean** is represented by the strings: 'TRUE' or 'FALSE'. The DDF symbol for boolean is 'B'.

6. **Attribute:Value** pairs consist of a string identifier (the attribute), a separator ':', and a value. The value may be any of the basic data types: string, integer, real or boolean. The DDF symbols for attribute:value pairs are:

   - AVS - String value
   - AVI - Integer value
   - AVR - Real value
   - AVB - Boolean value

7. Data type **pointer** indicates that the associated field will point to nested modules. This means that module definitions in the database can be trees containing other modules with different basic structures. The motivation for this data type can be seen by considering the problem of specifying the physical pin on a module. Even in the TTL module set there are different numbers of pins on various modules. It would be possible to allow spaces for up to 24 pins in each module definition. However, this would not be general and it would make it difficult to access the pin information. A better way is to make a PIN field that is a **pointer** to pin 'modules'. The list can be of variable length and will contain field definition (again, from the DDF) that are tailored to a description of the pin information. The DDF symbol for pointer is 'P'.

The complete data definition for a module databook is included below.

```
FNAME       0   0   S           ! Common function name
DATE        0   1   R           ! Date of introduction: YY.MM
AUTO        0   2   B           ! Allow or Disallow Autobinding
TYPE        1   0   S           ! Type: VO, VC, PO, PC
OPN         1   1   P           ! Operations performed by module
( OPN           0   0   AVS     ! CLASS:OPN (LOGIC:AND, etc.)
)
DIBWD       2   0   I           ! Data Path (parallel) input bit width
DINS        2   1   I           ! Number of Data Inputs
DIFLAG      2   2   P           ! Data Input Flag List
( NAME          0   0   S       ! Input Pin Name
  FLAG          0   1   P       ! Pointer to Flag List
  ( FLAG            0   0   S   ! Link Flag that can be matched
  )
)
DOBWD       3   0   I           ! Data Path (parallel) output bit width
DOUTS       3   1   I           ! Number of Data outputs
REPS        4   0   I           ! Replications
COST        4   1   R           ! Cost
LOAD        4   2   I           ! Input Load Units
DRIVE       4   3   I           ! Output drive (Load Units)
POWER       5   0   AVR         ! Power used (MW:10.0), W, MW, UW, NW
DELAY       5   1   AVR         ! Delay: MAX:9.5
MFGR        6   0   S           ! Manufacturer Name Abbreviation
COMPLX      6   1   I           ! Equivalent gates
ATTR        7   0   P           ! Attributes: LSHIFT, INCR, etc.
( ATTR          0   0   S       ! LSHIFT, RSHIFT, INCR, DECR, CLEAR
)
DATAIN      8   0   P           ! Data Input Pins
( PN            0   0   S       ! Pin Name
  INPUT         0   1   I       ! Input Number
  VP            0   2   I       ! Data Vector Position
  TSETUP        1   0   I       ! Setup time (Nsec)
  THOLD         1   1   I       ! Hold time (Nsec)
)
DATAOUT     8   1   P           ! Data Output Pins
( PN            0   0   S       ! Pin Name
  OUTPUT        0   1   I       ! Output Number
  VP            0   2   I       ! Data Vector Position
)
PWR         9   0   P           ! Power Voltage Levels
( PN            0   0   S       ! Pin Name
  VOLTAGE       0   1   I       ! Voltage
  CURRENT       0   2   R       ! MA
)
PIN        10   0   P           ! Package Pin Definitions
( PN            0   0   S       ! Pin Name
  REP           0   1   I       ! Replication Pin Number
)
CTLLINES   14   0   I           ! Number Of Control Lines For This Device
CTLNAME    14   1   P           ! Control Line Names (Ordered)
( PINNAME       0   0   S       ! Pin Name
```

```
  PINTYPE      0  1  I          ! Pin Type 0=Select, 1=Evoke
  NONEVOKE     0  2  S          ! Value For This Pin To Nonevoke (H L X)
  SUBMODNO     0  3  I          ! Submodule Class Number For This Line
)
CTLESEQ   14   2  P             ! Control Evoke Sequences
( EVOKLINE     0  0  I          ! Ctlname Number Of Active Evoke Line
  EVOKSTEP     0  1  I          ! Eseq Number Of Actual Evoke Step
  SUBMOD       0  2  I          ! Submodule Required For This Operation
  MAXTIME      0  3  I          ! Time to perform this operation
  ESEQ         0  4  P          ! Evoke Sequence For This Operation
  ( EVAL          0  0  P       ! Evoke Values For The Control Pins
    ( BITVAL         0  0  S    ! Values For The Pins (P N H L S X)
    )
  )
)
```

A.3 Database Editor

The database editor (DBEDIT) was developed to make it convenient to maintain module set information. The editor is able to operate on any of the accesslevels.

When the editor is invoked, it reads a specified database file and creates a backup copy of that file. It then enters the command mode. At that point, the following commands/capabilities are available:

```
COMMANDS          EXPLANATION
--------          -----------
P<Range>  .  .  . PRINT: Modules, a Module, a Line or a Field
L<Range>  .  .  . LIST: print to a file rather than the screen
I<Range>  .  .  . INSERT: a Module, a Line or a Field
D<Range>  .  .  . DELETE: a Module, a Line or (default) a Field
R<Range>  .  .  . REPLACE: a Module, a Line or a Field
O<Range>  .  .  . WINDOW: Print IDs of: Pred, Current, Succ
B<Range>  .  .  . BACK: Move to Module/Line/Field
  <       .  .  . PRED: Move to pred Module/Line/Field (Prints)
  >       .  .  . SUCC: Move to succ Module/Line/Field (Prints)
  W       .  .  . SAVE: Save the edit, then continue editing
  ↑       .  .  . POP: Return to next highest edit level
  E       .  .  . EXIT: after saving the edit
  Q       .  .  . QUIT: without saving the edit
  N       .  .  . NEXT: Save the edit then edit another file
  G       .  .  . GO: quit then edit the next file
  H       .  .  . HELP: Print this list
H<Topic>  .  .  . Prints help on <Topic>
                  Type H TOPICS for Topic list
```

The <RANGE> specification directs the editor to operate on modules, lines, or fields.

A.4 Database Access

A BLISS-10 compatible database access package called MDBLIB has been created to make it easy to retreive module information. The access package is designed to completely insulate the user from any future changes in the actual storage and retrieval mechanisms used to operate the database.

Initialization is performed to set up the accesslevels of the database. The initialization entry point is called three times: the first call initializes the database index; the second call initializes a design style set; the third call initializes the databook. After initialization, accesses can be performed on any level of the database.

The access mechanism consists of a query/response format: the query for information is stated by parameters passed to the called entry point. The response is information returned to a predefined location called the **returnblock**. The returnblock contains a copy of the requested information and several pieces of status information. The status information includes a **returncode** which allows the user to determine if the query was successful or exactly why the query failed if it was unsuccessful.

The display entry point is provided primarily for use with interactive systems. Database modules may be displayed in a format similar to that provided by DBEDIT.

The global entry point MDBLOAD must be called three times to fully initialize the database. The first call to MDBLOAD must be to initialize the database index. The supported database index file is called:

```
rtnblk _MDBLOAD('NDX',PLIT ASCIZ 'MDB.NDX');
```

The variable 'rtnblk' is defined by the user and receives the address of a returnblock initialized by the 'NDX' call to MDBLOAD. The first parameter to MDBLOAD is a short string telling it that the index file is to be loaded. The second parameter is a pointer to a long string giving the file specification.

The second call to MDBLOAD initializes the 'DSS' index. This call differs from the 'NDX' call in two of ways; the call does not return a pointer; MDBLOAD returns TRUE (1) if the initialization was successful or FALSE (0) if the initialization failed. The first parameter is similar to the first parameter of the index initialization call. The second parameter, however, is not a file name, it is the name of a design style set. The name must correspond to a design style set defined in the NDX file. The NDX file contains the file access information. An example of design style set index initialization is:

```
IF NOT MDBLOAD('DSS',PLIT ASCIZ 'DIST') THEN RETURN;
```

The second parameter is a pointer to a long string that is the valid identifier for the distributed design style set.

The final call to MDBLOAD opens the lowest accesslevel of the database, the databook. Since the format and returns are similar to those for the DSS initialization, the example will be presented with further discussion:

```
IF NOT MDBLOAD('DBK',PLIT ASCIZ 'TTL') THEN RETURN;
```

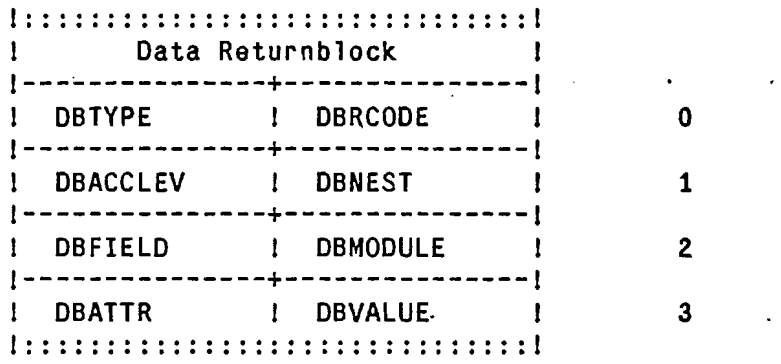
Once the database is initialized, any level may be accessed. The single entry point MDBACCESS is provided to retreive several kinds of information either about the state of the database processing or about the modules stored in the database. To do all of that with a single entry point requires that several parameters be passed with each call. The number of parameters will vary depending of the activity requested. Version 1A of MDBLIB supports twelve different activities that fall into the rough categories of status query, positioning commands, and access commands. The general format for a query is:

```
MDBACCESS(<parameters>,<command>,<accesslevel>,<returnblock>)

Where:
    <parameters> ::= 0 to 2 parameters as required by the
                     requested activity.
    <command>    ::= <shortstring>
    <shortstring>::= 'CHECK' | 'VERS'  | 'HEAD'  | 'TAIL'  |
                     'SUCC'  | 'PRED'  | 'FIND'  | 'ENTER' |
                     'EXIT'  | 'LINK'  | 'TYPE'  | 'VALUE'
    <accesslevel>::= 'NDX'   | 'DSS'   | 'DBK'
    <returnblock>::= pointer to a returnblock
                       (if initialized);
                     pointer with value zero
                       (if the pointer is to
                        be initialized)
```

MDBACCESS directly returns the same value of the returncode that is placed in the returnblock. Only the 'success' returncode evaluates to TRUE. All error codes evaluate to FALSE. This feature makes it easy to determine if an access succeeded or failed without having to first read the returnblock.

