A Framework for Practical Microcode Synthesis

Michael Douglas Poe
Ph.D. Thesis
Computer and Information Science Department
University of Massachusetts

COINS Technical Report 87-70

February 1987

A FRAMEWORK FOR PRACTICAL MICROCODE SYNTHESIS

A Dissertation Presented

by

MICHAEL DOUGLAS POE

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree

DOCTOR OF PHILOSOPHY

February 1987

Department of Computer and Information Science

The following are the trademarks of Digital Equipment Corporation:
DEC PDP-8 PDP-11 VAX VMS DIGITAL

The following are the trademarks of Hewlett-Packard Company:
HP

The following are the trademarks of Zilog:
Zilog Z80

The following are the trademarks of Motorola Incorporated:
Motorola 68000

A FRAMEWORK FOR PRACTICAL MICROCODE SYNTHESIS

A Dissertation Presented

by

MICHAEL DOUGLAS POE

Approved as to style and content by:

_____

Dr. Robert Moll, Chairperson of Committee

_____

Dr. Victor Lesser, Member

_____

Dr. Howard Peelle, Member

_____

Dr. Jack Wileden, Member

_____

Dr. W. Richards Adrion, Department Head
Computer and Information Science

TABLE OF CONTENTS

Chapter

Dedicated to:

The Microkids of the book Soul of a New Machine

"Software is still a cottage industry that needs to
come through an industrial revolution."
William Wulf [WULF82]

Thanks to:

James E. Carr of Palo Alto, California
and the authors of MacProof ™
for technical-writing assistance

ABSTRACT

A FRAMEWORK FOR PRACTICAL MICROCODE SYNTHESIS

FEBRUARY 1987

MICHAEL DOUGLAS POE, B. A., REED COLLEGE

M.S., Ph.D., UNIVERSITY OF MASSACHUSETTS AT AMHERST

Directed by: Professor Robert Moll

This thesis describes a knowledge engineering approach to microcode synthesis. The implementation of the thesis is a rule-based expert system called UMS (Universal Microcode Synthesizer). UMS takes a high-level procedural description of the desired microcode and target hardware, and produces a microprogram for that hardware. The synthesis process uses a minor amount of example-dependent information. An example of synthesizing microcode to emulate a PDP-8 shows that procedural hardware and instruction-set descriptions contain enough information for microcode synthesis.

The UMS code analysis techniques for procedural descriptions create highly detailed control-and-data-dependency graphs used in later synthesis processing. Description techniques are provided in the thesis for many advanced microengine features not usually discussed in the microcode compilation and synthesis literature. The code analysis techniques are demonstrated to be powerful enough for independently written descriptions in a synthesis example.

General truth-preserving code-transformation rules are applied to the microcode description until it becomes the desired microprogram. This transformation process has a large search space of three interrelated tasks: 1) determine how to change the operations specified in the high-level microcode to those available in hardware; 2) search the hardware for ways to make these operations actually occur; and 3) search to make the resulting microprogram efficient in speed and space.

The design of the UMS software kernel, called the inference engine, is a primary contribution of the thesis. Using cost analysis, the engine manages search to make microcode synthesis practical. Specialized search techniques control how the synthesizer inference engine matches parts of the instruction-set control-and-data-dependency graph to its hardware description counterparts. Many novel equivalence classes for semantically similar graph fragments allow matches between graph fragments that are not literal matches. Graph-correspondence search is decomposed into multiple node and arc types, arranged by cost of heuristic search. Arc feature constraints are accumulated while node choice is delayed as long as possible. Based on search cost estimates, a search path may be suspended and possibly resumed later. The engine tries to find a way for each search path to fail. Such a failure triggers a shift in search direction.

LIST OF FIGURES

LIST OF ALGORITHMS

CHAPTER I

OVERVIEW AND INTRODUCTION

1.1 Overview

1.1.1 Problem statement

Different models in a family of computers are based on different hardware designs. The models have a common instruction set, which allows them to all run the same software. A software-based technique called microprogramming achieves this uniformity by controlling hardware actions. Each computer model requires a new microprogram that typically is developed with a low-level assembly language. This thesis describes a strategy to automate microcode production.

The world of the microprogrammer is fundamentally different than that of a conventional programmer, and conventional programmers who enter the field must go through a period of reorientation. The microprogrammer must deal with at least an order of magnitude more details per line of code than the conventional programmer. Particularly, the hardware topology and characteristics of the datapaths must be understood, and these may vary with each microinstruction. In this decade, computer architectures are increasing in their regularity. This makes for easier compilation, and increases code quality.

On the other hand, most current hardware microarchitectures, which have been profoundly irregular, are becoming correspondingly more complex. However, the most important difference between conventional programming and microprogramming is parallelism, where multiple actions occur simultaneously in the hardware.

Due to the volume of microcode required by current instruction sets, the design of the microprogram has become as challenging and error prone a problem as that of the rest of the hardware [PATGPS81], [KIDD81]. Falk summarizes the state of microprogramming:

"At present, microprogramming is an elite activity, performed effectively only by a small number of expert practitioners. The work is detailed, precise, time-consuming, and considerably more expensive than present-day software programming" [FALK74].

1.1.2 Thesis approach

This thesis describes microcode-synthesis techniques and their implementation, collectively called the Universal Microcode Synthesizer (UMS), which automates the microprogramming task. The eight chapters of the thesis include a discussion of microprogramming's history and technology, a comparison of microcode-development methodologies (hand-coding, compilation, and synthesis), a literature review, a description of each phase of the microcode-synthesis process, results from a code-generation experiment using UMS, and conclusions.

1.1.3 UMS approach

UMS is a knowledge-engineering approach to microcode synthesis implemented in over 300 pages of Prolog source code as a rule-based expert system. The UMS method of automatic microcode generation relies heavily on ideas from artificial intelligence (see [BARR81], [BARR82], [COHF82], and discussion in Chapter II), in particular search theory ([BARR81], [PEA84]), program synthesis ([BARS79]), program transformation ([FEA82]), and program refinement ([SMI83], [WAL77]).

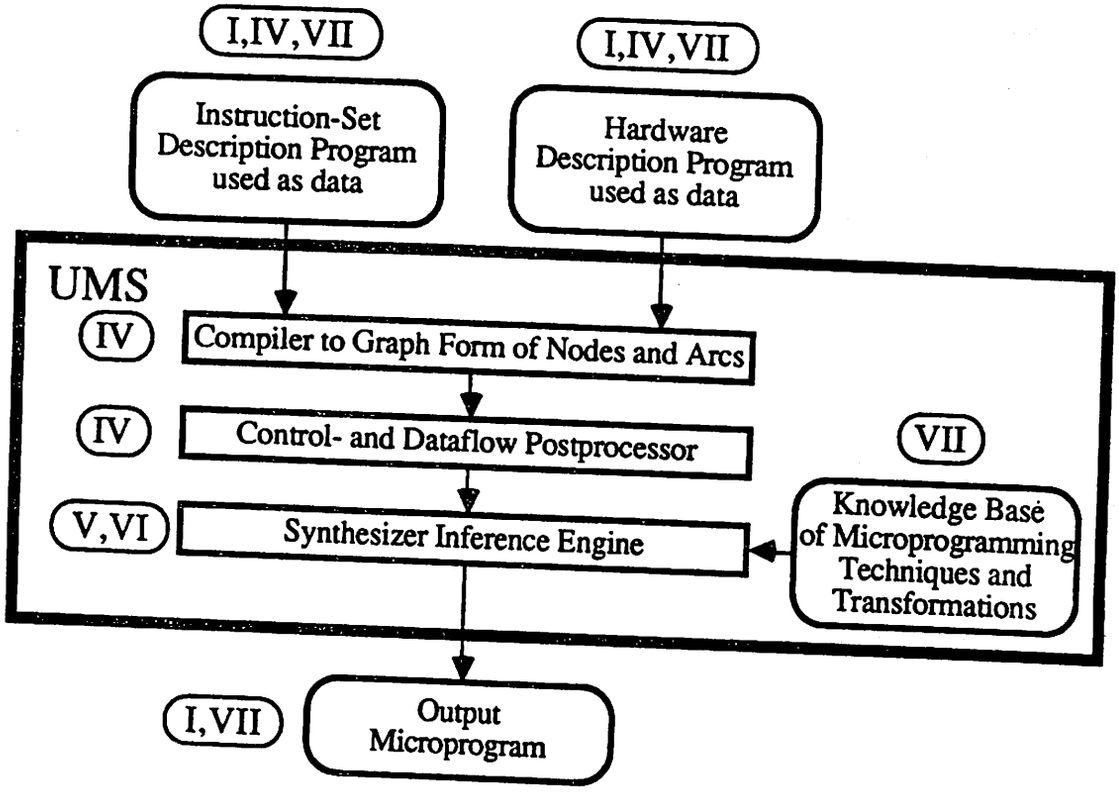


Figure 1.1: Structure of the Universal Microcode Synthesizer (UMS). The large box shows the contents of the thesis implementation. The numbered ovals refer to the chapters that correspondingly describe each component.

UMS microprogram creation is based primarily on two description inputs: a description of the target hardware and the desired instruction set (see Figure 1.1). These descriptions are written in a relatively conventional procedural language without expectation of their use by a synthesis system. These two inputs are compiled into a special format, which is later annotated with additional control-and-data-dependency analysis information that describes which conditions and data values affect each calculation respectively. The inference engine of a rule-based expert system transforms the instruction-set specification into microcode based on further analysis of hardware requirements. This activity uses a knowledge base of microprogramming techniques and transformations. The output of UMS is a microprogram that implements the instruction set within the hardware.

Higher level languages for microprogramming have not become commonplace because quality code generators for microcode compilers are very difficult to write. UMS uses some techniques similar to conventional retargetable compiler technology. However, microcode synthesis differs from conventional compiler technology in two ways:

- Object microcode must take into account the operand-routing characteristics of the target hardware.
- In order to produce efficient microcode at minimum hardware resource cost, parallelism must be detected, analyzed, and exploited.

1.1.4 Major UMS contributions

This thesis extends the use of expert system technology to a new problem domain: microcode synthesis. UMS provides an integrated approach to microprogram synthesis using domain-specific improvements in data structures, hardware and instruction-set description techniques, and search strategies.

1.1.4.1 Hardware description and code-analysis techniques. The UMS code analysis techniques for procedural hardware and instruction-set descriptions create detailed control-and-data-dependency graphs used in later synthesis processing. These control-and-data-dependency graphs are implicit in the semantics of ordinary procedural languages such as ISPS (which is used by UMS), C, or Pascal. The procedural hardware-description techniques support many advanced microengine features (such as tristate buses, pipelined datapaths, variable time-delay operations, and asynchronous events) that are not usually discussed in the microcode-compilation or microcode-synthesis literature. Previous synthesis systems have relied on difficult-to-use instruction-set (source) and hardware (object) description languages with specialized or nonprocedural semantics.

1.1.4.2 Microcode synthesis based on graph data-structure search, comparison, and transformation. The hardware description graph is searched for similarity to and compared with each fragment of the

instruction-set description graph. Where the graphs are members of the same graph-semantics equivalence class, the instruction-set graph fragments are associated with similar hardware fragments (for dataflow synthesis) and their corresponding microword field values (for controlflow synthesis). Where the two graphs are not members of the same equivalence class, the instruction-set description is transformed to more closely resemble the hardware. This microcode-synthesis technique has a simpler theoretical basis than previous unimplemented formal synthesis methods.

1.1.4.3 Navigation through a large graph-equivalence and code-transformation microcode-synthesis search space using domain-specific search-management techniques. The software kernel of UMS, called the inference engine [POE84], is implemented as a heuristic-controlled production system. The inference engine applies general truth-preserving code-transformation rules or programming tricks (such as in [KIB78]) to rewrite and refine the instruction-set description until it becomes target-hardware microcode. This transformation process requires navigation through a large search space for three interrelated tasks:

1) Determine how to change the operations specified in the high-level microcode to those available in hardware;

2) Search the hardware for ways to make these operations occur in a coordinated manner, and

3) Search to make the resulting microprogram efficient in speed and space.

The first task is application of conventional microprogramming techniques. The thesis contributes new techniques for solving the last two tasks in a practical way.

At a high level, synthesis is controlled by cost analysis and search management. At a lower level, an approach to simultaneously considering all data dependencies, resource constraints, code quality (especially horizontal parallelism), and cost (time and space) has been designed and appears to perform a type of microcode compaction. This approach is based on an unimplemented data structure and its associated iterative algorithm called the Resource Utilization Template, and is intended to be used during code generation instead of as a separate optimization phase. For the examples considered in the thesis, the dataflow synthesis techniques allow generation of microcode with quality comparable to hand-generated code.

1.1.5 Overview of chapter I

After defining the thesis topic, this chapter overviews the thesis contents. Later in the chapter a microcode sample is used to provide a detailed comparison of conventional microassembler, conventional microcode compiler, and new microcode-synthesis software technology.

1.1.6 Overview of chapter II: literature review

Chapter II describes previous microprogram-development methodology approaches along with some of the relevant ideas from program synthesis and other areas of artificial intelligence. The new techniques developed for this thesis are described in more detail in the appropriate chapters.

One of the key issues in microprogramming is the parallelism that a microprogrammer must consider in every line of code. A whole V-Compiler [PATGPS81] phase deals with this problem and entire Ph.D. dissertations (such as [MAL78], and [FIS79]) have dealt with this topic. Optimally placing the microoperations into microwords has been shown to be exponentially hard even for the small fragments called basic blocks [ULL73]. Fortunately, carefully chosen heuristics can be shown to be near-optimal solutions [FIS79]. Much of this thesis, as well, depends on heuristic search ([NIL80], [BARR81], [BARR82], [COHF82], etc.). Chapter II describes in detail how these and other issues of artificial intelligence and traditional computer science relate to this thesis.

1.1.7 Overview of chapter III: design of a retargetable microcode compiler to use synthesis techniques

The design of UMS builds on many of the ideas of the V-Compiler [PATGPS81], a retargetable microcode compiler described in Chapter III. Since the general V-Compiler organization is similar to conventional compilers, its description provides insights to readers new to microcode compilation.

However, the V-Compiler implements most phases of compilation with productions [ROS83], [GRE81], [WATH78], [DAVK77]. The use of productions allowed tuning continually each phase of the compiler for individual microengines, and the code produced was expected to reach hand-coded quality. Discussion in other parts of the thesis describe how UMS output is expected to exceed the code quality of the V-Compiler.

1.1.8 Overview of chapter IV: routing and parallelism analysis of machine descriptions

UMS activities are based on analysis of hardware and instruction-set descriptions written in a procedural form. Both descriptions are written in the programming language ISPS [BARBCS78], [BAR79], [PARTCC79], [BARS82]. Chapter IV discusses techniques to procedurally describe many common microengine features. This chapter also describes two primary types of machine-description analysis of these descriptions required by UMS; these are operand routing and microoperation parallelism.

In UMS source code must be compiled into an internal form before performing operand routing or microoperation-parallelism analysis. Instead of producing conventional object code, UMS compilation produces a graph-like data structure. A later part of compilation called the postprocessor performs global controlflow and dataflow analysis (discussed in [ANKCHM82], [MUCJ81], and [HEC77]) to extract completely the program's meaning.

A high-level task for the hardware to perform may be realized in more than one way. A new but unimplemented concept, called the Resource Utilization Template (RUT), has been developed to describe this variability [POE81B]. The RUT can compare different microcode translations with a single measurement. This measurement is intended to aid parallelism analysis within microcode fragments. The RUT can assist deciding which semantic translation should be used in a particular situation while implicitly taking into account the set of microoperation-placement permutations.

1.1.9 Overview of chapter V: microcode dataflow-synthesis heuristics

Chapter V describes how UMS dataflow synthesis derives from the instruction-set description a microprogram consisting of microoperations to be run on the target hardware. Although some of the heuristics used in this process are introduced in Chapter I, these heuristics are described in detail in Chapter V. Implementation of UMS's heuristics were discussed from an expert system point of view in [POE84].

1.1.10 Overview of chapter VI: controlflow synthesis and microcode compaction

Chapter VI describes microcode compaction, the technique that assigns microoperations to microwords in a manner consistent with the details of the microengine design. This is the primary way to exploit hardware parallelism in

microcode. The dataflow synthesis described in Chapter V does not consider the microengine's controlflow constraints, since specific microwords are not associated with synthesized microoperations. In UMS, compaction is intended to be based on the RUT described in Chapter IV.

Microcode compaction, also called packing, has been shown to be exponentially difficult [ULL73]. Chapter VI reviews some of the techniques that allow efficient compacting of local microcode (code without controlflow forks or joins). Local compaction is now relatively well understood, but a new problem has taken its place: global microcode compaction (code with controlflow forks or joins).

Global microcode compaction can move microoperations between basic blocks (segments of code with controlflow forks or joins only at their beginning or end). Global packing can also duplicate or consolidate microoperations at controlflow forks and joins or move microoperations in to and out of controlflow loops. A phase of the V-Compiler written in Bliss performs this activity [POE80], and an intermediate language has been designed to facilitate its use [POE81A]. The UMS global compaction strategy extends previous work and is intended to be based on using the RUT.

1.1.11 Overview of chapter VII: an example of synthesizing and compiling optimized microcode

Chapter VII contains the primary example of UMS at work. Siewiorek, Bell and Newell's classic book, *Computer Structures Principles and Examples* [SIEBN82], contains the principal example of microcode synthesis for this thesis. The book provides a procedural description of a PDP-8 instruction set (page 125, also discussed as an example in [BAR79]), a procedural description of an AM2901 bitslice chip-based microengine (page 219), and a corresponding hand-written microprogram (page 226), all written in ISPS ([BARBCS78], [BAR79], [PARTCC79], [BARS82]). Chapter VII compares the quality of this synthesized microprogram to the hand-written version.

1.1.12 Overview of chapter VIII: conclusions and suggested future work

Chapter VII summarizes experience with UMS and suggests that microcode-synthesis techniques will become a practical technology. Future work will offer speed improvements that will blur further the distinction between synthesis and compilation. The work done here could provide a basis for the synthesis of microarchitectures or hardware.

1.1.13 Thesis-based publications

Six previously published papers document intermediate results of the thesis effort. Chapter III describes a group effort in which the author took part to

design an easily retargetable microcode compiler. An early version of some of this work appeared as "V-Compiler: A Next Generation Tool for Microprogramming," by D. Patterson, R. Goodell, M. D. Poe, and S. Steely, *Proceedings NCC*, 1981, pages 103 to 109. Chapter IV is devoted to analysis of machine description, particularly approaches to detect and represent the inherent parallelism found in horizontal microengines. An early version of this work appeared as "Issues of the Design of a Low Level Microprogramming Language for Global Microcode Compaction," by M. D. Poe, R. Goodell, and S. Steely, in *Proceedings of the 14th Annual Microprogramming Workshop*, ACM, 12-Oct-81, pages 88 to 94. The implementation in Chapter V of the thesis describes dataflow synthesis of microcode from a procedural description of the microengine hardware. Some of this work is represented as heuristics in the microprogramming knowledge base. Chapter V is based in part on "Control of Heuristic Search in a PROLOG-based Microcode Synthesis Expert System," by M. D. Poe, *International Conference on Fifth Generation Computer Systems*, 6-Nov-84, pages 589 to 595. As an alternative to the microword-packing phase of microcode compilers, potential parallelism is manipulated using techniques described in Chapter VI. This work on parallelism is in part based on "Heuristics for the Global Optimization of Microprograms," by M. D. Poe, *Proceedings of the 13th Annual Microprogramming Workshop*, ACM, 30-Nov-80, pages 13 to 22, and "Measurement and Manipulation of Potential

Parallelism in Microcode," by M. D. Poe, 8-Sep-81, Euromicro81, which is published in *Implementing Functions*, edited by L. Richer, P. Le Beux, G. Chroust, and G. Noguez, pages 351 to 360. The implementation of UMS is written in the Prolog language, using techniques reviewed in part in "A KWIC (Key Word In Context) Bibliography on PROLOG and Logic Programming" by M. D. Poe, R. Nasr, J. Potter, and J. Slinn, *Journal of Logic Programming*, Vol. 1, No. 1, 1-Jun-84, pages 81 to 142.

1.2 Instruction Set and Processor Evolution

Historically, microcode-development methodology became important only when complex instruction sets became common. This section discusses how complex instruction sets evolved and the techniques used for their implementation.

The instruction sets of the first electronic computers were relatively simple, usually based on a straightforward mapping between instructions and hardware. The instructions directed the hardware to perform tasks such as loading a value into the accumulator register, reading memory, adding, etc. The control units of these computers, were correspondingly simple; they received instructions from main memory and directly controlled the rest of the hardware.

Most early computers spent a great deal of time in (machine language) subroutines performing simple operations such as multiplication. Hardware could perform such tasks ten or more times faster than a corresponding machine language subroutine. Even greater speed often can be realized when hardware directly performs the inner loops of other common algorithms. These performance improvements motivated building machines with expanded instruction sets. The most common extensions of the instruction set were numerical, initially integer multiplication and division. Later, floating-point operations, parts of the operating system, and even movement and comparison of byte character-strings became part of instruction sets. Extensions to the instruction set similarly increased the complexity of the control unit. Today, although some computer architects advocate simple instruction sets (called the RISC approach, discussed later and in [WIL82]), most feel that complex instruction sets are necessary.

1.2.1 Unstructured CPU design

The initial ad hoc methods of designing computers with complex instruction sets were not reliable enough for timely and correct implementation. Errors were very difficult to detect and, when detected, would often delay project development. The control unit of such a computer's Central Processing Unit (CPU) receives a list of machine language instructions. Together, the control unit and the Arithmetic Logic Unit (ALU) execute these instructions.

Originally, individual transistors of the control unit were dedicated to some specific instruction in the instruction set. The resulting control unit was very complicated, with wires going in many different directions (hence the term random logic). This design style is sometimes used today, as in the control unit of the Zilog Z80 microprocessor. A small example of random logic can also be found [RUBI82] in the upper right-hand portion of the Digital J11 datapath chip ([RUBI82] page 134, see also [PAT83]). The visual complexity of these examples suggest how difficult making modifications would be.

Proponents of the random-logic style call the control unit "hardwired"; critics call it a "spaghetti design." Hardwired designs often contain many initial errors because it is difficult for the designer to keep in mind how each part may affect every other. Accordingly, correcting a problem is difficult in such a design. The random-logic style is often a poor choice for VLSI-based designs, because the cost of adding a wire or transistor can be enormous; sometimes an entire chip must be redesigned when an error is discovered.

1.2.2 Structured CPU design

To make the design of a control unit simpler and more orderly, Wilkes [WIL51] suggested that the structure of the control unit resemble the entire computer itself, almost a computer within a computer. In the spirit of Wilkes' suggestion, most modern CPUs contain a special memory (the microstore) where the algorithmic specification of the instruction set resides in program

form (also called microcode, microprogram, or firmware). These modern CPUs are complex enough to take up to tens of person-years to design, but can be modified relatively easily.

The microstore is an array of memory cells, with each consecutive cell labeled with a consecutive numeric address. Each memory cell, called a microword, can store many binary digits (or "bits"). The number of bits in a microword is known as its width. Neighboring bits of a microword, which are related in the portion of the hardware they control, are conceptually grouped together as a field. The activity performed by a particular microword-field value is called a microoperation. The contents of a cell at a particular microstore address, which contains the microword fields (or microoperations) that define the numeric value of the microword, is called a microinstruction.

The control unit (the microsequencer) operating on the microcode, the microstore, and the hardware under control is called collectively the microengine. A microinstruction controls the action of the microengine for one time step or cycle. Thus, the analogy of a computer within a computer is complete: A computer (the microengine) runs a program (the microcode) to simulate or emulate a different computer (the one described in the instruction set and seen by the machine language programmer).

Typically, a large array of memory is used to implement the microstore, as on the Motorola 68000 microprocessor ([STA83] page 343), the microstore

of the Digital J11 chip set ([RUBI82] page 136), or the Hewlett-Packard 32-bit microprocessor ([GUPT83] page 14). The microstore accounts for a large percentage of the hardware (measured by chip cost or number of transistors) in a microprogrammed control unit. Much of the complexity of the design (again measured by chip cost or number of transistors) arises from the large amount of information that describes different microinstructions (usually represented as numbers) in the microstore. A microprogram coding error in the operation of a computer instruction can usually be localized to a small number of microinstructions; this error can be changed easily during the design process. Microprogramming is a more cost-effective design methodology, since it is much easier to understand and correct an individual microinstruction (changing a few numbers) than to cope with the complexity, wires, and transistors of a hardwired control unit.

1.2.3 Microcoded CPU design methodology

Microcoded CPU designs are not completely optimized. Each microword field controls a part of the microengine. However, only a small number of the permutations of microword-value patterns are typically found in microprograms. Similarly, only a few of the possible letter-sequence combinations are found in an English dictionary. If the permutations actually used are described by consecutive numbers, then a much smaller maximum number is needed. A small-sized maximum number requires description in a smaller quantity of bits

(and memory) and allows using a narrower microword. For most microprograms, the information thus can be represented in a smaller number of bits, resulting in a narrow microstore (i.e., one that is smaller and cheaper to build).

During computer design optimization, the width of the microword is sometimes decreased past the point where all the desired control values may be represented. If a required action cannot be represented in one microword, then multiple microword sequences with the equivalent semantics are used. In this case, the microprogram's width is decreased at the expense of creating a longer, and slower, microprogram. The computer designer can make tradeoffs between width of the microstore and length of the microprogram. By necessity, microstore memory must operate at very high speed and is therefore quite expensive. Because a longer microprogram (operating with a narrower microstore) runs slower than a shorter one (it takes more cycles to complete), speed and cost are interrelated. Minimizing the number of bits per microword usually accounts for a large part of the microengine design effort (see [SCHW68], [GRAM70], [DASBC73], [MONT74], [ROBE79], [BAEK79], [NAGLE81]). Minimizing width is complicated when many parts of the microengine perform operations in parallel (discussed in more detail later).

The microstores for the early computers ([WILS53], in [SIEBN82]) and some of the latest 8-bit microcomputers are quite small, with only a few hundred

microinstructions. A more contemporary example is the Hewlett-Packard HP9800, built in the 1970s with a microstore of 256 words of 28 bits, for a total of 7168 bits [SIEBN82]. The microstore of the more sophisticated instruction set of the IBM 360 series contains about one quarter of a million bits (e.g., model 30 contains 4096 60-bit wide words of microstore for a total of 245,700 bits [WEB67], [AGRR76], [SIEBN82]), and members of the DEC VAX family almost twice that. The ability to automate microprogramming has thus become increasingly necessary.

1.2.4 Cost-effectiveness of large instruction sets

Researchers who had worked primarily in microcode methodology and microarchitecture design began in the early 1980s to question the cost-effectiveness of complex instruction sets implemented in microcoded hardware. These complex machine designs were compared to simple instruction sets implemented on high-speed hardware of simple design. This class of simple designs is called RISC (for Reduced Instruction-Set Computers) in contrast to conventional Complex Instruction-Set Computers (labeled CISC). David Patterson of the University of California at Berkeley is the principal leader in this movement (see [BARM86], [COLHJ83], [DITP80], [FIZFKLPPPSSV81], [FODVP82], [HANLMMP82], [LAR82], [PATP82], [PATD80], [PATS82], [PATS81], [RADI82] and see [CLAL82], [CLAS80] for a rebuttal). This section discusses this controversy.

1.2.4.1 Processing simple instructions. To understand the attractiveness of RISC, consider simple operations such as integer add or register move. These typically can be performed in only one microinstruction on a microcoded microengine. Performing the same semantics as interpreted machine instructions (by the microcode) on the same microengine would take perhaps tenfold more time. The following example describes why simple microprogrammed instructions can be so slow.

In contrast to a RISC, a CISC must perform a series of microinstructions to execute each machine instruction. Figure 1.2 shows the controlflow for microcode emulation of instructions in a CISC. First, the number representing the instruction is fetched from memory. Based on the type of operation code and the source of each operand, the second box performs multiple-way branches to fetch all the required operands using boxes of type three. A multiple-way branch statement in box four, controlled by the instruction number, is executed to begin the instruction execution shown in type five boxes. Box six, another multiple-way branch, stores all results using code in type seven boxes. Box eight checks any are pending interrupts and processes them. This series of steps is called the instruction fetch-execute loop. The architecture of the microcoded fetch-execute loop allows fetching a larger number of operands in very flexible ways at a cost of some execution speed.