Before undertaking a detailed discussion of the access calls it is necessary to define the general form of the returnblock and specify the codes and types of values that are returned.

The returnblock is large enough to contain all of the information that may be returned by the database. However, the user only needs to be concerned with the first four words. The picture of the returnblock is:



The DBTYPE field will return a number identifying the datatype for any access or inquiry that returns field information. The DBRCODE field will contain a returncode specifying the success or failure of any attempted access. This is the same value as is returned directly by the MDBACCESS routine. Only the 'success' return evaluates as TRUE. The returncodes are:

```
RCAF    = Access Failure
RCSUCC  = SUCCESS
RCUV    = Uninitialized Value
RCNSF   = No Such Field
RCNSM   = No Such Module
RCNDBK  = No DBK Loaded
RCNDSS  = No DSS Loaded
RCNNDX  = No NDX Loaded
RCNSL   = No Such Level
RCNSC   = No Such Command
RCNP    = Not a Pointer
```

The DBACCLEV field will contain an integer identifying the index or databook level that is being queried. The values are:

```
Index level ('NDX')              ::= 1
Design Style Set level ('DSS')   ::= 2
DataBook level ('DBK')           ::= 3
```

The DBNEST field will contain an integer identifying the nestinglevel that is being processed. The outer nestinglevel has the value 0. This level corresponds to the individual SN74XX modules in the TTL.DBK. Pointer fields within individual modules allow nested lists of modules. If processing is being done within such a list, the value returned in DBNEST will indicate the depth of the list.

The DBFIELD, DBMODULE, and DBATTR fields are all pointers to strings. DBFIELD points to the fieldname, DBMODULE points to the modulename, and DBATTR points to the attribute (always a string) of an attribute-value pair.

The DBVALUE field is an pointer to a value returned by an access.

The returncode (DBRCODE), accesslevel (DBACCLEV), and the nestinglevel (DBNEST) are filled for each access to MDBACCESS regardless of the type of activity requested. The remaining fields are selectively filled or returned containing (or pointing to) no value (zeros) depending on the type of access.

Two commands are provided to retreive information about the state of the database or the database access package. The CHECK command provides a means to test if an accesslevel is initialized. The VERS command returns the version number of the MDBLIB database access package.

By far the largest number of commands (eight) are devoted to positioning to appropriate levels or modules prior to accessing information. Four of the positioning commands (HEAD, TAIL, SUCCessor, and PREDecessor) allow relative motion within the list of modules at a particular nestinglevel. These commands to not require a module identifier. The FIND command does require a module identifier and is used to search for a particular module. ENTER and EXIT are used to traverse nestinglevels. The LINK command is used to traverse accesslevels.

HEAD returns the modulename of the first module in the list. TAIL returns the modulename of the last module in the list. The DBMODULE field of the returnblock points to the modulename.

The general form of the call is:

```
MDBACCESS('SUCC','DBK',rtnblk);
MDBACCESS('PRED','DBK',rtnblk);
```

If the access is successful, the modulename of the successor or predecessor to the current module is returned (pointed to by the DBMODULE field of the returnblock). The most likely failure mode is RCNSM (no such module) which could occur if the predecessor of the HEAD module or the successor of the TAIL module is requested.

The FIND command is used to position the database access mechanism to a specific module.

The ENTER and EXIT commands are used to traverse nestinglevels within a module. ENTER is used to change the scope of the accessing mechanism to make a nested list available for further processing. EXIT is used to return to the next outer nestinglevel. The EXIT command will change the scope back to the calling nestinglevel and will cause the DBNEST field to be decremented. at nestinglevel 0, the access will fail and the returncode will indicate no such level (RCNSL). If EXIT succeeds, the modulename will be the same as the module from which the ENTER command was executed.

The LINK command is used to traverse the access hierarchy. As the database is initialized, a module from the NDX points to a DSS index. A module from the DSS points to a specific DBK. The LINK command applied to an accesslevel will return the modulename of the module that points to the next lower accesslevel.

Two commands are provided to get information from the database. The TYPE command is used to determine the datatype of a specific field. The VALUE command is used to return both the value and the datatype of a field. In general, VALUE will return either the value of the requested field or the attribute and value for attribute-value fields. The only exception is the VALUE of a pointer field which will return a decimal integer equal to the number of modules nested under the pointer.

Appendix B
Synthesis Equivalence Language

The Synthesis Equivalence Language (SEL) provides a means to specify equivalence transformations that can be used to replace nodes in a design. The language is a relatively simple tree structured language, but a powerful set of specification features allow designers the flexibility to describe most interesting transformations. These features include:

- The ability to specify an indeterminate number of sources.
- The ability to an output bits from one node as inputs to another node.
- The ability to extend the syntax by supplying qualifiers.
- The ability to specify constants as sources.

This appendix provides a complete BNF of the SEL, and a sample of the standard equivalence transformations used for synthesis of the designs described in the thesis.

B.1 Syntax

```
(*          ::= begin-optional
*)          ::= end-optional
%           ::= begin-comment
%           ::= end-comment

equivalence ::= class:classtype  *lineid   typeqv |
                equivalence
                *lineid+1 typeqv

class       ::= NODE       | SYMBOL     | OPER       |
                LOGIC      | RELAT      | ARITH

classtype   ::= nodetype   | symtype     | opertype   |
                relattype  | arithtype
```

```
nodetype    ::= REG          | TREG         | MEMORY       |
                OPER         | MUX          | DEMUX        |
                VC  %variable carrier%      |
                VO  %variable operator%     |
                PC  %path carrier%          |
                PO  %path operator%

symtype     ::= REG          | TREG         | FLAG         |
                TFLAG        | MEMORY

opertype    ::= LOGIC        | RELAT        | SHIFT        |
                ARITH        | MLTFNC       | COMPOS

logictype   ::= NOT          | AND          | OR           |
                NAND         | NOR          | XOR          |
                EQV

relattype   ::= TEST         | EQL          | NEQ          |
                LSS          | LEQ          | GEQ          |
                GTR          | TEST2C       | EQL2C        |
                NEQ2C        | LSS2C        | LEQ2C        |
                GEQ2C        | GTR2C        | TEST1C       |
                EQL1C        | NEQ1C        | LSS1C        |
                LEQ1C        | GEQ1C        | GTR1C        |
                TESTSM       | EQLSM        | NEQSM        |
                LSSSM        | LEQSM        | GEQSM        |
                GTRSM

arithtype   ::= INCR         | DECR         | NEG          |
                SUBTWO       | NEG2C        | NEG1C        |
                NEGSM        | ADD2C        | ADD1C        |
                ADDSM        | SUB2C        | SUB1C        |
                SUBSM        | MULT2C       | MULT1C       |
                MULTSM       | DIV2C        | DIV1C        |
                DIVSM        | MOD2C        | MOD1C        |
                MULT         | DIV          | MOD
```

```
lineid       ::= number<bitmap>{qual}
number       ::= digit   | number(*digit*)
digit        ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

{qual}        ::= nil              |
                  {CI} %Carry-In%  |
                  {CO} %Carry-Out% |
                  {string}
typeqv       ::= eqv | eqv,eqv
eqv          ::= identity   | structeqv
identity     ::= nodetype   | symtype    | opertype

<bitmap>     ::= nil                              |
                 <absolute>                       |
                 <absolute,absolute>              |
                 <absolute,relative>              |
                 <absolute,relative+absolute>     |
                 <relative+absolute,absolute>     |
                 <relative+absolute,relative>     |
                 <relative+absolute,relative+absolute>

absolute     ::= number
relative     ::= alpha
alpha        ::= A | B | C | D | E | F | G | H | I | J | K | L | M |
                 N | O | P | Q | R | S | T | U | V | W | X | Y | Z

structeqv    ::= root<bitmap>{qual} src<bitmap>{qual} |
                 structeqv src<bitmap>{qual}

root         ::= logictype | relattype | arithtype

src          ::= baresrc | nestedsrc

baresrc      ::= $number<bitmap> {qual}   % Relative Source %   |
                 *lineid<bitmap> {qual}   % Line Reference %    |
                 #number<bitmap> {qual}   % Constant %          |
                [$number<bitmap>]{qual}   % Replicated Source % |
                [*lineid<bitmap>]{qual}   % Replicated Reference %

nestedsrc   ::= (structeqv)<bitmap>{qual}
```

B.2 Examples

```
NODE:REG        *1    VC,REG
NODE:LINK       *1    PC,LINK
NODE:MUX        *1    PO,MUX
NODE:OPER       *1    VO,OPER

LOGIC:OR        *1    OR   $1 (OR  [$2])
LOGIC:XOR       *1    OR  (AND $1 (NOT $2)) (AND (NOT $1) $2)

RELAT:EQL       *1    XOR $1 $2
                *2    NOR [*1<0>]
RELAT:NEQ       *1    NOT (EQL $1 $2)
RELAT:GTR       *1    SUB $1 $2
                *2    AND *1<0> (OR [*1<1>])
RELAT:GEQ       *1<0> SUB $1 $2
RELAT:GEQ       *1    OR  (EQL $1 $2) (GTR $1 $2)
RELAT:LSS       *1    NOT (SUB $1 $2)<0>
RELAT:LEQ       *1    OR  (EQL $1 $2) (LSS $1 $2)
RELAT:EQL2C     *1    EQL $1 $2
RELAT:LSS2C     *1<0> SUB $1 $2
RELAT:GEQ2C     *1    NOT (LSS2C $1 $2)
RELAT:LEQ2C     *1    SUB $1 $2
                *2    OR  *1<0> (NOR [*1<1>])

RELAT:GTR2C     *1    XOR $1<0> $2<0>
                *2    AND *1 $2<0>
                *3    AND (NOT *1) (GTR $1<1:N> $2<1:N>)
                *4    OR  *2 *3

RELAT:GEQ2C     *1    OR  (GTR2C $1 $2)<0> (EQL2C $1 $2)<0>
RELAT:LEQ2C     *1    NOT (GTR2C $1 $2)
RELAT:NEQ2C     *1    NOT (EQL2C $1 $2)

ARITH:INCR      *1    ADD $1 #1
ARITH:DECR      *1    SUB $1 #1
ARITH:ADD2C     *1    ADD $1 $2 #0<0>{CI}
ARITH:SUB2C     *1    ADD $1 (NOT $2) #1<0>{CI}
```

Appendix C
ISP Descriptions

C.1 Change Mechanism ISP

```
Change.mechanism :=
    Begin

** Declarations **

Quarter.bit<>{system.input},      ! quarter deposited
Dime.bit<>{system.input},         ! dime deposited
Nickel.bit<>{system.input},       ! nickel deposited
cost<4:0>{system.input},          ! cost of item selected
nickel.out.pin<>,                 ! Activate nickel return
dime.out.pin<>,                   ! Activate dime return
quarter.out.pin<>,                ! Activate quarter return
junk.out<>,                       ! give person selected junk
sum<4:0>,                         ! Amount deposited.
                                  ! Place value (40,20,10,5)
noquarter<>,                      ! out of quarters
nodime<>,                         ! out of dimes
nonickel<>,                       ! out of nickels
correct.change.only<>,
```

```
** Main.Process **{US}

!     When coins are inserted, record their entry

    start{main} :=
        Begin
        Correct.change.only = (noquarter OR nonickel OR nodime) Next
        cost = 0 Next
!       WAIT (cost NEQ 0) Next  ! Button specifies cost
        IF Quarter.Bit => (Sum = Sum + 5; Quarter.Bit = 0) Next
        IF Dime.Bit    => (Sum = Sum + 2; Dime.Bit    = 0) Next
        IF Nickel.bit  => (Sum = Sum + 1; Nickel.Bit  = 0) Next
        IF sum GEQ cost =>
            Begin
            junk.out=1 Next      ! signal for junk dispenser
            junk.out=0 Next
            IF ((Sum = Sum - Cost) GTR 0) => Coins.out.pin()
            End Next
        Restart start
        End,
```

```
Coins.out.pin :=
    Begin
    nickel.out.pin=dime.out.pin=quarter.out.pin=0 Next
    Run.Coins.out :=
       Begin
       IF not (noquarter AND nodime AND nonickel) =>
           Begin
           IF sum GEQ 5 =>
               Begin
               IF not noquarter =>
                   Begin
                   quarter.out.pin = 1 Next
                   quarter.out.pin = 0
                   End Next
               IF (sum = sum - 5) GEQ 5 => Restart Run.Coins.out
               End Next
           IF sum GEQ 2 =>
               Begin
               IF not nodime =>
                   Begin
                   dime.out.pin = 1 Next
                   dime.out.pin = 0
                   End Next
               IF (sum = sum - 2) GEQ 2 => Restart Run.Coins.Out
               End Next
           IF sum GEQ 1 =>
               Begin
               IF not nonickel =>
                   Begin
                   nickel.out.pin = 1 Next
                   nickel.out.pin = 0
                   End Next
               IF (sum = sum - 1) GEQ 1 => Restart Run.Coins.out
               End
           End
       End
    End,

End
```

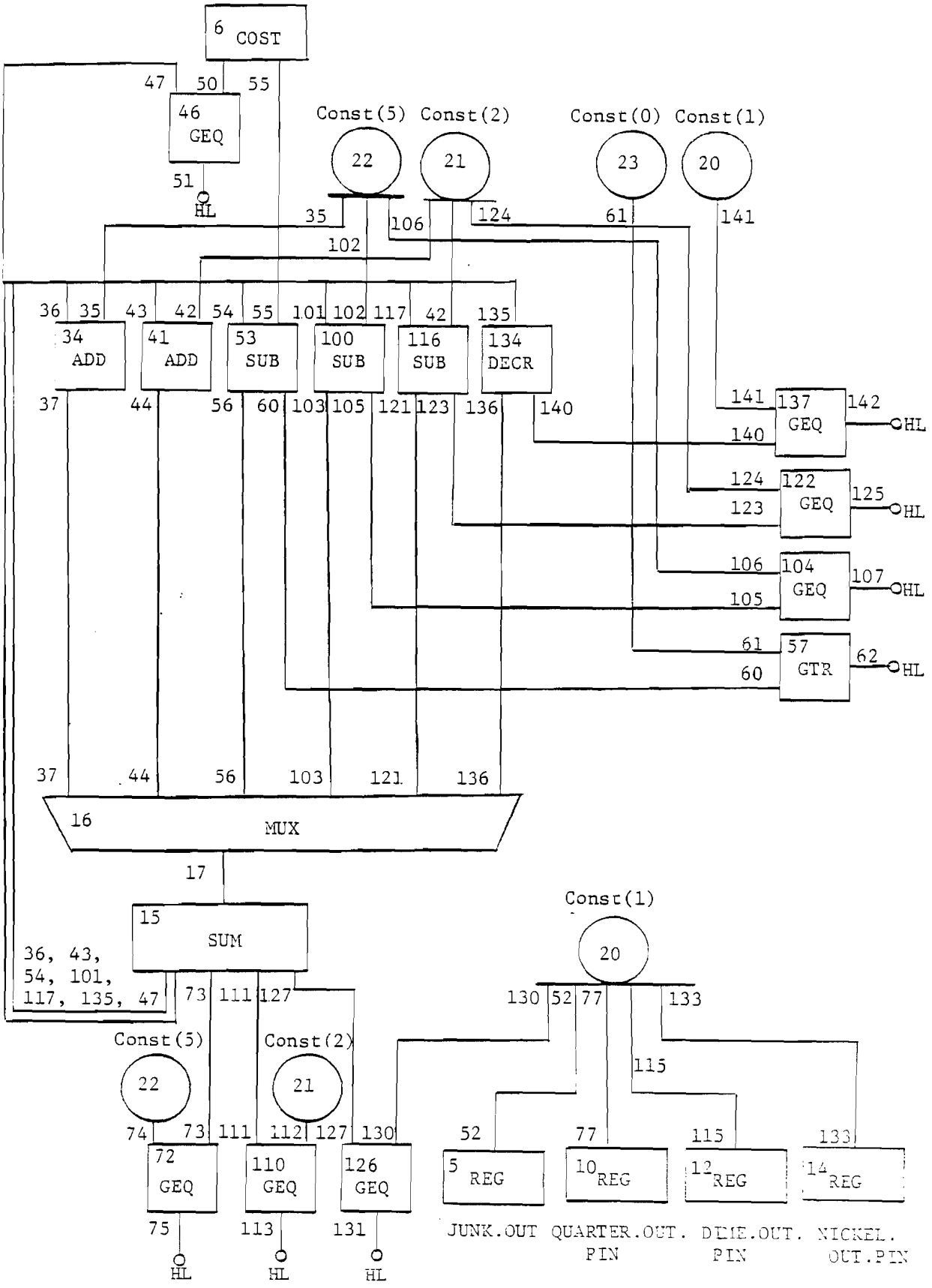


Figure C-1: Change Mechanism Path Graph (1/2)