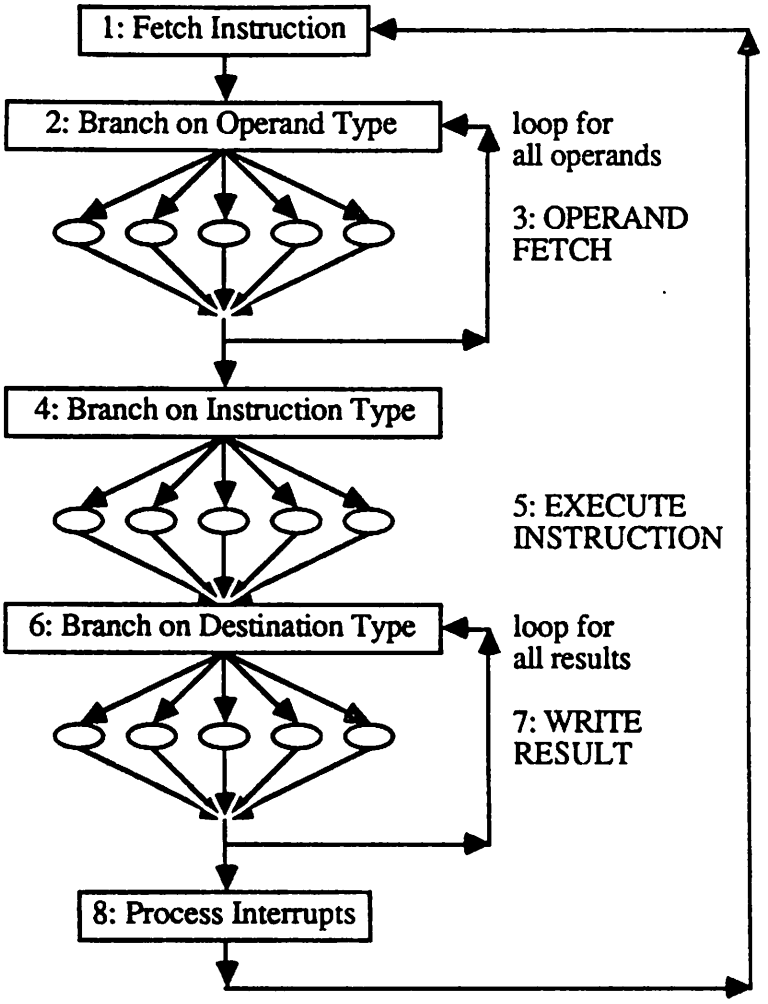


Figure 1.2: Controlflow structure of a typical microcoded emulator. This shows the flow for a CISC (Complex Instruction Set Computer). The outer-most path is called the fetch-execute loop.

Performing a machine instruction with simple semantics, such as "add two registers," would require a long sequence of microinstructions as an emulated machine instruction in a CISC. Addition of two registers is typically performed in one step within a RISC design. The difference in speed is due not to the inherent addition speed of the hardware, but to the architecture of the RISC or CISC instruction set and the way the hardware was structured to run it. The

equivalent semantics of this simple add instruction, in many cases, could be achieved in a single microinstruction of a CISC microengine if the microcode architecture of Figure 1.2 did not need to be adhered to. The RISC architecture performs operand fetch and write with fewer operands in a less flexible but faster way. Thus, a microcode implementation of an instruction set can complicate what was originally a very straightforward mapping between the instructions of the user program and the activity of the microengine. Sometimes the CISC design style incurs up to a tenfold speed penalty for processing simple instructions. However, more complex instructions, such as floating-point operations or character-string searches, cannot be performed within the single fetch-execute cycle of a RISC-type architecture and require costly sequences of instructions. Where tradeoffs should be made for faster execution of either simple or complex instructions is still controversial.

1.2.4.2 Hardware speed improvements. After a basic microengine has been designed, extra transistors or VLSI area can be used to speed up commonly used simple instructions or to run specialized instructions more quickly. Simple instructions such as integer addition can be speeded up with carry-lookahead hardware. Complex instructions such as character-string manipulation can be speeded up with special datapaths to extract characters from a word.

The RISC advocate would argue that speeding up special instructions is not the most efficient way to optimize computer performance. For a fixed number of transistors, a CISC microengine sacrifices speed of simple operations. This tradeoff is documented in [CLAL82] and [FIZFKLPPPSSV81]. Proponents of CISC argue that this is compensated for by the speed of complex microcoded instructions, such as floating-point arithmetic, or the programming simplicity of a virtual-memory environment [BELLMM78], [LEVYE80], [ORGA83]. Furthermore, complex instruction sets such as for the VAX are more highly encoded [PATS82], [FIZFKLPPPSSV81] so that their object programs require less space. The increased code density usually found in CISC designs is increasingly important in virtual-memory systems [ABUKL79], where page faults can drastically impair performance.

For general-purpose machines that run many high-level languages, a CISC may be the most effective design style because different high-level languages use very different instruction mixes [CLAS80]. For example, FORTRAN typically performs a large number of floating-point operations while COBOL mainly uses the packed-decimal format for numbers. A CISC often contains efficient implementations of the most commonly used instructions for different languages.

1.2.4.3 Computer family design techniques. Whereas the RISC approach is very closely tied to gate (the basic electronic circuit) speed, the

CISC machine potentially can exploit single or multiple ALUs, datapaths, shifters, and other special logic. Further, a CISC approach allows microengine builders design flexibility in the horizontal-vertical microcode dimension (described in the next section). These design parameters allow building CPUs (which can run exactly the same software) with a wider performance range than with the RISC approach. Families of CPUs that run the same object code preserve a user's investment in software, which is often the most expensive part of a computer system. When the performance range of a computer family is wide, it is more likely that a member of a CISC family will be available at the desired cost.

In contrast, some of the newer system programming languages, such as C, allow easier retargeting of software for new RISC architectures than was previously possible. Such software portability, when truly successful, eliminates the need for members of a computer family to run the same object code. However, the range of family member costs, pipelining, and code density remain for RISC if the code-retargeting problem alone is solved.

Although the debate still rages about where tradeoffs between complex microcoded instruction sets and simple instruction sets should be made [WIL82], the majority of current computer architectures are complex enough to benefit from microprogramming. Many major computer manufacturers are

developing both RISC and CISC products, and the controversy will probably continue into the next decade.

1.2.5 Microcoded CPU styles

Originally, microcoded computers employed a microengine in which only one hardware unit performing simple or low-level semantics was active at any given time. This design is called a vertical microengine because the microprogram is a sequential, vertical listing of activities for the units. Later designs employ horizontal microengines in which many hardware units operate simultaneously. In horizontal microcode, words of the microstore usually contain fields associated with different microoperations. Typically, these simultaneously performed microoperations are printed horizontally on a line of the microprogram listing. The horizontal design approach attempts to take advantage of the potential parallelism found in algorithms that implement the instruction set.

Keeping track of simultaneous operations in a microengine makes horizontal microprogramming much harder than vertical microcoding. Even if the parallelism analysis is automated, its computation is quite expensive (see [ULL73], [COFF76]). Some RISC advocates argue that this type of parallelism is not worth the design effort and that alternative design techniques could achieve the desired performance. Other researchers argue that parallelism is not present in the first place. The automation of the microcode-generation

process described in this thesis hopefully will encourage more design and exploitation of parallelism in hardware than has been employed with manual methods.

Existing horizontal microarchitectures prove it is practical to execute many subtasks in parallel when the instruction set reflects complex operating system functions or high-level language constructs. The use of parallelism further biases the design tradeoffs away from simple instruction sets and microengines toward complex ones. Since these design tradeoffs are still very controversial, it is likely that the future will see many examples of CISC with both vertical and horizontal microprogramming.

1.3 Comparison of Synthesis with Hand Coding and Compilation

This section introduces many of the UMS techniques described in the rest of this thesis. To show how conventional microcode-development methodology constrains microcode quality, this section compares a small sample of microcode synthesis performed by UMS to hand-written and compiled microcode. The discussion in this section is confined to describing what UMS does; later chapters describe the technology in more detail.

A continuum exists between synthesis and (table-driven) compilation. Synthesis deals with analysis, creation, and then the use of a mapping

between the hardware and the requirements of the instruction-set specification. Table-driven (or production-system-driven) compilation deals with using the mapping from such an analysis, as represented in tables, to compile an instruction-set specification. Synthesis potentially recreates this mapping for every piece of microcode; compilation repeatedly applies a previously determined mapping to each similar source-code fragment. If the synthesis stores previous mappings, then compilation is taking place when these maps are reused.

The computational machinery that synthesizes microcode in UMS would be called a synthesizer if used alone or a production generator if used with the table-driven compiler. The current implementation of UMS does not process controlflow or catalog alternative translations in RUT form. Previous work by this author ([PATGPS81], [POE80], [POE81A]) discussed controlflow processing techniques for horizontal microengines. The details of integrating a synthesizer with an existing microcode compiler are beyond the scope of this thesis.

Very little human interaction is required to run UMS. For microcode synthesis, an expert need not reconfigure UMS for a particular microengine. In fact, UMS is ideal for experimenting with microarchitectures, expressed by different microengine descriptions, and measuring the resulting code quality. An expert's role is only to add rules to the microcode knowledge base. Previous results with expert systems indicate that, as the system is used for a

larger number of examples, the number of new rules needed tapers off (see [FRASER77] page 40, [BARS79] pages 225 to 229, or [BARS85A]). Therefore, it is expected, but not yet shown, that only a few rules will need to be added to UMS for each new microengine.

The only information given to UMS when it begins work is a list of pointers to the beginning of the procedural hardware specification and the procedural instruction-set specification. Since each specification is composed of many modules and subroutines, one module or subroutine must be specified as the main body of code. This is equivalent to where a "call" to either of these behavioral descriptions (or simulations) would begin execution. This situation, in which a small initial amount of information about the microengine and microprogram used by UMS is available, is similar to that facing a human microprogrammer on the first day of work; the programmer can be expected to absorb only a limited volume of new information.

The sample code shown in the top of Figure 1.3 is one source line (written in the ISPS language) from an instruction-set description for the PDP-8. ISPS ([BARN78], [BAR79B], [BAR79C], [BARBCS78], [BAR79], [PARTCC79], [BARS82]) is a general-purpose programming language often used for hardware and instruction-set descriptions (also see [LIP78]). Figure 1.3 also shows the internal UMS data structure, consisting of directed-acyclic graphs (DAG) composed of arcs and nodes, which correspond to the ISPS code.

ISPS source code:

cpage = PC<0:4> next

corresponding arc and node structure:

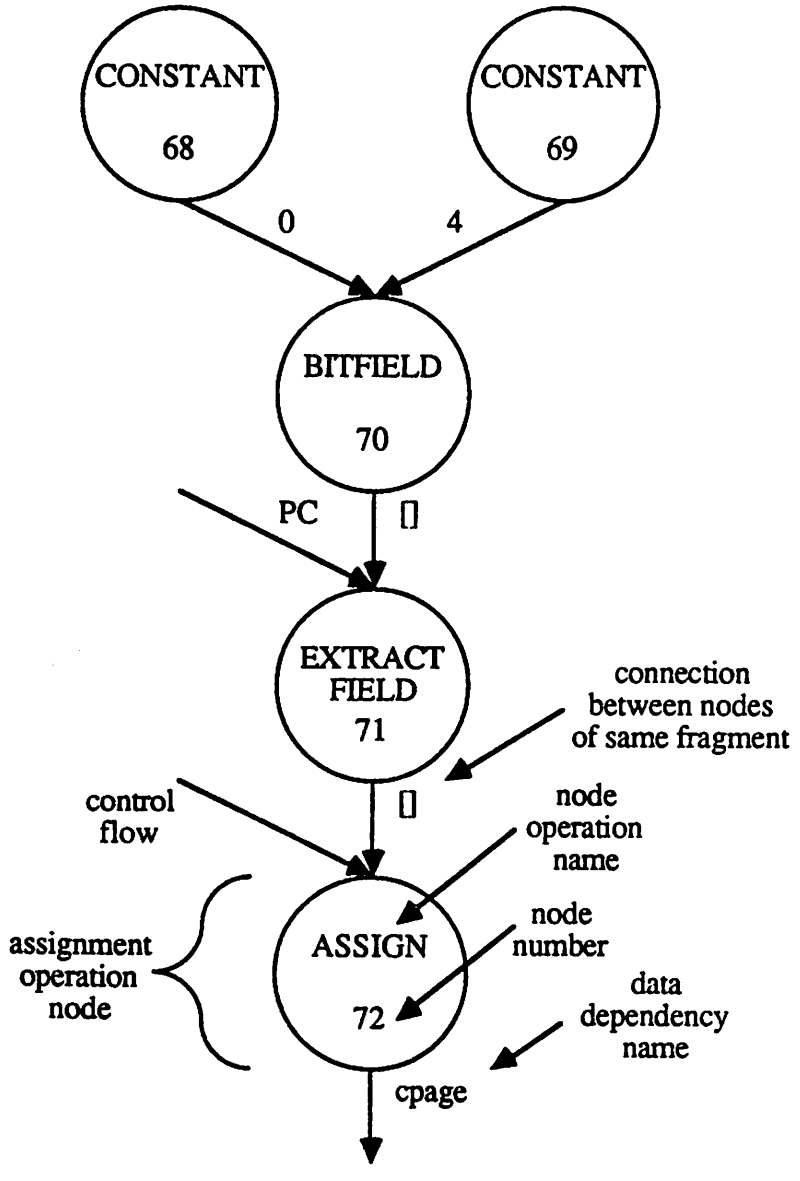


Figure 1.3: ISPS source code (top) compared to a Directed Acyclic Graph (DAG) representation (bottom). The DAG consists of arcs and nodes.

The PDP-8 has an unusual type of addressing technique (for the direct-addressing mode) in which the high-order bits of the program counter (representing the current memory page of the current instruction) are also used as the high-order bits of the operand address. The current 128-word page is determined in the code of Figure 1.3 when the extraction of the high bits of the program counter sets the variable "cpage".

The sample code is part of the effective address calculation (and is also shown in Figure A.3). It represents a portion of the fetch half of the instruction fetch-execute loop (see Figure 1.2). The sample code in Figure 1.3 reads the five high-order bits of the variable "PC" (which stands for program counter) and writes them into the variable named "cpage" (which stands for current memory page). In a line of code is shown here, the variable "PC" is defined as 12 bits wide, where the high-order bit is numbered 0, the low-order bit 11. The ISPS statement separator ";" (not shown in Figure 1.3) indicates which statements may execute in parallel while the keyword "next" indicates serial execution. It is similar to that found in [SIEBN82] page 125, [BAR79] and [BARS82]. The example target hardware ([SIEBN82] page 219) used here is based on AM2901 ALU bitslice chips [AMD81]. Chapter VII uses the same sample instruction set and target hardware.

1.3.1 Hand-written microassembler coding

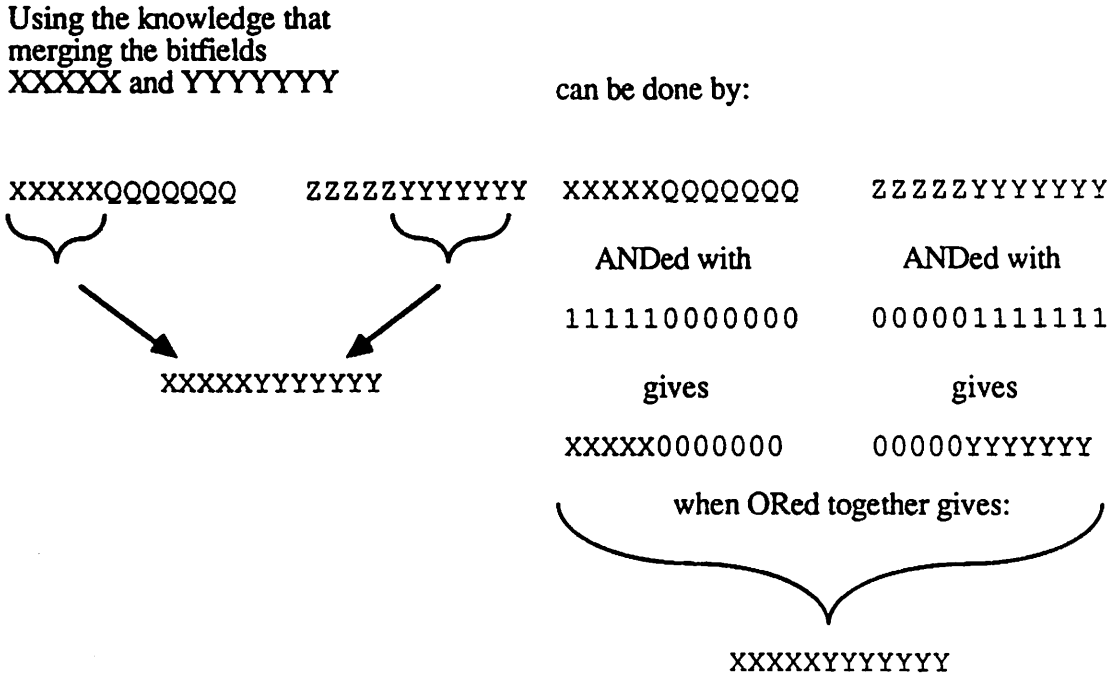
This section describes the programming effort for hand coding the sample in a microassembler. A microprogrammer's first task is to become familiar with the microengine. Minimization of the microstore width has made microoperation encoding complex. Usually, the microengine is programmed with a microassembler language, which often provides a macro facility (for using abbreviations or a type of shorthand). Even if a macro facility is used, many fields must be individually specified. A typical macro might replace "LOAD A 92" (meaning load register A with the contents of working register 92) with a long sequence of microengine control field specifications. A macro may determine the value of many microword fields and specify the routing (movement) of data through a complex sequence of hardware gates. The microprogrammer may require many weeks to become familiar with the microengine and macros used to control it.

The sample line of code "cpage = PC<0:4>" stores the five high-order bits (labeled 0 to 4 or <0:4>) of the program counter "PC" in a new variable "cpage". The microprogrammer's previous register allocation determines the location of the "PC" variable. The microprogrammer searches the hardware description to see how this register may be read by the microengine and how the five-bit wide field may be directly extracted. Unfortunately, there is no hardware, such as a special multiplexer, that extracts this field.

After searching the description of the instruction set, the programmer may notice that only the "eadd" (effective address calculation) subroutine uses the variable "cpage". Here, the high-order bits are concatenated to some low-order bits to form an effective address. Now a programming trick can be used. This may not be appropriate if the "cpage" high-order bits were to be used as the low-order bits in some intermediate calculation. When two bitfields that are in the correct positions in their respective operands are concatenated to form a longer field, the two bitfields are "AND-ed" to clear their unimportant bits, then "OR-ed" together to merge the bitfields. Figure 1.4 shows this technique. This techniques is exactly what microprogrammer Michael Tsao does ([SIEBN82] page 226, microwords 020, 021 and 022). It will be shown later how UMS automatically tries this programming trick with ANDs and ORs. These logical operations avoid (the often computationally expensive) shifting of the bitfield to align it differently within a data word, because the alignment does not need to be changed.

The microprogrammer then determines the final translation of the line of code into microcode. The microprogrammer tries to determine how they can be most efficiently placed into microwords. The order they were found in the source code is most convenient in this example. Note that the microprogrammer must have made appropriate register allocations and be able

to notice the applicability of the programming trick to achieve the code quality described here.



the synthesizer can transform the code:

```
cpage = PC<0:4> next
MA<0:4> = cpage
```

into:

```
A = (#7600 and PC);
B = (#0177 and MA) next
MA = A or B;
```

Figure 1.4: A source-code transformation. PC, MA, A, and B are all 12 bit wide words, and the constants #0177 and #7600 are represented in octal.

1.3.2 Microcode compiler coding

Microcode can be created using a compiler. Originally, this thesis was conceived of as working with a complete microcode-compilation system called the V-Compiler, which the author partly designed and implemented [PATGPS81] at Digital Equipment Corporation. However, when the project was canceled, the author had to implement much of this software to show the capabilities of the synthesizer described in this thesis. The following example describes the use of the V-Compiler [PATGPS81]. In a microcode compiler, much of the analysis of the microengine is performed before the compiler is ever run. This customization of the compiler for a particular microengine is performed by the author of the compiler.

In this example, the variables "cpage" and "PC" (the current user-memory page and program counter) typically would be defined for compilation as particular pieces of the hardware state. The definitions are located in the tables, or productions, of the compiler either implicitly (as in the code-generation productions for "cpage") or explicitly (as in the V-Compiler external allocator for "PC") or even in the source code itself. A compiler expert writes these definitions. Since the variable "cpage" is used across subroutine call boundaries, only the most sophisticated compiler could deduce that it is read only once. Without this insight, "cpage" must be categorized as a global

state variable (a variable that retains its value between passes of the fetch-execute loop).

The bitfield-extraction phrase "<0:4>" triggers special code-sequence generation during table look-up. No attempt is made by the compiler to understand the context in which the phrase is used. To place the required value in the "cpage" register, the generated code clears the unimportant bits, performs an alignment shift, and then does a store. When the value of "cpage" is used later as high-order address bits, it will be shifted again to its original alignment in the word.

This example shows that microcode compilation, particularly in the code-generation phase, is mainly a table look-up process. Since this process does not take into account much of the context for each microprogram fragment, quality is impaired. The example showed that without some data-dependency information, conservative code generation was required for use of the variable "cpage". The compiler-generated code performs unnecessary shifts and masking operations.

1.3.3 Microcode synthesis

The following section describes the UMS microcode synthesis approach, which could generate the information for a microcode compiler's tables. Typically, a compiler considers only one alternative for each code fragment; this alternative is the one represented within its look-up table. However, UMS

considers many code-generation alternatives, as is discussed below. Thus, the quality of synthesized code is potentially much higher than that of compiled code.

1.3.3.1 Hardware description techniques. The hardware description for UMS is written in procedural form (which would allow it to be used as a hardware simulation) using ISPS language. A temporary variable represents a bus or wire that transfers a value from one part of the hardware to another. A variable that could be read at the beginning of a microengine fetch-execute cycle and written at the end of the cycle represents a register. A line of code with a plus sign in it represents an adder. A multiway branch or case statement represents a complex ALU. Each branch alternative describes a different function type, such as addition or subtraction. A small bitfield of the microinstruction buffer usually determines (is connected to) the index of the case statement. This buffer would in turn receive its value from (or be connected to) a code fragment that reads an element of the microstore array (containing the microprogram). The array element accessed by the microstore array read would be determined by the variable representing the microprogram counter.

1.3.3.2 Data structures. Typical source code describes concisely what operations should be performed when. However, such source code does not describe concisely where data values come from, where values are later

used (recall the use of the "cpage" value across subroutine boundaries, described earlier), or what conditions invoke code fragment execution (a subroutine does not necessarily know where it was called from). Typical source code only represents implicitly all this information. Since UMS bases much of its activity on these control and data dependencies, for efficient synthesis they are described explicitly in an internal graph-style representation (see Figure 1.3 and [POE84]). Otherwise, search would be required to infer this information repeatedly from the source code every time it was needed during synthesis.

One part of UMS is a compiler that translates ISPS source code into a graph form. A node of the graph represents an individual operation (such as addition or assignment) of the source code. Each arc indicates the local source and destination of a control or data dependency (such as a value that comes from an input parameter to a subroutine). This type of code representation is often found in the optimization phases of compilers (see [ANKCHM82], [AHOSU86], [AHOU72], [AHOU77], [SET75], and [WULJWHG75] or discussion).

A second part of UMS, called a postprocessor, annotates the graph form with additional global control-and-data-dependency information (such a subroutine input value that actually comes from a specific calculation before the subroutine call). Other often needed information is also made explicit by adding attributes to some graph arcs, such as detailed descriptions of the

variable characteristics (i.e., number of bits, their numeric names, etc.). Such information is usually described only in variable definition statements and is not locally present in source code.

UMS does not depend on ISPS programmer-provided sequencing constraints such as ";" and "next". Instead, it conservatively interprets each statement as if it had a "next" statement at its boundaries. Analysis of control and data dependencies by the postprocessor then determines if any parallelism is present.

1.3.3.3 General synthesis process. The graph representing the instruction-set specification is modified extensively during synthesis by the UMS production system. The goal of the modification is to create a mapping (or binding) between each piece of the instruction-set graph and a part of the hardware description, which is also in graph form. The graph representing the hardware is never altered. In general, analysis of the hardware description guides instruction-set description modifications. However, when the mapping is not directly successful, the instruction-set graph is sometimes analyzed to determine where it should be changed to make it closer in structure to the hardware graph.

Implicit in the microengine hardware description is a fetch-execute loop for microinstructions; in this example, the loop is conceptually simpler than the one for the instruction-set description shown in Figure 1.2. As the

instruction-set graph is modified, it is mapped to the fetch-execute loop of the microengine, which is discovered within the hardware description. A many-to-one mapping of the hardware fetch-execute loop to the instruction-set fetch-execute loop is required because multiple microinstructions are usually needed to perform an instruction of the instruction set (see Figure 1.2), and all instructions of the instruction set must be represented in microcode.

As a consequence of the many-to-one mapping, most features of the microengine are used at least once by the synthesized microcode. This is expressed by mapping most features of the microengine graph to a part of the instruction-set description graph. Correspondingly, a part of the hardware description is mapped to many parts of the instruction description over many hardware fetch-execute loop cycles. Each of these mappings represents a reuse of the same piece of hardware, but at different instances of time. Multiple use of the same piece of hardware at the same instant of time can be prevented by use of the RUT. The RUT analysis can identify constraints of the microengine and illuminate potential parallelism in the code. The instruction-set graph modification is guided by incrementally more complex code-generation attempts based on transformations, which rely on analysis of the hardware description.

The challenge of the UMS synthesis approach is to modify and match parts of an instruction-set description to these hardware code fragments. Some

of UMS's modifications simply rewrite the instruction set into a series of intermediate forms while others modify these forms to adapt them to the target hardware. The results of modifications performed previously are intended to be stored in a catalog, in part using an unimplemented data structure called a RUT. Although currently unimplemented in UMS, the RUT is intended to allow efficient reuse of previous synthesis results. To exploit programming tricks, the production system accepts advice in the form of rules. These rules represent some of the experience and insight of experienced microprogrammers. An individual rule would describe the programming trick used in Figure 1.4.

1.3.3.4 Dataflow synthesis example. The same sample line of code discussed earlier ("cpage = PC<0:4>") requires bitfield extraction followed by an assignment. UMS searches first for a bitfield-extraction mechanism in the microengine description. None is found, and UMS determines that the bitfield-extraction portion of the expression causes search failure. The knowledge base of program-transformation rules is searched exhaustively for anything that involves bitfield extraction. The knowledge-base rules are ordered in increasing generality, so that UMS attempts to use first the most specific special cases. In this example, the first special case used by UMS is shown in Figure 1.4. This transformation rule describes bitfield extraction from a variable, followed by use of the extracted value with its original alignment within a word.

The transformation shown in Figure 1.4 creates new temporary variables A and B of the instruction-set description. At this point in the synthesis example, the code has not been mapped into the facilities of the microengine but merely changed into something that might be successfully mapped. Transformations such as this one (shown in detail in Figures A.9, A.10, A.11, and A.12) are easily written for UMS and represent the human microcoder's knowledge of programming tricks. Transformations may have arbitrarily complex preconditions and are represented as "before" and "after" descriptions of the graph. The rule precondition of Figure 1.4 requires that the bitfields XXXXX and YYYYYYY remain in their original alignment within the data word.

At this point, UMS has generated a collection of subproblems that must be solved before synthesis can proceed to the remaining parts of the instruction set. These subproblems are performing the two ANDs, the OR, and generating the needed mask constants. As before, the synthesis begins by searching the microengine-description operators before considering the operands. The initial search determines where to perform the "AND" operation of "A = (#7600 and PC)". The ALU can perform this function. When the desired "AND" operation is found in the microengine, the hardware operands of that operation are inspected to determine if the necessary values can be routed to them. Unfortunately, UMS cannot route the operand "PC" (which at this point is mapped to the Memory Address Register "MAR" of the hardware) to the ALU

(Chapter VII describes this scenario in more detail). A search through the hardware description for alternate locations to perform the "AND" also fails. So that the ALU can perform the needed "AND" function, it thus seems necessary to modify the synthesis so far achieved.

1.3.3.5 Variable-remapping example. Most of the activity of UMS is heuristic-guided manipulation of the instruction-set description graph, as constrained by the hardware-description graph. One of the most commonly used heuristics is called "variable remapping." This heuristic deletes a previous binding between an instruction-set variable and a piece of hardware, determines alternative bindings based on how information is routed through the hardware, and then records a newly chosen binding. During variable remapping, other microcode segments, which used the previous binding, are also modified to reflect these changes.