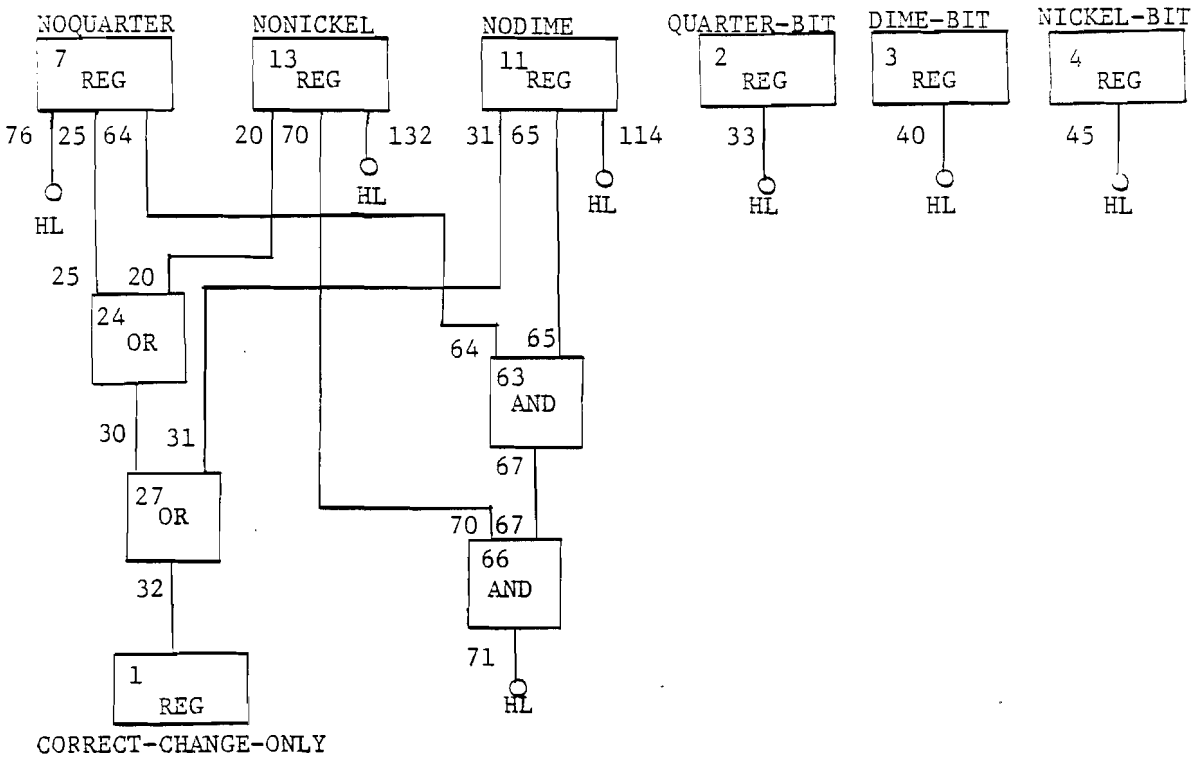


Figure C-2: Change Mechanism Path Graph (2/2)

C.2 Truncated PDP-8 ISP

```
PDP8 :=
    Begin

** Memory.State **

    mp[#0:#7777]<0:11>,                 ! Main memory (4k words)
    mb<0:11>                            ! Memory buffer

** Processor.State **

    L<>,                                ! Link bit
    AC<0:11>,                           ! Accumulator

    PC<0:11>                            ! Program counter


** Instruction.Format **

    IR\instruction.register<0:2>,       ! Operation code

    group<>             := MB<3>,
    CLA<>               := MB<4>,
    CLL<>               := MB<5>,
    CMA<>               := MB<6>,
    CML<>               := MB<7>,
    RAR<>               := MB<8>,
    RAL<>               := MB<9>,
    RTx<>               := MB<10>,
    IAC<>               := MB<11>

** Instruction.Interpretation **

    run\instruction.interpretation{main} :=
        Begin
        MB = MP[PC] next
        PC = PC + 1 next
        exec () next
        RESTART run
        End
```

```
** Instruction.Execution **{US}

    exec\instruction.execution :=
        Begin
        IR = MB<0:2> next
        DECODE IR =>
            Begin
            #0 := AND.  := AC = AC and MB,
            #1 := TAD   := L@AC = L@AC +{TC} MB,
            #2 := ISZ   := Begin
                           MB = MB + 1 next
                           If MB eql 0 => PC = PC + 1
                           End,
            #3 := DCA   := Begin
                           MB = AC next
                           AC = 0
                           End,
            #7 := OPR(),
             Otherwise := NO.OP()
            End
        End,

    opr :=
        Begin
        If not group =>
            Begin
            If CLA => AC = 0;
            If CLL => L = 0 next
            If CMA => AC = not AC;
            If CML => L = not L next
            If IAC => L@AC = L@AC + 1 next
            DECODE RTx =>
                Begin
                0 := Begin
                     If RAL => L@AC = AC@L;
                     If RAR => AC@L = L@AC
                     End,
                1 := Begin
                     If RAL => L@AC = L@AC slr 2;
                     If RAR => L@AC = L@AC srr 2
                     End
                End
            End
        End
    End
```

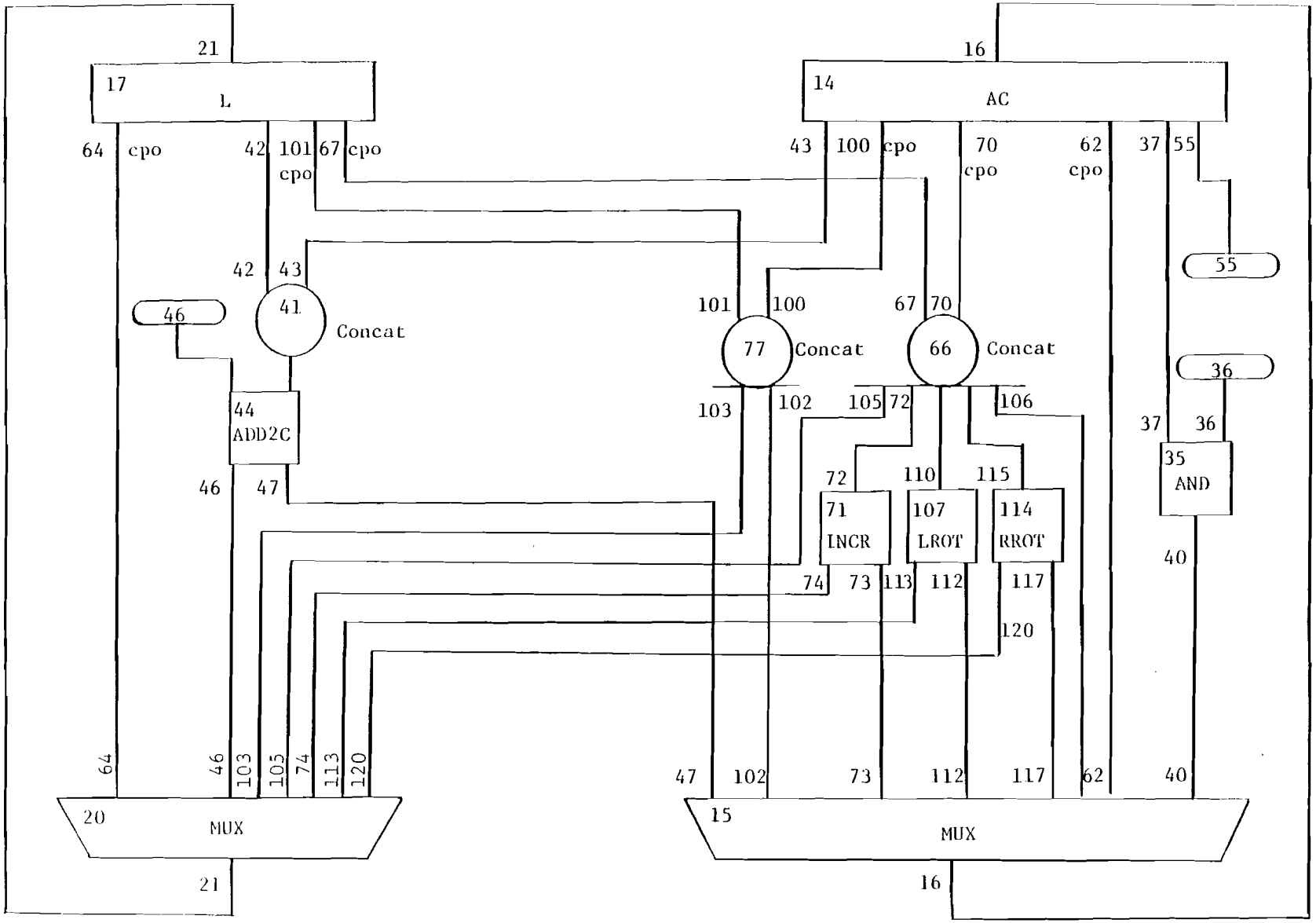


Figure C-3: Small PDP-8 Path Graph (1/2)