In a conventional microcode compiler, intermediate temporary variables may be mapped to hardware with register-allocation algorithms. However, changing the mapping of important instruction-set state variables to hardware components would be performed manually by the compiler user. If this mapping was incorrectly specified, then the compiler does not perform satisfactorily. In contrast, UMS performs this activity automatically for both types of variables.

The creation of the variable-remapping heuristic was motivated by the observation that a variable within the hardware (representing either the value on a bus or part of the hardware state) may be used for multiple purposes. If a variable is mapped to a piece of the hardware with a very specific use, then the mapping may constrain the use of the hardware too much. This problem exists in the example of the mapping between "PC" and "MAR". If the variable is alternatively mapped to a different piece of hardware, which can move the value to the original location on demand, then the same functionality exists, but the frequently needed hardware may now be considered for other uses. Because routing variable values is the most frequent and critical operation of UMS, the variable-remapping heuristic is the instruction-set modification technique tried first.

In the example, variable remapping allows the ALU to perform the required "AND" operation on the value of "PC". Initially, the "MAR" hardware resource is used for only one purpose to hold the value of the program counter "PC" in the synthesis example. This is shown in the data-dependency and variable map of Figure 1.5 by writing "PC" after "MAR" and separating the two by a colon.

MAR : PC

is modified to become:

ALU.out : PC

↓

MAR

and later modified to become:

2

↓

A

↓

RAM[A] : PC

↓

A.LATCH

↓

Y

↓

out

↓

AM2901

↓

Temp.AM2901

↓

ALU.out

↓

MAR

Figure 1.5: An example of variable remapping. Initially, the instruction-set variable "PC" is mapped to the hardware variable "MAR", and is shown here as the two variables being on the same line and separated by a colon. Later synthesis constraints require modifying this previous result by mapping "PC" to a part of the hardware "upstream" from "MAR". As further synthesis constraints are discovered, "PC" is bound to RAM[2] (see text).

However, this uses the MAR too restrictively. The hardware cannot move the instruction-set description value "PC" into the ALU when it is mapped to the hardware variable "MAR". The hardware description is searched along the DAG arcs, which represent hardware buses, for a variable whose value may flow to MAR through a sequence of assignment statements. The variable "ALU.out" (which is the output of the ALU) is found to meet this constraint. The instruction-set fragment "i = M[PC]" was bound previously by UMS to the hardware fragment "MBR = MP[MAR]". This instruction-set fragment is now transformed and resynthesized into "MAR = ALU.out, MBR = MP[MAR]". The variable mapping is then changed ("remapped") to reflect that the instruction variable "PC" is now mapped to the hardware variable "ALU.out". Figure 1.5 shows this as "PC" located next to "ALU.out". The data-dependency arcs flow downward in the figure so that "MAR" receives a value from "ALU.out" (this can be represented as "MAR = ALU.out"). At this point, the hardware variable "MAR" is no longer mapped and may therefore be used elsewhere by the microcode. Fortunately, the movement of "PC" to "MAR" can still occur within one microword, although the variable-remapping heuristic does not require this.

Now that the subproblem of moving the map of "PC" to a different part of the hardware seems to be solved, the problem of synthesizing "A = #7600 and PC" resumes. However, the search for a second hardware-variable read of the

value "PC" (now newly mapped to the hardware entity "ALU.out") still fails. The process of remapping the variable "PC" is performed repeatedly until the value of "PC" can be routed to the input of the ALU (see Figure 1.5) so that the "AND" may be performed. UMS does not stop remapping with either variable "A.LATCH" or "Y" of Figure 1.5 because they are eventually discovered to be just temporary variables or wires. Since "PC" is determined to be a state variable of the instruction set, it must be mapped to a hardware-state variable such as RAM. The manner in which state variables are detected in ISPS code is described in Chapter IV.

In the variable remapping just described, the routing test succeeds when "PC" is mapped to the memory array "RAM" because "RAM" can be input to the ALU. Additionally a note is made to indicate that the ALU using these variables performs this "AND" operation. To make the comparison with the hand-written microcode of [SIEBN82] easier, the second element of the "RAM" memory array is chosen to contain the value of "PC" (this is the choice made in the hand-written microcode). At last both constraints for read access to "PC" have been met; "PC" can now be used as an index, as required by a previous instruction-set description expression "i = M[PC]" (otherwise not discussed here) and as an ALU input, as in the expression "A = #7600 and PC".

The other input operand is the constant bit pattern "#7600". Search is performed to determine if any part of the microengine already contains this

pattern. The search fails, and the microengine is searched to see if any variable input to the ALU has "read-only" characteristics (this search is prompted by the observation that the operand is a constant). The constant array "mask" is both easily routable and has read-only characteristics. To ease the comparison of this synthesis example, the fifth element of the "mask" memory array is chosen to contain the value of "#7600" (see the left side of Figure 1.6), so that it matches the array indices of an example [SIEBN82] of hand-written microcode. The dataflow synthesis just described is representative of the reasoning processes used in the current UMS implementation.

1.3.3.6 Controlflow-synthesis example. Up to this point, this discussion has considered only the data dependencies of microengine operations. For the instruction-set fragment "A=#7600 and PC", these dependencies are represented as the graph on the left-hand side of Figure 1.6. The individual statements of the hardware description that make up this graph are shown in the middle of the figure. Most parts of the hardware description have control dependencies, which in turn typically require specific microword-field values. In the hardware description, the value of a microword field is usually the index value of a "case statement" that determines how the microengine behaves. Those microword-field values are shown in the right-hand side of Figure 1.6, adjacent to their corresponding hardware-

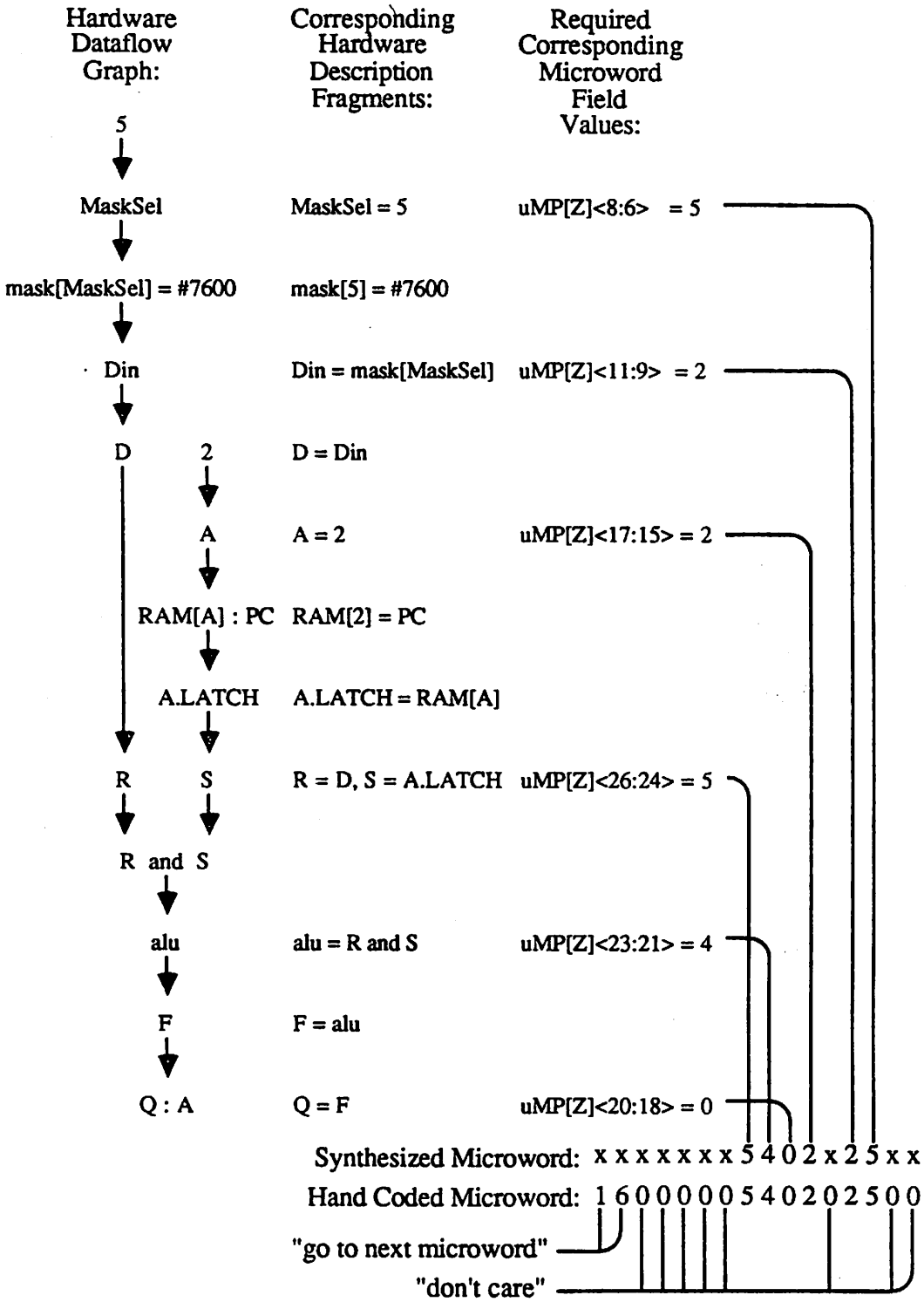


Figure 1.6: Hardware data-dependencies and control constraints in an example of synthesizing microcode for the ISPS instruction-set description phrase "A = #7600 and PC". "x" means don't care in the synthesized microword (see text).

description fragments in the middle of the figure. These microword-field values are also shown in the bottom of Figure 1.6, and compared with a hand-written microword. Also shown in this synthesized microword are irrelevant labeled fields, which control parts of the microengine that do not affect the desired computations (shown as "don't care"), or trivial fields, which control the simple data dependencies for the microsequencer to automatically step to the next microword. UMS does not currently display these associated microword-field values during synthesis, although it would be simple to do so.

As each decision is made about the dataflow part of synthesis, the values of the control part of the microengine description should be checked for inconsistencies (such as asking a simple ALU to perform addition and subtraction simultaneously). This inconsistency checking is not microword packing. Since a microword field may determine the behavior of more than one part of the hardware, this check merely determines whether the microword choice precludes completing the intended operation (such as a microword-field value that prevents routing an addition result to a register for storage). The inconsistency check is currently not performed by UMS, but it too could be easily implemented by tracing through the control dependencies of the DAG.

1.3.3.7 Variability in microcode generation. For horizontal microengines, much work remains to be done after UMS completes the initial translation of each microcode fragment to microcode. A translation typically

represents one of a large class of alternative translations, called a variation set. This section describes one variation set of these microcode translations. The next section describes how to select one of these translations for the final microcode.

Three statements from the instruction-set description, "A = (#7600 and PC); B = (#0177 and MA) next MA = A or B", are used to show variability of microcode translation. This code is a translation of the statements "cpage = PC<0:4> next MA<0:4> = cpage" used in the earlier example. The ALU could calculate the value of "A" in two ways: "PC and #7600" or as "#7600 and PC". Similarly, the "A" instruction-set register could be mapped to either the Q register or a member of the RAM registers (a target microarchitecture for this example is shown in Figure 7.2). Combined, these two choices allow microcode generation in four alternative ways.

Although Figure 1.7 shows 64 microcode-generation alternatives for this simple example, typical microengines would allow for many more variations. In particular, if different ways to move values through the hardware were considered, the number of permutations would be much greater. The careful choice between alternatives may allow some neighboring code fragments to share the microwords of this fragment. A microcode synthesizer must consider each of these alternatives in the context of the neighboring microcode to ensure

Code:	Semantic Variations:
A = (#7600 and PC);	2 variations for operand order, 2 variations for Q or RAM register destination
B = (#0177 and MA) next	2 variations for operand order, 2 variations for Q or RAM register destination
MA = A or B;	2 variations for operand order, 2 variations for execution order (calculate A or B first)

Figure 1.7: Microcode-synthesis variation (64 permutations) in example microcode.

maximum microcode compactness and efficiency. Currently, UMS chooses only the first alternative that fits the current context.

1.3.3.8 Use of the catalog. As UMS generates microcode for a new microengine, it stores the results of the code-generation experience in a catalog. This catalog is initially empty. After a fragment type has been synthesized the first time, the computational efficiency of the synthesis process can approach that of a conventional microcode compiler when reusing the results of this previous work.

The catalog describes successful transformations of the ISPS instruction-set specification language to microcode. Further, the choice between many alternative translations in the catalog generates potentially superior microcode. Although not all alternatives are described, those that show major differences in hardware use or routing are included. For example, multiplication, addition, and shifting would be under the catalog entry of "integer multiplication by two." The contents of this catalog also represent the

code-generation productions necessary for a retargetable-microcode compiler (such as the V-Compiler [PATGPS81]).

When implemented, the RUT (see Figure 1.8, [POE81B], and Chapter IV) could be used to describe the code fragments for efficient use with a catalog. The RUT is an inexpensive estimation technique designed to allow UMS to analyze the surrounding context for each fragment of microcode. The analysis determines which of many combinations of alternatives produce the most efficient microcode. For example, doubling the value of an integer may be performed in two ways, by adding the number to itself or by shifting it left one place (most microengines do not directly support multiplication). The two alternatives use different hardware resources: the adder or the shifter, respectively. In most microengines both of these hardware resources can be used within one microword.

Depending on what code has already been synthesized for a particular microword, UMS could use the RUT to choose a translation for a new code fragment that may fit in the same microword. Instead of exhaustively testing each permutation of alternatives, the RUT could efficiently prune the search space with a heuristic based on the data dependencies and hardware-resource use of the code fragment. Thus, what might in some cases take two microwords to perform can be optimized to take one microword (when both the adder and the shifter are used).

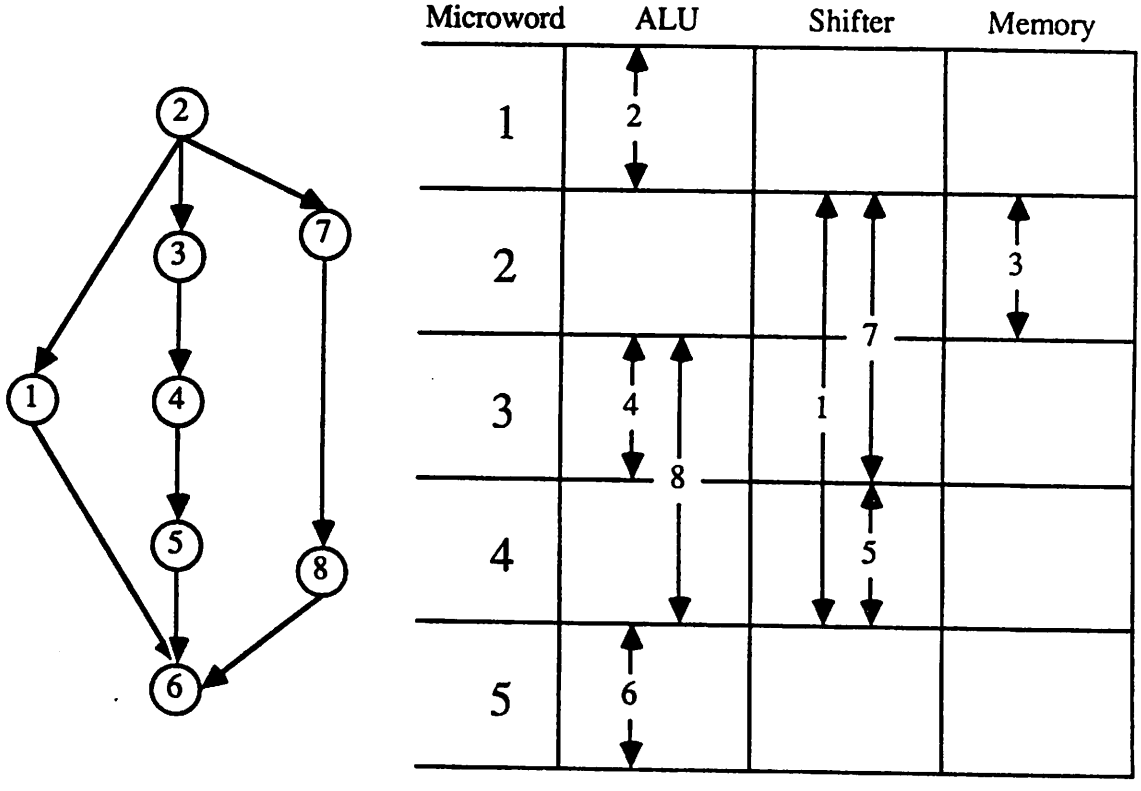


Figure 1.8: A data-dependency graph and a diagram of its RUT (Resource Utilization Template).

The RUT can also be used to determine how many microwords two microcode fragments require individually and how many microwords they require when merged or compacted together. The RUT is therefore a technique that analyzes how hardware-resource conflicts interact with microoperation-placement variability and data-dependency constraints.

To understand how the RUT works, consider Figure 1.8. Each numbered box on the left side of the figure represents a microoperation. The arrows on the left side of Figure 1.8 represent the data dependencies between microoperations; microoperation 1 reads a value that microoperation 2 writes.

The longest path through the graph is called the critical path. It determines the minimum number of levels needed to represent the hardware resources used in the graph (see the right side of Figure 1.8).

In a simple horizontal microengine, microoperations 2 through 6 would be placed in consecutive microwords. Due to its data dependencies, microoperation 1 may be placed in microwords 2 through 4 (see Figure 1.8). Microoperation 1 uses only the shifter of the microengine. The potential use of the shifter by microoperation 1 within microwords 2 through 4 is described as the vertical "1" double-headed arrow on the right-hand side. Potential use of other hardware resources by microoperations are similarly shown on the right-hand side of the figure. Note the interaction between data dependencies and resource use; because microoperation 5, which can be only in microword 4, uses the shifter, microoperation 1 is restricted to microwords 2 or 3. One possible grouping of microoperations into microwords as constrained by their hardware requirements is (2),(1,3),(4,7),(8,5),(6). When dozens of hardware resources are used by tens of microoperations, determining what is a legal grouping becomes very difficult. Using conventional techniques to question if one more microoperation may be added to an existing piece of microcode can be very expensive computationally. However, the unimplemented RUT described in Chapter IV is a computationally inexpensive way to answer these questions.

The heart of UMS, called the inference engine, is described in Figure 1.9. Controlflow determines the node-processing order of the inference engine. As shown in step 1, if any code fragments remain to be synthesized, one fragment is chosen. The choice is based on the top-down, left-to-right transversal of the graph (described in the previous section). In step 2, arcs are traced from this chosen fragment to determine a complete list of those nodes comprising the original fragment of source code. If the currently implemented synthesizer was part of a microcode-compilation system, then step 3 would determine whether a translation of the current source-code fragment to microcode was in a catalog of compilation techniques and simply apply the technique. Step 4 applies the power of the inference engine to the source-code fragment to generate microcode without changing its semantics. If this effort is unsuccessful, then step 5 determines in which node the problem is localized. Step 6 changes the semantics of the problematical part of the source code fragment so that it might be successfully synthesized and completes the main loop.

1.3.3.9 Microcode synthesis discussion. The actions of UMS are represented in many of the phases of the V-Compiler ([PATGPS81]), although in more sophisticated forms for synthesis. The external allocator of the V-Compiler sets aside space for global state variables as specified by the microprogrammer. In contrast, UMS deduces where state variables may be stored in the microengine. Further, it also decides where to store globally used

variables (such as the program counter and user registers). UMS has access to a general production system, which it uses to optimize the instruction-set specification, as does the machine-independent optimizer of the V-Compiler.

Synthesis Algorithm:

1) If a node remains to be synthesized
   then pick next node to synthesize; else end.
2) Expand node into microoperations that contain it.
3) If microoperation in catalog
   then compile it and go to 1.
4) If mapping is trivial
   then make map and go to 1.
5) Choose hardest node in microoperation to map.
6) Transform hardest node and go to 1.

Figure 1.9: Outline of synthesis algorithm.

The UMS productions form a knowledge base about microprogramming and contain programming rules and transformations (see [KIB78]). Unlike most other production systems that modify lists or strings, the one in UMS manipulates graphs of control and data dependencies. Also unlike production systems that perform syntactic transformations, the UMS production system can invoke arbitrarily complex subprograms for computational assistance. It is likely that more complicated microarchitectures and instruction sets are likely to increase synthesis times and/or the size of the required knowledge base. However, analysis of these knowledge bases for completeness and consistency, and developing proofs of design optimality, are still a research topic.

In the V-Compiler, creating microoperations and simple microcode optimizations are the activities of the code generator and machine-dependent optimizer phases. To finally bind microoperations into individual microwords, the V-Compiler performs register allocation (internal allocator) and microword packing (packer). Two unimplemented parts of the V-Compiler, the feedback analyzer and executive, were intended to initiate and control any small code-fragment reprocessing based on newly discovered contextual information. Code-fragment reprocessing prompted by contextual information such as register allocation regularly occurs in UMS. UMS keeps records showing which pieces of microcode are affected by different synthesis decisions. If it is later determined that a decision must be changed, only those microcode fragments directly affected are modified. Finally, the assembly function of the compiler is performed when values are determined for individual microword fields.

1.3.4 Comparative code quality

It is quite inexpensive to design-in additional features to certain types of VLSI hardware. For example, incrementers have been added easily to VLSI registers [SHRO82], [RUBI82]. When these hardware additions are made, their simultaneous operation increases the parallelism (see the CMUDA "distributed design style" [THOMAS77], [PARH79], [PARTCC79]) and therefore the functional speed of the hardware. Unfortunately, these features also increase

the number of code-generation alternatives and compound the difficulty of microcode generation.

Until now, there have not been any fully automated techniques for code generation in highly parallel microengines. A microprogrammer simply cannot consider each of the thousands of permutations of microword compilation and placement alternatives, and in conventional microcode compilers few, if any, alternatives are considered. Although one approach is to hand-develop compiler software that is triggered by special-case optimizations (such as peephole optimizers [DAVF79], [DAVF80], [TANVS82]), this is merely an ad hoc solution at best that is used only at a particular phase in the compilation process. It is expected that microcode synthesizers, such as the one described in this thesis, will eventually become the primary microcode-production tool. The thesis shows that the UMS approach can produce microcode of hand-coded quality.

CHAPTER II

LITERATURE REVIEW

This chapter describes previous research that led to the UMS microcode generation approach. To generate microcode, this approach repeatedly replaces code from an instruction-set description control-and-data-dependency graph with semantically equivalent code constructs closer to what the hardware can use. Four main developments have made this thesis possible (listed in order of their presentation and increasing influence): 1), research in program semantics (especially verification- and transformation-based program synthesis), 2), research in compiler and computer-aided-design technology (particularly "compiler-compilers" and code-generator generators), 3), logic programming (mainly the Prolog language), and 4), artificial intelligence search techniques. These fields are not cleanly separated; many fields have taken ideas from another to improve the work in their own. This chapter concludes with a discussion of early microcode synthesis research.

2.1 Program Semantics

The design of UMS is based in part on earlier work in definitions of program semantics, which is discussed in this section. The primary goal of a

formal specification for programming language semantics is to communicate the intentions of the language's designer to the language's user community. Without such a specification of the source language ISPS, the UMS compiler would have been much more difficult to create. Formal descriptions are used in analysis of program correctness (program verification). UMS is primarily a code-transformation system directed by heuristics that process the initial instruction-set description as a (relatively) very high-level description. These two topics are typically based more on ad hoc techniques than on mathematics and are discussed under automatic programming and program synthesis in a later section on artificial intelligence. First program verification will be discussed, followed by microcode verification efforts and use of formal semantics in microcode synthesis.

2.1.1 Program verification

Work in the area of program-correctness proofs [MANW85] is the part of program-semantics research most strongly related to UMS. These proofs describe when a computer program both terminates and computes the appropriate function (i.e., provides a specific type of answer). This research has a strong mathematical basis, and has concentrated on languages such as Algol [RUS80], Pascal [PAG81], or pure Lisp [BOYM79]. There are also more mundane uses of such analysis, such as to ensure that two compilers conform to a standard or to ensure that two implementations of the same computer

architecture are consistent. The program-analysis approaches used in program-correctness proofs form the basis for UMS hardware and instruction-set description analysis. Early attempts to improve microprogram-development methodology were based on program-proof techniques [JOYCL76]. Much of the early work in microprogram synthesis was based on microprogram-verification techniques already established in the literature.

Program verification based on mathematical analysis is still a difficult research problem [GER78], [SCH78], [LON79], [CON79], [JEFF80] and has been met with limited success. Alternatively, other interesting advanced uses of program analysis have become possible; these include code-transformation systems [BRO83], [DAR81], [DAR78], [SMIS79], [STAHKN76], [WEG76], [FEA79], [BURD77], [DARB73] and the synthesis of programs from very high-level specifications [BAL81], [DARF80], [DEA80] (also see [MUEJ81] for a comparison of verification and synthesis). These other uses of program analysis are discussed in the artificial intelligence section.

There are three main schools (such as described in [DON76] and [PAG81]) of programming language semantics: operational, denotational, and axiomatic (also called propositional). Much of the research on program verification (for example, see [AND79]) is based on the idea of partial correctness. Given a description of the expected range of input data values

(typically a subset of all possible inputs), when the program terminates, mathematical analysis can determine if it computes the desired function for the small set of input values. This mathematical analysis often culminates in an inductive proof. The inductive-proof step proves that if the program works correctly for some inputs, then it will work for slightly different inputs. The proof is designed so that the proven input values and the small differences recursively cover the set of all input values. Program termination is not an issue in UMS; the instruction-set description is assumed to be correct. However, the program-analysis techniques developed to determine if the correct function is performed influenced the UMS dataflow analyzer design.

An important difference between operational and denotational semantics is their analysis of loops; operational semantics requires that the induction of program correctness proofs be performed on different distinct iterations of the loop, whereas denotational semantics requires induction on different depths of functionally nested expressions (also see [STO77], [GOR79], [BAK80]). Another way of looking at this difference is that operational semantics describes a specific sequence of computation steps while denotational semantics does not imply any particular sequence in its description of state-variable functions.

In comparison, axiomatic semantics uses a set of logical assertions about state-variable values at each part of a program-controlflow graph. These assertions may contain conditions. An axiomatic correctness proof ([WALL77],

also see [MIL85], [GOO85], [CLA85]) requires logical analysis, based on the intervening computation steps, of the beginning and ending assertions for every path through the controlflow graph. Explicit induction is avoided by evaluating the complete set of these logical expressions for each part of a program. The reader is directed to [DON76] for a clear comparison of these three techniques. This previous work provided the theory for the UMS program analysis and condition testing. It allows UMS to replace code from a control-and-data-dependency graph with equivalent code constructs. Some of the semantic conditions tested before applying a specific UMS production are reminiscent of assertions at the beginning or ending of a path.

2.1.2 Microcode verification

Two primary previous efforts in microprogram verification based on an axiomatic approach are discussed here. One, at IBM, consists of computer-assisted efforts to prove equivalence between microcode and a simplified abstract machine [JOYCL76], [CARJB78]. The second effort, by Patterson at Berkeley [PAT76], [PAT77], [PAT81], is the primary example of automated microprogram verification of real instruction sets on real hardware. It requires the microprogrammer to create the (high-level) code, descriptions, and constraints that describe the intended behavior of the microprogram. A theorem prover determines the consistency of the code with the set of descriptions and constraints. The system can check "edge conditions" in

bitfield operations, for example, such as those occurring during addition of 16-bit signed integers that cause 0 = 32767 + 1 < 32767, due to carry overflow [FOSI85], [HWA79] into the sign bit. Both the IBM and Berkeley research groups had interest in extending their work into synthesis.

The types of capabilities just described are certainly an important component of a microcode development environment. From the UMS point of view, however, they involve analysis of an instruction set and the algorithms it depends on, which is beyond the scope of this thesis. UMS does not try to verify the instruction set itself (this issue is also discussed below in the section on automatic programming), so UMS is not required to understand the meaning of the instruction-set description or its effect on the instruction stream [BARS85B]. From the point of view of UMS, arithmetic constraints and hardware idiosyncrasies are dealt with by transformation rules in the programming knowledge base or are implicitly part of the instruction-set description. Any sequence of truth-preserving transformations is assumed to leave unaltered the original semantics of the instruction set. UMS tries only to implement the microcode description correctly by transforming it; UMS does not attempt to check if previously created microcode conforms to a description.