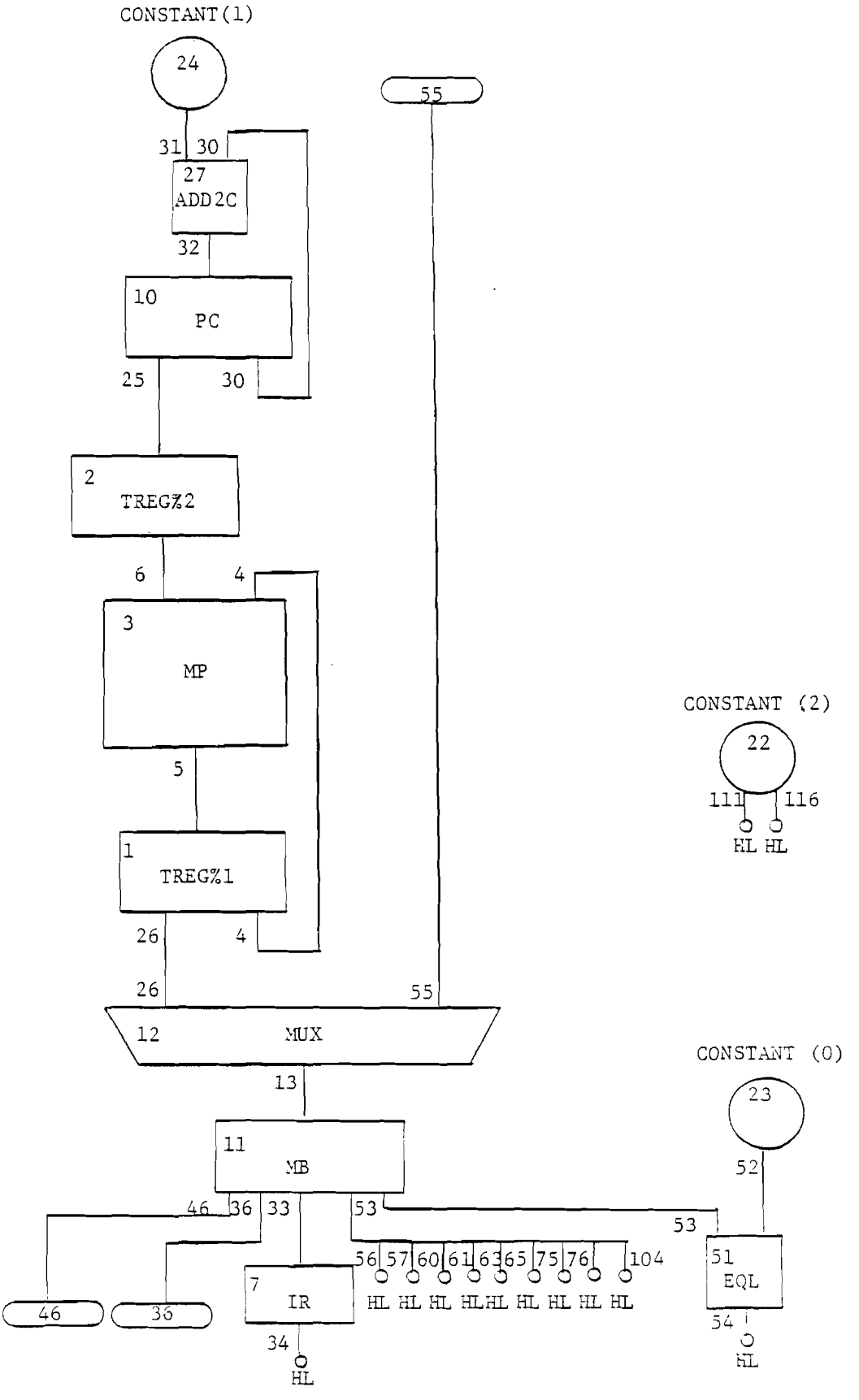


Figure C-4: Small PDP-8 Path Graph (2/2)

C.3 Full PDP-8 ISP

```
PDP8 :=
Begin

! The basic PDP-8 instruction set, not including the
! extended arithmetic element (EAE) option.  I/O instructions
! are limited to those dealing with the interrupt mechanism.
!

** Memory.State **

M\Memory[0:4095]<0:11>,

** Processor.State **

PC\Program.Counter<0:11>,

cpage\current.page<0:4>,

lac<0:12>,

   L\Link<>              := lac<0>,
   AC\Accumulator<0:11> := lac<1:12>,

mq\Multiplier.Quotient.Register<0:11>,

interrupt.state<>,

interrupt.request<>,

switches<0:11>,
```

```
** Instruction.Format **

i\instruction<0:11>,

    op\operation.code<0:2> := i<0:2>,
    ib\indirect.bit<>      := i<3>,
    pb\page.0.bit<>        := i<4>,
    pa\page.address<0:6>   := i<5:11>,

    io.select<0:5>         := i<3:8>,   ! device select
    io.control<0:2>        := i<9:11>,  ! device operation
        IO.PULSE.P1<>      := io.control<0>,
        IO.PULSE.P2<>      := io.control<1>,
        IO.PULSE.P4<>      := io.control<2>,

    rot<0:2>               := i<8:10>,  ! rotate group
    group<>                := i<3>,     ! microinstruction group
    sma<>                  := i<5>,     ! skip on minus AC
    spa<>                  := i<5>,     ! skip on positive AC
    sza<>                  := i<6>,     ! skip on zero AC
    sna<>                  := i<6>,     ! skip on AC not zero
    snl<>                  := i<7>,     ! skip on L not zero
    szl<>                  := i<7>,     ! skip on L zero
    is<>                   := i<8>,     ! invert skip sense
    cla<>                  := i<4>,     ! clear AC
    cll<>                  := i<5>,     ! clear L
    cma<>                  := i<6>,     ! complement AC
    cml<>                  := i<7>,     ! complement L
    iac<>                  := i<11>,    ! increment AC
    osr<>                  := i<9>,     ! logical or AC with SWITCHES
    hlt<>                  := i<10>,    ! halt the processor

** Address.Calculation **

eadd\effective.address<0:11> :=
   Begin
   Decode pb =>
       Begin
       0 := eadd ='00000 @ pa,
       1 := eadd = cpage @ pa
       End Next
   If ib =>
       Begin
       If eadd<0:8> Eql #001 => M[eadd] = M[eadd] + 1 Next
       eadd = M[eadd]
       End
   End,
```

```
** Interpretation.Process **

main interpret :=
    Begin
    Repeat
        Begin
        i = M[PC]; cpage = PC<0:4> Next
        PC = PC + 1 Next
        execute() Next
        If interrupt.state And interrupt.request =>
            Begin
            M[0] = PC Next
            PC = 1
            End
        End
    End,

** Execution.Processes **

execute :=
    Begin
    Decode op =>
        Begin
        #0\and := AC = AC And M[eadd()],
        #1\tad := lac = lac + ('0 @ M[eadd()]),
        #2\isz := Begin
                  M[eadd] = M[eadd()] + 1 Next
                  If M[eadd] Eql 0 => PC = PC + 1
                  End,
        #3\dca := Begin
                  M[eadd()] = AC Next
                  AC = 0
                  End,
        #4\jms := Begin
                  M[eadd()] = PC Next
                  PC = EADD + 1
                  End,
        #5\jmp := PC = eadd(),
        #6\iot := input.output(),
        #7\opr := operate()
        End
    End,
```

```
input.output :=
    Begin
    Decode i<3:11> =>
        Begin
        #001\ion := Begin                ! turn Interrupt ON
                    interrupt.state = 1 Next
                    Restart interpret
                    End,
        #002\iof := Begin                ! turn Interrupt OFF
                    interrupt.state = 0
                    End,
       Otherwise := No.Op()              ! not implemented
        End
    End,
```

```
skip<>,

skip.group :=
    Begin
    Decode is =>
        Begin
        0 := Begin
             skip = 0 Next
             If snl And (L Eql '1{US}) => skip = 1;
             If sza And (AC Eql 0) => skip = 1;
             If sma And (AC Lss 0) => skip = 1
             End,
        1 := Begin
             skip = 1 Next
             If szl And Not (L Eql{US} '0) => skip = 0;
             If sna And Not (AC Neq 0) => skip = 0;
             If spa And Not (AC Geq 0) => skip = 0
             End
        End Next
    If skip => PC = PC + 1                          ! Skip
    End,

temp6<0:5>,     ! temporary register used in byte swap
temp12<0:11>,   ! 12 bit temp register used in ac/mq swap

operate :=
    Begin
    Decode group =>
        Begin
        0 :=  Begin                                 ! group 1
              If cla => AC = 0;
              If cll => L = 0 Next
              If cma => AC = Not AC;
              If cml => L = Not L Next
              If iac => lac = lac + 1 Next
              Decode rot =>                         ! rotate group
                  Begin
                  #0     :=  No.Op(),
                  #1\bsw :=  Begin
                             temp6 = ac<0:5> Next
                             ac<0:5> = ac<6:11> Next
                             ac<6:11> = temp6
                             End,
                  #2\ral :=  lac = lac Slr 1,
                  #3\rtl :=  lac = lac Slr 2,
                  #4\rar :=  lac = lac Srr 1,
                  #5\rtr :=  lac = lac Srr 2,
                  #6     :=  No.Op(),
                  #7     :=  No.Op()
                  End
              End,
```

```
        1 :=  Begin                                  ! groups 2 and 3
              Decode i<11> =>
                 Begin
                 0 :=  Begin                         ! group 2
                       skip.group() Next
                       If cla => AC = 0 Next
                       If osr => AC = AC Or switches;
                       If hlt => STOP()
                       End,
                 1 :=  Begin                         ! group 3
                       Decode i<4:5> @ i<7> =>
                           Begin
                           0     :=  No.Op(),
                           1\mql :=  (mq = ac Next ac = 0),
                           2\mqa :=  ac = mq Or ac,
                           3\swp :=  (temp12 = mq Next
                                      mq = ac Next
                                      ac = temp12),
                           4\cla :=  ac = 0,
                           5\cam :=  (ac = 0; mq = 0),
                           6     :=  No.Op(),
                           7     :=  No.Op()
                           End
                       End
                   End
              End
        End
    End

End
```