One of the most ambitious efforts at microcode verification was based at ISI [CROMV80A], [CROMV80B]. To prove correctness, the verification system receives four files: the host (hardware) description, (hand-coded) microcode,

target (instruction-set) description, and proof (manually created). The system proves correctness in two steps. First, the system proves that the microcode running on the hardware correctly implements an intended algorithm (such as floating-point square root). Secondly, it proves that the algorithm has the correct properties (such as always returning "error" with negative number inputs). This highly experimental research has been applied to only a small number of examples, and has yet to be used in a practical environment. In relation to UMS, if UMS is correctly implemented, then the first verification step (beyond running the resulting instruction-set microcode) would be unneeded. The second verification step is a research goal beyond the scope of this thesis. A subsequent effort by these researchers using techniques closer to those described earlier shows great promise [LEV84], [MARCL84]. To date, work in microcode verification has been considered to be an important step in empirically demonstrating an understanding of microcode semantics in a formal manner.

2.1.3 Role of formal semantics in microcode synthesis

The use of formal semantics in microcode synthesis is described here. In operation, the UMS inference engine does not use program-correctness proof technologies. This contrasts with earlier research by Mueller [MUE80A], [MUE80B], which is described in detail later, that achieves similar goals by using formal principles. Similarly, Ulrich [ULR80] has suggested that

microcode synthesis should be based on formal principles. However, this earlier unimplemented work does not mention practical approaches for managing the large search space present in microcode synthesis.

The UMS inference engine is primarily a pattern-matcher specifically designed to process control-and-data-dependency graphs. This pattern matcher is not based on formal principles. When required, UMS modifies fragments of the instruction-set control-and-data-dependency graph using transformations described in the programming knowledge base.

Validity of a UMS transformation is usually self evident. However, the axiomatic approach would be the best methodology for formally checking UMS transformations (see [DEA80] or [KIB78] for sample transformations and [WAGD83] for sample proof techniques) because of similar program representation styles. The UMS knowledge base describes transformations in a control-and-data-dependency graph representation form without detailed reference to a program context. Similarly, a small set of axiomatic assertions for the beginning and ending of each controlflow-path fragment in a transformation completely specifies any relevant context. Simplification of these logical assertions for more efficient verification could be based on the symbolic execution (also see [STEK85], [KANN83], [COH83]) techniques of Oakley [OAK79], which are in the domain of hardware register-transfer-level semantics.

2.2 Compiler Technology and Computer-Aided Design

This section describes how UMS relates to other types of software tools. UMS is one of many software technologies which can be employed in the design and use of computers. From this point of view, conventional compiler technology is the primary success in this area, since today machine language programming is seldom necessary. Instead, we are able to specify the computer behavior we desire in a language with semantics close to the application problem domain. Given the large number of computers produced from a single design, the amount of design effort invested by the manufacturer per computer is quite small. However, many business and scientific computers use application programs that were custom written or modified for them. This application-software design effort per computer is probably much greater proportionally than its hardware or system-software design effort. Correspondingly, much more effort has been put into software tools to make application programs, particularly compilers, easier to write.

Historically, the importance of this "computer-aided design of machine code" performed by compilers suggests why it has motivated more research than any other type of design automation. Other computer-aided design systems contain similar types of technical problems (although perhaps with

different degrees of complexity and regularity). This section discusses how both compiler and computer-aided-design technology relate to UMS.

2.2.1 Conventional compiler technology

Compiler technology originated from research in the field of artificial intelligence ([BARR82] page 297, [BAL85B]), where it was classified as automatic programming. As algorithms and techniques became better understood, the discipline became a center of activity in its own right. This section discusses how two issues in compiler implementation, register allocation and span-dependent code, relate to UMS. A later section discusses a third area: automating code generator creation.

2.2.1.1 Register allocation. In typical user-level computer architectures, the small number of general-purpose registers is the only hardware resource that must be rationed within a program. All other hardware resources, such as the ALU or barrel shifter, are at a lower level of detail and are implicitly allocated during the execution of each machine instruction. Since access to the general-purpose registers is often much faster than memory access, programs run faster when values are stored within them. Register allocation, which is absent from UMS, is a separate optimization phase in compilers (see [AHOSU86], [GRI71], [WULFJWHG75], and [ANKCHM82]). Instead, UMS considers registers to be one of many machine resources that must be allocated. The key to this process is analyzing how many registers are

required at each microcode address and when their values are no longer needed.

The unimplemented RUT is intended to be the basis for register allocation in UMS. A catalog of alternative code translations and the RUT could be used to measure the effects of code generation. Such a register-allocation approach would be closer to the problem domain (and use more information from it) than previous attempts using And/Or trees, iterative approaches, or thresholds (fixed or arbitrary) [VEG82B]. Other researchers are moving toward integrating register allocation, code generation, and compaction in microcode [MUEDO84], although not yet to the degree of the UMS design.

2.2.1.2 Span-dependent code. UMS does not currently distinguish between microarchitectures with span-dependent jump instructions (where the destination of a jump must be with a limited range from the current program counter value) and those without. Code with these constraints could use existing techniques ([LEV80], [ROBE79A], [SZY78], [FRIS76], [WILL79]). Alternatively, RUT-based resource processing could expand to consider microword address availability as an additional constraint using the pseudomicroword field techniques described in [POE81A].

2.2.2 Microcode compilation

Microcode compilation [DAV83], [DAVS80A], [DAV78], [SINT80], which specifies the contents of a control store for emulation of a computer architecture

instruction set [PAT83], [LEWS81], should not be confused with the programming of (single-chip) microcomputers. Beginning with the pioneering work of Eckhouse [ECK71], experimental high-level language compilers have been written to generate microcode. Some of the latest microcode-compilation systems can receive genuinely high-level languages and generate code of quality competitive with hand coding [PAT81], [PAT77]. In some cases [PAT78], compiled microcode is superior to microassembler code. This is most likely due to exploitation of global code optimizations, which are most effective in large microprograms and are very difficult to process in microassembler coding. This section discusses previous microcode-compilation techniques and how they relate to the UMS design.

2.2.2.1 Tool building in firmware engineering. Since software tool building itself is an expensive and time-consuming process, the use of tools to build or customize (retarget) other software tools has become increasingly important. A primary measure of progress in firmware engineering is the level of automation achieved, expressed in the use of tools. Unfortunately, microcode-compilation techniques have yet to become commonplace in universities or industry. A major reason is the expense of writing new microcode compilers or retargeting old ones to new microarchitectures. This section discusses the problems that firmware engineering tools must solve.

The basic action that a microcode compiler or synthesizer performs is intelligently binding [MUEDO84] variables and operators from the high-level source language to the target hardware. Weidner [WEI80] suggests categorizing microprogramming languages into a spectrum of six categories: 1) bit stuffer, 2) assembler, 3) register transfer, 4) macro register transfer, 5) procedure-oriented, machine dependent, and 6) procedure-oriented, machine independent. Binding techniques can vary across this spectrum: they can be manually specified in each line of the source code, manually specified by machine-dependent regions within otherwise machine-independent source code, partially automated by using special tables for the compiler code generation, or completely automated. UMS is a procedure-oriented, machine-independent microprogram-development methodology that achieves a new level of automation. UMS does not require either writing a compiler or any other retargeting effort.

Variable binding in microcode has many aspects in common with register allocation in conventional compilers; these include controlflow and dataflow analysis. However, the variability of microcode's hardware-register characteristics makes it much more challenging. A further major complication in microcode is the datapath routing of variables to operators in irregular microarchitectures. Whereas most microcode compilers require manual

evaluation of these details and their expression in tables, UMS automatically processes them.

Operator binding is called code generation in conventional compilers. The design and representation of machine-independent compiler tables of microword (control constraints) or microarchitecture (datapath) structure have been a major issue in microprogramming research [MUE80A], [DAS84], [SINT80]. As is described in the remainder of this thesis, UMS represents the far end of the spectrum for both variable and operator binding: It offers complete automation based on analysis of a procedural machine description. UMS is designed to use the same source code on different microarchitectures without the effort of retargeting a compiler. Davidson [DAV83] argues that this meaning of machine independence is likely to be the long-term research trend in microcode compilation instead of the machine-specific use of general programming concepts discussed in the next section.

Consider that multiple source-code statements can be translated into one microoperation. In contrast, one source-code statement would translate into one or more target-machine primitives in conventional compilers. Sint [SINT80] notes that no approaches have been proposed to handle the problem and reflects:

"This problem is still too difficult to attack. Its solution would involve extensive semantic analysis of the source program, but no suitable techniques for such an analysis are available." [SINT80].

This problem is routinely solved by the UMS strategies for dataflow analysis and synthesis, particularly for sequences of assignment statements. Avoiding shifts in the bitfield extraction example of Chapter VII is another an example. UMS creates mappings between two sets of operations (the hardware and instruction set, where either set may be larger than the other.

2.2.2.2 Machine independence. Some researchers use the term "machine independent" (discussed in [SINT80]) to mean that the same microcode compiler could be used for different architectures. However, the compiler would require retargeting of the machine-dependent aspects (such as register allocation) implicit in the source code. In these compilers [TAKTBAY82], [BABH81], [LLO74], which are best described as representative of the first part of the microprogramming language spectrum, the semantic level of the source code is so close to microcode that little machine-independent compilation takes place. This section describes previous machine-independence work in microcode compilation.

The common approach to machine-independent compilation is a two-step process: first from a high-level language into machine-independent intermediate-level code, then machine-dependent translation from the intermediate language into object code. These translation steps are usually expressed in manually created tables. It is extremely difficult to design a

general-purpose intermediate-level language, which is a problem very similar to designing a general-purpose microarchitecture model, described below.

YALLL by Patterson [PATLT79] is a sample intermediate-level language for a microcode-compilation system that requires some machine-dependent variable binding. The work by Ma [MA78], [MAL80], [MAL81] uses hand-written tables to create a mapping between the intermediate-level language and the target hardware. MARBLE [DAVS80B] and Micro-C [HOPHA85] requires similar tables. Too often these languages are too restrictive for practical code representation [MAL81]. These approaches (also see [GIES82]) require manually creating additional tables of microoperation resource use and timing constraints. These techniques require considerable manual effort compared to UMS's approach of automatically using information implicit in a procedural description.

2.2.2.3 Microword models. Most researchers continue to base microcode assemblers and compilers on architectural preconceptions with limited flexibility, as documented in this section. Although none of these models has proven to be completely satisfactory, many researchers still feel the need to "Develop a 'complete' model of the semantics of microoperations and microinstructions" [DAVS78]. Sint [SINT81] provides one of the most flexible microinstruction description languages for timing constraints. Dagsupta [DAS80], [DAS79] in the S* formalism allows specifying microinstructions as

Pascal-like records and supports multiple execution phases of a microword. Work done at the University of Louisiana [LANDSM80] provides a fairly complete functional microword model relative to compaction without specifying how it would be described in the source code of a machine description. In contrast, UMS uses a single procedural description, and its code-analysis techniques are used to extract resource and timing information.

2.2.2.4 Microarchitecture models. Microcode compilers must cope with more architectural variability [MALL75] than conventional compilers (discussed later in the section on automatic code generation) while dealing with the same problem of overconstrained machine models ([MAL81], [MAL80], [DAS78], [DAS80A], [DAS80], [AGRR76]). These efforts are described here. Although other work has not addressed the questionable need for microarchitecture models, it has refined existing modeling techniques to make models easier to create. The "tuple" construct in S* [KLAD81] allows using alternative names for variables or registers with flexibility comparable to ISPS (also see [DAS84]). The V-Compiler [PATGPS81] provides conventional variable-definition facilities and manipulates individually written machine-dependent microword descriptions (called FIT and COMBINE) with a custom-designed model of the microword's contents. Microoperation-sequencing constraints between microwords in the V-Compiler are explicitly described in the source code [POE81A].

This author argues that it is premature to try to develop a model of hardware or microprogram semantics given "the explosive rate of progress of hardware technology" observed by Ma and Lewis [MAL81]. Instead of a fixed declarative model, UMS uses a procedural model of both hardware and instruction-set architecture and semantics, which are written similarly in the same language (ISPS). Pattern matching between these two models performs variable and operator binding in microcode synthesis.

2.2.2.5 Microword compaction. Microword compaction efforts (such as [ATK80], [BABH81], [DAVLSM81], [DEW76], [FISLS81], [HENMA83], [KLER71], [MAL78], [TOKTT81], [TOKTTT77], [TOKTTY78], [TSUG76], [TSUG74], [VEG82B], [WOO79], [WOO78]) are divided into two types within the literature: local (see [LANDSM80], [AGE76], [DAS79] for reviews) and global (see [POE80], [FIS81], [FIS79] for reviews). Local compaction places microoperations into sequences of microwords (called basic blocks) that do not contain controlflow forks or joins (except possibly at their boundaries). Global microcode compaction, a more complex extension of this work, can process controlflow forks, joins, and loops. This section places compaction into a historical perspective.

Global microcode compaction did not achieve wide acceptance by the research community until the Fisher thesis [FIS79] showed the close relationship between job-shop scheduling and microcode compaction. Fisher

also proposed a technique called trace scheduling that picks for compaction a complete execution trace through a microprogram, across any basic-block boundaries. Improvements or extensions have been proposed recently to the original trace scheduling ideas [SUD85], [SUDJ84], [LIN83].

Although trace scheduling has gained increased acceptance, it nevertheless contains two primary flaws.

The first flaw, common to most scheduling algorithms, is that it does not make use of all available data-dependency and resource-constraint information. The primary difference between scheduling algorithms is the determination of the order in which microoperations are serially chosen for placement into microwords. When a microoperation is considered for a microword, little if any information from the unchosen microoperations affects placement. In contrast, the UMS RUT delays making placement decisions until compelled to, while globally analyzing how all microoperations affect each other.

The second flaw of trace scheduling (also discussed in [POE80]) is that parts of the microprogram that are never used together in practice may be placed together in the same microwords at the expense of other potential optimizations. For example, consider a microcoded emulator like the one shown in Figure 1.2, and the parts of its microprogram that perform the most commonly used operand specifier (part 3) and the most commonly used

machine instruction (part 5). Trace scheduling would place together microoperations from the most common operand specifier and the most commonly used machine instruction (even though they may seldom be used together) at the expense of other operand specifier and machine instruction code. This strategy is likely to result in slower speed for the remaining operand specifiers and machine instructions, and a generally slower computer speed.