C.4 Mark-1 ISP

```
!              The Manchester University Mark-1 Computer
!
! This is the ISPS description  of the first version
! of the machine, as reported in:
! [Lavington, S.H.,
! "A History  of Manchester  Computers",
! National Computing Centre Publications,
! Manchester, England, 1975]
!
!          Mario R. Barbacci (BARBACCI@CMUA)
!
MARK1 :=
    Begin

** Memory.State **

    M[0:8191]<31:0>,

** Processor.State **

    PI\Present.Instruction<15:0>,
        F\Function<0:2> := PI<15:13>,
        S<0:12>         := PI<12:0>,
    CR\Control.Register<12:0>,
    Acc\Accumulator<31:0>,

** Instruction.Execution ** {TC}

Main I.Cycle :=
    Begin
    PI = M[CR]<15:0> next
    Decode F =>
        Begin
        0\JMP   := CR   = M[S],
        1\JRP   := CR   = CR + M[S],
        2\LDN   := Acc  = - M[S],
        3\STO   := M[S] = Acc,
      4:5\SUB   := Acc  = Acc - M[S],
        6\CMP   := If Acc Lss 0 => CR = CR + 1,
        7\STP   := Stop()
        End next
    CR = CR + 1 next
    Restart I.Cycle
    End
End
```

Appendix D
Run Examples

This appendix includes samples of three outputs available from SYNNER. All of the outputs were captured from processing a PDP-8 design using the TTL module set. The first output is a synthesis trace that monitors the design process and gives an indication of the transformations applied to the design. The second output is a summary of the "before and after" state of the design by node class. The final output is the Module Utilization Table which summarizes the modules used in the design and provides cost and performance estimates of the design.

The outputs chosen to be included here are a small sample of the information that can be produced by SYNNER. There are sixteen (16) other types of output that can be explicitly requested.

D.1 Synthesis Trace

The Synthesis Trace is the instrumentation output from SYNNER. It documents the state of processing switches and lists the constraints in effect during processing of the design. It keeps both wall clock and CPU timing information for all the major events in the automatic processing. It logs the application of transformations and documents both trial and actual equivalence synthesis.

An indicator of SYNNER's performance can be derived from the very last line of this output. It shows that a design the complexity of the PDP-8 can be processed in under three minutes (wall clock) using eight CPU seconds.

```
SYNNER V1T(1)-1 Instrumentation Log: 27 Aug 80  02:18:45
ALLOCATOR   VER 2C(5)   FRIDAY 18 JUL 80   2:50 AM      PDP8.ISP


Entering AUTOBIND       02:19:04 [CPU:00:00.29, DIF:00:00.29]

PROFILE:
```

```
        ECHO    LOAD    LOG     VERBOSE INSTR   HSPLIT  INVERT
         OFF     OFF     ON       ON      ON      ON      ON

        REDUCE  SYNT    UNBIND  VJOIN   RENUM   QUIET   CAND
         ON      ON      OFF     ON      ON      OFF     ON

Style:  DIST            DSS File:  DIST.DSS
Family: TTL      Module Set File:  TTL.DBK

Overhead Units: DOLLARS, Overhead Operation: PLUS

Constraints:

[001]   COST MIN        Weight: 0.00
[002]   DELAY MIN       Weight: 0.00
[003]   POWER MIN       Weight: 0.00

Phase   I: Unbind/Invert        02:19:04 [CPU:00:00.29, DIF:00:00.01]
Phase III: E/T/B                02:19:05 [CPU:00:00.33, DIF:00:00.03]

Attempting Split (1) on: NODE: #0036(REG, LAC)
    Adding:   NODE: #0340(REG, %OLAC)
    Adding:   NODE: #0341(CONCAT)
    Moving:   LINK: #0115 S: #0341 D: #0113
    Moving:   LINK: #0123 S: #0341 D: #0122
    Moving:   LINK: #0133 S: #0341 D: #0003
    Moving:   LINK: #0151 S: #0341 D: #0150
    Moving:   LINK: #0157 S: #0341 D: #0026
    Moving:   LINK: #0200 S: #0341 D: #0177
    Moving:   LINK: #0212 S: #0341 D: #0036
    Moving:   LINK: #0173 S: #0341 D: #0305
    Moving:   LINK: #0204 S: #0341 D: #0311
    Moving:   LINK: #0205 S: #0341 D: #0313
    Moving:   LINK: #0226 S: #0341 D: #0317
    Moving:   LINK: #0234 S: #0341 D: #0321
    Moving:   LINK: #0253 S: #0341 D: #0327
    Moving:   LINK: #0261 S: #0341 D: #0333
    Moving:   LINK: #0273 S: #0341 D: #0337
    Moving:   LINK: #0212 S: #0340 D: #0036
    Adding:   LINK: #0342 S: #0036 D: #0341
    Adding:   LINK: #0342 S: #0036 D: #0341
    Adding:   LINK: #0343 S: #0340 D: #0341
    Adding:   LINK: #0343 S: #0340 D: #0341
    Adding:   LINK: #0345 S: #0340 D: #0036
    Adding:   LINK: #0345 S: #0340 D: #0036
    Adding:   LINK: #0346 S: #0036 D: #0340
    Adding:   LINK: #0346 S: #0036 D: #0340

Synthesizing Occurrence 1: NODE: #0122(OPER, ADD2C)
ARITH:ADD2C     *1 ADD $1 $2 #0<0>
Weight = 1.00

Installing replacement for: NODE: #0122(OPER, ADD2C)
```

```
    Adding:    NODE: #0350(CONST, 0)
    Adding:    NODE: #0353(OPER, ADD)
    Moving:    LINK: #0123 S: #0341 D: #0122
    Moving:    LINK: #0124 S: #0117 D: #0122
    Adding:    LINK: #0354 S: #0350 D: #0353

Attempting Split (1) on: NODE: #0353(OPER, ADD)
    Adding:    NODE: #0355(OPER, ADD)
    Adding:    NODE: #0356(CONCAT)
    Moving:    LINK: #0125 S: #0356 D: #0037
    Adding:    LINK: #0357 S: #0353 D: #0356
    Adding:    LINK: #0357 S: #0353 D: #0356
    Adding:    LINK: #0360 S: #0355 D: #0356
    Adding:    LINK: #0360 S: #0355 D: #0356
    Adding:    LINK: #0361 S: #0355 D: #0353
    Adding:    LINK: #0361 S: #0355 D: #0353

Synthesizing Occurrence 1: NODE: #0177(OPER, ADD2C)
ARITH:ADD2C      *1 ADD $1 $2 #0<0>
Weight = 1.00

Installing replacement for: NODE: #0177(OPER, ADD2C)
    Adding:    NODE: #0365(CONST, 0)
    Adding:    NODE: #0370(OPER, ADD)
    Moving:    LINK: #0200 S: #0341 D: #0177
    Moving:    LINK: #0201 S: #0044 D: #0177
    Adding:    LINK: #0371 S: #0365 D: #0370

Attempting Split (1) on: NODE: #0370(OPER, ADD)
    Adding:    NODE: #0372(OPER, ADD)
    Adding:    NODE: #0373(CONCAT)
    Moving:    LINK: #0202 S: #0373 D: #0037
    Adding:    LINK: #0374 S: #0370 D: #0373
    Adding:    LINK: #0374 S: #0370 D: #0373
    Adding:    LINK: #0375 S: #0372 D: #0373
    Adding:    LINK: #0375 S: #0372 D: #0373
    Adding:    LINK: #0376 S: #0372 D: #0370
    Adding:    LINK: #0376 S: #0372 D: #0370

Attempting Split (4) on: NODE: #0340(REG, %1LAC)
    Adding:    NODE: #0401(REG, %0%1LAC)
    Adding:    NODE: #0402(CONCAT)
    Moving:    LINK: #0212 S: #0402 D: #0036
    Moving:    LINK: #0343 S: #0402 D: #0341
    Moving:    LINK: #0212 S: #0401 D: #0036
    Adding:    LINK: #0403 S: #0340 D: #0402
    Adding:    LINK: #0403 S: #0340 D: #0402
    Adding:    LINK: #0404 S: #0401 D: #0402
    Adding:    LINK: #0404 S: #0401 D: #0402
    Adding:    LINK: #0406 S: #0401 D: #0340
    Adding:    LINK: #0406 S: #0401 D: #0340
    Adding:    LINK: #0407 S: #0340 D: #0401
    Adding:    LINK: #0407 S: #0340 D: #0401
```

```
Attempting Split (4) on: NODE: #0401(REG, %1%1LAC)
    Adding:   NODE: #0410(REG, %0%1%1LAC)
    Adding:   NODE: #0411(CONCAT)
    Moving:   LINK: #0212 S: #0411 D: #0036
    Moving:   LINK: #0404 S: #0411 D: #0402
    Moving:   LINK: #0212 S: #0410 D: #0036
    Adding:   LINK: #0412 S: #0401 D: #0411
    Adding:   LINK: #0412 S: #0401 D: #0411
    Adding:   LINK: #0413 S: #0410 D: #0411
    Adding:   LINK: #0413 S: #0410 D: #0411
    Adding:   LINK: #0415 S: #0410 D: #0401
    Adding:   LINK: #0415 S: #0410 D: #0401
    Adding:   LINK: #0416 S: #0401 D: #0410
    Adding:   LINK: #0416 S: #0401 D: #0410

Synthesizing Occurrence 1: NODE: #0056(OPER, ADD2C)
ARITH:ADD2C     *1 ADD $1 $2 #0<0>
Weight = 1.00

Installing replacement for: NODE: #0056(OPER, ADD2C)
    Adding:   NODE: #0420(CONST, 0)
    Adding:   NODE: #0423(OPER, ADD)
    Moving:   LINK: #0057 S: #0032 D: #0056
    Moving:   LINK: #0060 S: #0044 D: #0056
    Adding:   LINK: #0424 S: #0420 D: #0423

Synthesizing Occurrence 1: NODE: #0251(OPER, NEQ2C)
RELAT:NEQ2C     *1 NEQ $1 $2

    Synthesizing Occurrence 1: NODE: #0430(OPER, NEQ)
RELAT:NEQ       *1 NOT (EQL $1 $2 )
    Weight = 0.50

    Synthesizing Occurrence 2: NODE: #0430(OPER, NEQ)
RELAT:NEQ       *1 XOR $1 $2
                *2 OR [*1<0> ]

        Synthesizing Occurrence 1: NODE: #0453(OPER, OR)
LOGIC:OR        *1 OR $1 (OR [$2 ] )
        Weight = 1.00
    Weight = 1.00
Weight = 0.25

Synthesizing Occurrence 2: NODE: #0251(OPER, NEQ2C)
RELAT:NEQ2C     *1 NOT (EQL2C $1 $2 )

    Synthesizing Occurrence 1: NODE: #0431(OPER, EQL2C)
RELAT:EQL2C     *1 EQL $1 $2
    Weight = 1.00
Weight = 0.50

Installing replacement for: NODE: #0251(OPER, NEQ2C)
```

```
    Adding:   NODE: #0436(OPER, EQL)
    Adding:   NODE: #0432(OPER, NOT)
    Moving:   LINK: #0252 S: #0045 D: #0251
    Moving:   LINK: #0326 S: #0327 D: #0251
    Adding:   LINK: #0437 S: #0436 D: #0432

Synthesizing Occurrence 1: NODE: #0260(OPER, GEQ2C)
RELAT:GEQ2C     *1 NOT (LSS2C $1 $2 )

    Synthesizing Occurrence 1: NODE: #0444(OPER, LSS2C)
RELAT:LSS2C     *1 SUB $1 $2
    Weight = 1.00
Weight = 0.50

Synthesizing Occurrence 2: NODE: #0260(OPER, GEQ2C)
RELAT:GEQ2C     *1 OR (GTR2C $1 $2 )<0> (EQL2C $1 $2 )<0>

    Synthesizing Occurrence 1: NODE: #0444(OPER, EQL2C)
RELAT:EQL2C     *1 EQL $1 $2
    Weight = 1.00

    Synthesizing Occurrence 1: NODE: #0450(OPER, GTR2C)
RELAT:GTR2C     *1 XOR $1<0> $2<0>
                *2 AND *1 $2<0>
                *3 AND (NOT *1 ) (GTR $1<1:N> $2<1:N> )
                *4 OR *2 *3
    Weight = 0.07
Weight = 0.01

    Synthesizing Occurrence 1: NODE: #0444(OPER, LSS2C)
RELAT:LSS2C     *1 SUB $1 $2
    Weight = 1.00

Re-Synthesizing Occurrence 2: NODE: #0260(OPER, GEQ2C)
Weight = 0.50

Installing replacement for: NODE: #0260(OPER, GEQ2C)
    Adding:   NODE: #0451(OPER, SUB)
    Adding:   NODE: #0445(OPER, NOT)
    Moving:   LINK: #0262 S: #0045 D: #0260
    Moving:   LINK: #0332 S: #0333 D: #0260
    Adding:   LINK: #0452 S: #0451 D: #0445

Synthesizing Occurrence 1: NODE: #0136(OPER, ADD2C)
ARITH:ADD2C     *1 ADD $1 $2 #0<0>
Weight = 1.00

Installing replacement for: NODE: #0136(OPER, ADD2C)
    Adding:   NODE: #0454(CONST, 0)
    Adding:   NODE: #0457(OPER, ADD)
    Moving:   LINK: #0137 S: #0014 D: #0136
    Moving:   LINK: #0140 S: #0044 D: #0136
    Adding:   LINK: #0460 S: #0454 D: #0457
```

```
Synthesizing Occurrence 1: NODE: #0106(OPER, ADD2C)
ARITH:ADD2C     *1 ADD $1 $2 #0<0>
Weight = 1.00

Installing replacement for: NODE: #0106(OPER, ADD2C)
    Adding:   NODE: #0462(CONST, 0)
    Adding:   NODE: #0465(OPER, ADD)
    Moving:   LINK: #0107 S: #0102 D: #0106
    Moving:   LINK: #0110 S: #0044 D: #0106
    Adding:   LINK: #0466 S: #0462 D: #0465

Synthesizing Occurrence 1: NODE: #0126(OPER, EQL2C)
RELAT:EQL2C     *1 EQL $1 $2
Weight = 1.00

Installing replacement for: NODE: #0126(OPER, EQL2C)
    Adding:   NODE: #0472(OPER, EQL)
    Moving:   LINK: #0127 S: #0102 D: #0126
    Moving:   LINK: #0130 S: #0045 D: #0126

Synthesizing Occurrence 1: NODE: #0224(OPER, EQL2C)
RELAT:EQL2C     *1 EQL $1 $2
Weight = 1.00

Installing replacement for: NODE: #0224(OPER, EQL2C)
    Adding:   NODE: #0476(OPER, EQL)
    Moving:   LINK: #0225 S: #0045 D: #0224
    Moving:   LINK: #0316 S: #0317 D: #0224

Synthesizing Occurrence 1: NODE: #0233(OPER, LSS2C)
RELAT:LSS2C     *1 SUB $1 $2
Weight = 1.00

Installing replacement for: NODE: #0233(OPER, LSS2C)
    Adding:   NODE: #0502(OPER, SUB)
    Moving:   LINK: #0235 S: #0045 D: #0233
    Moving:   LINK: #0320 S: #0321 D: #0233

Synthesizing Occurrence 1: NODE: #0075(OPER, EQL2C)
RELAT:EQL2C     *1 EQL $1 $2
Weight = 1.00

Installing replacement for: NODE: #0075(OPER, EQL2C)
    Adding:   NODE: #0506(OPER, EQL)
    Moving:   LINK: #0076 S: #0014 D: #0075
    Moving:   LINK: #0077 S: #0050 D: #0075

AUTOBINDING Completed   02:21:57 [CPU:00:08.32, DIF:00:07.100]
    Total AUTO: Wall Clock: 00:02:52, CPU: 00:08.65
```

D.2 Synthesis Summary

The Synthesis Summary documents the net change in types of nodes due to design synthesis.

```
VARIABLE CARRIERS (VC)  Before:  28, After:  36, Change:   8
    REG Before:  18, After:  21, Change:   3
        REG     Before:   9, After:  11, Change:   2
        TREG    Before:   2, After:   2, Change:   0
        FLAG    Before:   7, After:   8, Change:   1
    MEMORY      Before:   1, After:   1, Change:   0
        MEMORY  Before:   1, After:   1, Change:   0
    CONST       Before:   8, After:  13, Change:   5
    TREG        Before:   1, After:   1, Change:   0
        TREG    Before:   1, After:   1, Change:   0

PATH OPERATORS (PO)     Before:  10, After:  15, Change:   5
    MUX         Before:   7, After:   7, Change:   0
    CONCAT      Before:   3, After:   8, Change:   5

PATH CARRIERS (PC)      Before: 146, After: 174, Change:  28
    LINK        Before: 119, After: 147, Change:  28
    HLINK       Before:  27, After:  27, Change:   0

VARIABLE OPERATORS (VO) Before:  39, After:  39, Change:   0
    OPER        Before:  39, After:  39, Change:   0
        ARITH   Before:   5, After:   9, Change:   4
            ADD2C       Before:   5, After:   0, Change:  -5
            ADD Before:   0, After:   7, Change:   7
            SUB Before:   0, After:   2, Change:   2
        RELAT   Before:   8, After:   6, Change:  -2
            EQL2C       Before:   3, After:   0, Change:  -3
            EQL Before:   2, After:   6, Change:   4
            LSS2C       Before:   1, After:   0, Change:  -1
            NEQ2C       Before:   1, After:   0, Change:  -1
            GEQ2C       Before:   1, After:   0, Change:  -1
        LOGIC   Before:  26, After:  24, Change:  -2
            AND Before:   9, After:   9, Change:   0
            OR  Before:   3, After:   3, Change:   0
            NOT Before:  14, After:  12, Change:  -2
```

D.3 Module Utilization Table

The Module Utilization Table summarizes the module requirements for implementing the design. It also provides performance information in terms of total cost, total power, and estimates of the speed of the design and the controller requirements.

```
ALLOCATOR   VER 2C(5)   FRIDAY 18 JUL 80   2:50 AM      PDP8.ISP

        ****** MUT -  MODULE UTILIZATION TABLE ******

-------------------------------------------------------------------

Module: SN7404  Function: NOT          Modules:  87    Packages:  15
Module: SN7408  Function: AND          Modules:  20    Packages:   5
Module: SN7432  Function: OR           Modules:  21    Packages:   6
Module: SN7474  Function: DFLOP        Modules:   8    Packages:   4
Module: SN7483  Function: ADD          Modules:  17    Packages:  17
Module: SN7485  Function: COMPA        Modules:  14    Packages:  14
Module: FMEMORY Function: MEMORY       Modules:   1    Packages:   1
Module: SN74150 Function: MUX          Modules:  13    Packages:  13
Module: SN74153 Function: MUX          Modules:  49    Packages:  25
Module: SN74157 Function: MUX          Modules:  12    Packages:   3
Module: SN74174 Function: REG          Modules:  16    Packages:  16
Module: SN74181 Function: ALU          Modules:   6    Packages:   6
Module: SN74194 Function: REG          Modules:   7    Packages:   7

Module Allocation Summary Data:

Total Bindable Nodes:     69
Nodes Actually Bound:     69
Percent Binding:         100%
Total Packages Used:     132
Total Modules Used:      271
Total Spare Modules:       7
Module Utilization:       97%
Total Power:          28437.00 MW
Control Words:          242
Control Word Size:       77 Bits
Max Control Path:        49
Max Path Delay:        1853.00 NS
Raw Package Cost:     $  88.13
Total Package Cost:   $ 484.13

-------------------------------------------------------------------
```

Appendix E
Module Database Entries

TTL Entries in the Module Database

Package	Function	
SN7400	NAND	(2 input)
SN7402	NOR	(2 input)
SN7404	NOT	
SN7408	AND	(2 input)
SN7410	NAND	(3 input)
SN7411	AND	(3 input)
SN7420	NAND	(4 input)
SN7421	AND	(4 input)
SN7427	NOR	(3 input)
SN7430	NAND	(8 input)
SN7432	OR	(2 input)
SN7474	D Flip-Flop	(1 bit)
SN7476	JK Flip-Flop	(1 bit)
SN7480	Adder	(1 bit)
SN7482	Adder	(2 bit)
SN7483	Adder	(4 bit)
SN7485	Compare	(4 bit)
SN7486	XOR	(2 input)
SN7491	Shift-Register	(8 bit)
SN74S133	NAND	(13 input)
SN74150	Multiplexor	(16 to 1)
SN74151	Multiplexor	(8 to 1)
SN74153	Multiplexor	(4 to 1)
SN74157	Multiplexor	(2 to 1)
SN74161	Counter	(4 bit)
SN74174	Register	(6 bit)
SN74181	ALU	(4 bit)
SN74191	Counter	(4 bit)
SN74194	Shift Register	(4 bit)
SN74260	NOR	(5 input)
SN74276	JK Flip-Flop	(1 bit)

Sandia Cell Entries in the Module Database

Package	Function	
ADD1	Adder	(1 bit - hand designed)
ADD4	Adder	(4 bit - hand designed)
SC1120	NOR	(2 input)
SC1130	NOR	(3 input)
SC1140	NOR	(4 input)
SC1220	NAND	(2 input)
SC1230	NAND	(3 input)
SC1240	NAND	(4 input)
SC1310	NOT	
SC1330	Multiplexor	(2 to 1)
SC1350	Multiplexor	(3 to 1 - unencoded select)
SC1420	RS Flip-Flop	(1 bit)
SC1430	Beginning Counter	(1 bit)
SC1440	Middle Counter	(1 bit)
SC1450	Ending Counter	(1 bit)
SC1460	D Flip-Flop	(1 bit)
SC1480	D Flip-Flop	(1 bit)
SC1490	D Flip-Flop	(1 bit)
SC1520	NOT	
SC1620	AND	(2 input)
SC1630	AND	(3 input)
SC1640	AND	(4 input)
SC1650	AND	(5 input)
SC1720	OR	(2 input)
SC1730	OR	(3 input)
SC1800	Multiplexor	(4 to 1 - unencoded select)
SC1820	D Flip-Flop	(1 bit)
SC2310	XOR	
SC2320	XOR	

References

[Barbacci 73] M. R. Barbacci.
Automated Exploration of the Design Space For Register Transfer (RT) Systems.
PhD thesis, Carnegie-Mellon University, November, 1973.

[Barbacci 79] M. R. Barbacci, G. E. Barnes, R. G. Cattell, D. P. Siewiorek.
The Symbolic Manipulation of Computer Descriptions: ISPS Computer Description Language
Carnegie-Mellon University, 1979.

[Barbacci 81] M. R. Barbacci.
Instruction Set Processor Specifications (ISPS): The Notation and its Applications.
IEEE Computer Society, Transactions on Computers C-30(1), January, 1981.

[Bevington 69] P. R. Bevington.
Data Reduction and Error Analysis for the Physical Sciences.
McGraw-Hill Book Company, 1969.

[Blakeslee 75] T. R. Blakeslee.
Digital Design with Standard MSI and LSI.
John Wiley & Sons, 1975.

[Clark 67] W. A. Clark.
Macromodular Computer Systems.
In *AFIPS Conference Proceedings 30*, pages 335-402. SJCC, Atlantic City, N.J, 1967.

[Cloutier 80] R. J. Cloutier.
Control Allocation: the Automated Design of Digital Controllers.
Master's thesis, Carnegie-Mellon University, April, 1980.

[Ernst 69] G. W. Ernst and A. Newell.
GPS: A Case Study in Generality and Problem Solving.
Academic Press, 1969.

[Hafer 77] L. J. Hafer.
Data-Memory Allocation in the Distributed Logic Design Style.
Master's thesis, Carnegie-Mellon University, December, 1977.

[Hafer 78] L. J. Hafer and A. C. Parker.
Register-Transfer Level Automatic Digital Design: The Allocation Process.
In *Proceedings of the 15th Design Automation Conference*. IEEE, 1978.

[Hafer 79] L. J. Hafer.
Micro-operation Documentation.
Technical Report, Carnegie-Mellon University, 1979.

[Jensen 74] K. Jensen and N. Wirth.
PASCAL User Manual and Report.
Springer-Verlag, 1974.

[Johannsen 79] D. Johannsen.
Bristle Blocks: A Silicon Compiler.
In *16th Design Automation Conference Proceedings*, pages 310-313. IEEE Computer Society, IEEE, 1979.

[Kim 79] J. H. Kim.
Issues in Translation of High Level Abstract Designs to IC Layout.
Master's thesis, Carnegie-Mellon University, November, 1979.

[Lawson 78] G. L. Lawson.
Design Style Selector, An Automated Computer Program Implementation.
Master's thesis, Carnegie-Mellon University, August, 1978.

[Leive 77] G. W. Leive.
The Binding of Modules to Abstract Digital Hardware Descriptions.
PhD Thesis Proposal, Electrical Engineering Department, Carnegie-Mellon University, 1977.

[Leive 79] G. W. Leive and D. E. Thomas.
The CMU Design System, Module Database - User's Guide
Carnegie-Mellon University, 1979.

[Leive 80] G. W. Leive.
The SYNNER's Guide
First edition, Carnegie-Mellon University, 1980.

[Marwedel 79] P. Marwedel.
The MIMOLA Design System: Detailed Description of the Software System.
In *16th Design Automation Conference Proceedings*, pages 59-63. IEEE Computer Society, IEEE, 1979.

[McFarland 79] M. C. McFarland.
Global Transformations on Abstract Hardware Descriptions: A Formal Approach.
PhD Thesis Proposal, Electrical Engineering Department, October 1979.

[Nagle 80] A. W. Nagle.
Automated Design of Digital-System Control Sequencers from Register-Transfer Specifications.
PhD thesis, Carnegie-Mellon University, 1980.

[Parker 79] A. Parker, D. Thomas, D. Siewiorek, M. Barbacci, L.Hafer, G.Leive, J.Kim.
The CMU Design Automation System: An Example of Automated Data Path Design.
In *16th Design Automation Conference Proceedings*, pages 73-80. IEEE Computer Society, IEEE, 1979.

[Rege 74] S. L. Rege.
Designing Variable Data Format Modules With Cost-Performance Tradeoffs.
PhD thesis, Carnegie-Mellon University, August, 1974.

[Sandia 78] Sandia Staff.
Standard Cell User's Guide.
Sandia Laboratories, 1978.

[Siewiorek 76] D. P. Siewiorek, M. R. Barbacci.
The CMU RTCAD System: An Innovative Approach to Computer Aided Design.
In *AFIPS Conference Proceedings, vol. 45*, pages 643-655. AFIPS, 1976.

[Snow 78] E. A. Snow.
Automation of Module Set Independent Register-Transfer Level Design.
PhD thesis, Carnegie-Mellon University, April, 1978.

[Sussman 79] G. J. Sussman, J. Holloway, and T. F. Knight, Jr.
Computer Aided Evolutionary Design for Digital integrated Systems.
Technical Report AI Memo No. 526, Massachusetts Institute of Technology, May, 1979.

[Thomas 81] D. E. Thomas and D. P. Siewiorek.
Measuring Designer Performance to Verify Design Automation Systems.
IEEE Transactions on Computers , 1981.

[Weiss 79] R. Weiss.
DataBook Editor.
Project Report, Electrical Engineering Department, Carnegie-Mellon University, 1979.

[Wiederhold 77] G. Wiederhold.
Database Design.
McGraw-Hill Book Company, 1977.

[Zimmermann 79] G. Zimmermann.
The MIMOLA Design System: A Computer Aided Digital Processor Design Method.
In *16th Design Automation Conference Proceedings*, pages 53-58. IEEE Computer Society, IEEE, 1979.