Researchers are becoming increasingly aware that decision-making order may have dramatic affects on the final outcome. Researchers at Carnegie-Mellon University noted:

"Premature binding of decisions can lead to poor designs. Better design choices can be made if decisions are postponed until adequate information is available to make them." [RAJT85].

The early work on local microcode compaction was one of the early examples of this trend. Further, in complex computer-aided design systems, the decision order may cut across traditional categories. Researchers are exploring the integration of register allocation, code generation, and compaction in microcode [MUEDO84], [VEG82A], [VEG82B]. The integration of microcode synthesis and compaction in UMS is a further step in this direction.

2.2.2.6 MIMOLA. The MIMOLA microcode compiler [MAR84], [MAR81B], [MAR80A] approach is discussed here. It was designed as part of a top-down hardware-synthesis system (described below in the section on computer-aided design). The hardware-description analysis of MIMOLA is less

extensive than that of UMS. It uses internally tree-like structures derived from descriptions in a Pascal-like language, although this language is not strongly typed. Before parallelism detection and analysis, it allocates those storage locations that have not been already allocated manually. In contrast, the intended use of the RUT by UMS should fully integrate storage allocation and parallelism exploitation. Instead of the UMS global dataflow analysis of buses for variable values, MIMOLA employs the concept of "detours." Detours are one step along a bus in a hardware description, and MIMOLA's actions limit their use to a fixed number. Experience with UMS has shown that global analysis is necessary.

The customization of MIMOLA for a specific microarchitecture is largely a manual process "because we believe in the great worth of human intuition and experience" [ZIM80]. Many program-transformation rules are applied unconditionally and are located within the hardware description to customize the compiler for a target microengine. This suggests that the system is "machine independent" in the same sense that Sint [SINT80] criticizes; this means that the same microcode-production system could be applied to different architectures but that the same high-level language microcode would most likely not work unchanged on a different microengine.

MIMOLA relies on user intuition and experience to minimize combinatoric growth of computational complexity. Although it is unclear how well this

succeeds in MIMOLA, other work in microcode compilation has achieved superior results with automated decision making [PAT81], [PAT77]. Like UMS, MIMOLA tries to match fragments of the instruction set to parts of the hardware. However, it does not support the analysis and modification of partial matches as UMS does. MIMOLA contains the concept of value-preserving operations using "neutral hardware elements," such as adding zero to a value, but the concept of commutativity (the ability to swap operands for some types of functions) is not mentioned. Changing previous mappings based on newly discovered constraints, such as the UMS variable-remapping technique, are also not mentioned. UMS is designed to solve a different set of problems than MIMOLA, which is an interactive, user-controlled microcode-development system.

2.2.3 Automatic compiler-generation techniques

This section discusses automatic compiler-generation techniques primarily for conventional compilers. Automatic parser creation [AHOSU86], [AHOU77] (which recognize symbols in the source code) is one of the success stories of computer science. However, a complete compiler requires more than construction of just the parser. After being temporarily stored in an intermediate form, the parsed symbol meanings must be translated into target instruction-set operation sequences by the code generator. So far, automatic code-generator creation has not been as successful as automatic parser creation [GANAFH82],

[JONS80], [WAI76]. UMS is designed to generate microcode automatically after analysis of the hardware description. This section discusses how the design of UMS relates to research in generating machine code automatically based on instruction-set analysis.

A common criticism of automatic code-generation research is that it has not yet provided practical tools [GANAFH82]. Most research has concentrated on code-generation choice at compiler-creation time (the compile-compile time described by [LEVCHNRSW79]). However, some of the latest research places high importance on the compile-time efficiency of the resulting code generator [GANAF85]. Code-generation often research has considered only arithmetic expressions while ignoring controlflow. Register allocation is sometimes considered to be a separate, unrelated topic. Only architecturally and mathematically clean instructions are considered, while the complications involved in various addressing modes (such as autoincrementing index registers or indirect addressing) are not processed. Expensive exhaustive search is used sometimes [AHOU76], [NEW75].

2.2.3.1 Machine-model types. Techniques to describe a target computer are discussed here. Two processes are involved in the automated code-generator creation. These are the information extraction about code-generation alternatives from a machine description and choosing from the alternatives of the most efficient code sequence for each parsed

source-language fragment. The second process is discussed below. Most previous research [GANAFH82] requires manual extraction of information from a machine description. This often requires the user of the code-generator system to describe the target instruction set in a special description language that is usually based on a model of an idealized machine architecture (see [AHOSU86], [VEG82B], [DONNF79], and [CATT78] for examples). In contrast, UMS automatically extracts architectural information from a procedural description (simulation) of the machine without architectural preconceptions.

To create the two processes described above, some approaches to automated code-generator creation contain two levels of translation: first from the source-code language to an intermediate form, and then from the intermediate form to the object-code language. This is often called an interpretative approach in the code-generator literature [GANAFH82] because the code generator creates a code stream that implicitly interprets in line the semantics of the intermediate-level language. Note that the machine description is strongly coupled with the code-generation algorithm. However, this code-generation approach does not consider the original source-code context. The interpretative approach was widely used in early compiler-generation research (such as UNCOL [STRWTOMS58], MIML [MILL71], P-code [AMM77], and more recent use [PATGPS81]). The intermediate-level language

design requires computer-model architectural preconceptions. Typically, these models contain limited numbers of stacks, registers, and addressing modes.

The intermediate-level language model of the target machine architecture may significantly constrain the best code expression for some types of source-language semantics or instruction sets. The lack of consensus on computer instruction-set design suggests how difficult it will be to design an intermediate-level language that represents semantics efficiently. Consider a search for the expression of intermediate-level language phrases by the code generator. During the search, the target architecture model may be inappropriate or severely constrain using some speed-enhancing features, particularly special instructions or registers. As an extreme example, the intermediate-level language may not contain the concept of byte character-string search. Such an operation would have to be expressed as a sequence of simpler operations in the intermediate-level language. If such an operation was available in the target architecture, it would not be exploited due to the difficulty in detecting what is done in the source code and matching it to the instruction description.

Although some of the latest research semiautomatically extracts needed information from a machine description, the same problem remains: an idealized machine model is used to view use of the computer. A procedural description (such as used by UMS) does not contain such a bias and may allow

recognizing and exploiting higher semantic-level primitives (due to the easier matching of code in graph form), such as the byte character-string search mentioned above. The description language used by UMS (ISPS [BARN78], [BAR79B], [BAR79C], [BARBCS78], [BAR79], [PARTCC79], [BARS82]) was designed to be rich enough to describe both instruction-set architectures and hardware at the register-transfer level.

2.2.3.2 Representation of machine descriptions. The way that a machine description is internally represented can constrain how it is used. Gradually, various researchers evolved the machine-description representation from a purely syntactic form (see the section on Miller, below) to a more abstract token tree (also discussed below). Since many compilers represent intermediate code in tree form, this may seem a trivial innovation. However, it changed the perspective of machine description use. Using trees instead of a linear syntax increased the number of pattern types that could be easily recognized and used. A node of a machine-description tree may represent one of many variously accessed operands, which in turn could represent different subtrees. One of the most successful examples of tree-based code generation is the portable C compiler [JOH78].

Unfortunately, tree representations limit the recognition of some types of patterns because trees describe only local data dependencies. The UMS example in Chapter VII uses knowledge available only from a global

data-dependency graph. It allows the value read/write pair and its previous calculation to be freely modified once it is determined that there is only one read to the data value. Other contemporary examples of machine descriptions have not yet demonstrated the ability to represent this type of information efficiently.

2.2.3.3 Miller. In early research based on intermediate-level languages, the code-generation algorithm was strongly integrated with the machine description. The first research to separate the code-generation algorithm and the machine description was performed by Miller [MILL71]. However, this early attempt is not much more than a taxonomy of conditionally applied instruction macros that must be rewritten by hand for each new machine.

2.2.3.4 Fraser. The research of Fraser [FRA77] uses a knowledge-engineering production-system approach to code-generator creation. The approach is similar to its predecessor (Wick, [WICK75] on the automatic assembler creation). The system analyzes a modified machine description written in an ISPS subset, relative to a predefined machine model. Little effort was made to deal with either code efficiency or the computational efficiency of the system. The use of trees to describe code fragments required throwing away potentially valuable information (such as condition codes) in most cases. Later research [FRA79] describes the peephole optimizations

needed to bring the generated code up to conventional standards of quality (also see [DAVF80], [DAVF79]). The research did operate on arithmetic expressions or controlflow operations (although expecting a particular type of architecture).

2.2.3.5 Formal approaches. Some of the more successful ideas from parsing began to influence code-generation research [RIP75], [GANRW77], so that some mathematical analysis was possible; compare this to the more common ad hoc approaches, which often used exhaustive or expensive search ([FRA77], CATT78]) to find provably optimal local-code sequences [AHOU76], [NEW75]. Unfortunately, these efforts also required the use of an intermediate-level language with the same drawbacks as the interpretative approach [GANAFH82]. As with research in program verification, the mathematics have not evolved [JONS80] to the point at which the technology can be a practical, commonly used tool.

2.2.3.6 PQCC. Perhaps the technology of traditional compilers closest to UMS is that of the PQCC (Production-Quality Compiler Compiler) project [LEVCHNRSW79], particularly research on automatic code-generator creation by Cattell [CATT78] at Carnegie-Mellon University. To emphasize some of the similarities of this work to the microcode domain, the V-Compiler has sometimes been called a Production-Quality Microcode Compiler. This and the following section will describe how PQCC relates to UMS.

The "CC" part of PQCC indicates that the system is a compiler-compiler: the system receives information about the source language and object-machine language and generates a compiler. The "PQC" part of PQCC describes that production-quality compilers (comparable to those commercially available) are the results of this system. However, the system is not as yet a "PQCC" (and may never be) because experts must "hand-hold" the system at every step in the development of a compiler. In comparison, the microcode-synthesis system described in this thesis is intended to be used by CPU engineers who are not experts in artificial intelligence, compiler design, or program synthesis.

2.2.3.7 Cattell. Cattell's research [CATT78] at one time was the basis for compiler retargetability in the Production-Quality Compiler Compiler (PQCC) project at CMU. This work's goal was to automatically derive code generators for algorithmic high-level languages from machine descriptions of register-oriented instruction sets. The following discussion compares this work to UMS.

Cattell describes his research as complementary to Fraser [FRA77]. The production system he uses can make syntactic transformations only on a representation of arithmetic code based on trees called TCOL. The production system does not compute parts of the RHS (Right-Hand Side, or the "after" part of a before-and-after description) of a production rule for a specific application,

a feature shown to be very useful in the example of Chapter VII. This research found a minimal sequence of operations to perform an AND operation on the PDP-11 (which has only inverted AND) and a SUBTRACT operation on the PDP-8 (which has only ADD and TWO's COMPLEMENT). These examples are relatively simple compared to the examples discussed in this thesis, in which there are a wider range of operations and orders of magnitude more detail. The problem of routing control and data values through a microarchitecture is not relevant to PQCC, which does not deal with parallelism or the problem of routing data through a maze of hardware registers, computational elements, or buses.

This research deals with trees that have up to about a dozen nodes. Since the code-generation objects in this work were structurally much simpler than those used in UMS, efficient data representation was not as important. Experience with UMS has shown that trees are not adequate to effectively represent more complex pieces of code. Later research on the CMU RT-CAD (Carnegie-Mellon University Register-Transfer Computer-Aided Design) system, which contained ideas from PQCC, instead uses UMS-like graphs called "Value Traces" or VT [SNOW78], [SNSITH78]. Other microcode-synthesis researchers have begun to see the advantages of graph representation [MUEVA84], [MUEDO84], [MUEV83]. UMS works with directed graphs of control and data dependencies that contain hundreds of nodes.

Cattell designs a model of instruction semantics called M-ops (for machine operations). M-ops contains three parts that describe machine instruction operands: 1) storage bases that describe parts of the processor state, 2) access modes that describe addressing modes, and 3) operand classes that describe which storage bases can use which access modes. Implicit in each M-op are the data types that it uses and the required instruction fields or formats. The use of such a model is fundamentally different to the philosophy of UMS, where only the ISPS code describes both the hardware machine and the instruction-set machine. UMS deduces any other required information. As mentioned earlier, arbitrary intermediate models of machines may not only lose information, but also may place arbitrary restrictions on the generality of a code-generation system.

In Cattell's system, M-op information is extracted manually from a machine description. Chapter IV of this thesis describes why the automation of this activity is not trivial, particularly for microarchitectures. Cattell's research is not designed to achieve the level of efficiency needed at compile time and is intended to be a compile-compile time tool. Perhaps the most important point is that the system is not practical; subsequent researchers using PQCC tend to write the code-generation portion by hand [NEWC80], [LEV82] instead of using Cattell's automatic code-generation approach.

2.2.3.8 Graham-Glanville. This section describes more formal techniques, which are based in the mathematics of LR parsing technology [AHOJ74], that were developed to represent and choose between code-generation alternatives (some of the latest research of this type comes from Berkeley [GLA77], [GLAG78], [GRA80], [GRAG78], [GRAHS82]). This work matches with parsing techniques the internal intermediate-level language code trees to chunks of semantics from an instruction-set description. The grammar of instruction-set primitives is almost always ambiguous since an instruction set can often perform some high-level actions in more than one way. A modified parsing algorithm tries to resolve ambiguity at the last possible moment, biased toward more complex instructions. Instead of using a fixed bias toward more complex instructions, UMS makes operation choices based on context. Emphasis was placed on compile-time efficiency, while special software tools create the necessary tables at compile-compile time. Some of the latest Berkeley experiments have shown that this compiler-retargeting approach is very efficient between similar machines and languages [HEN84].

2.2.3.9 Ganapathi. Ganapathi [GANA80], [GANAF85], [GANAF84], [GANAF81A], [GANAF81B], [GANAFH82] has extended the early Graham-Glanville research to better handle the semantic constraints, such as dedicated registers or even/odd register addresses, caused by some architectural irregularities. The internal representation is tree-like, which

constrains machine-model semantics and requires an integrated peephole optimizer [GANAF85]) to improve pattern-matching speed. Some of the latest experiments have also shown this approach to be very efficient [GANAF85]. However, like the Graham-Glanville research, the code is separated into basic blocks (regions of semantics bounded by labels and jumps). In contrast, UMS is designed to process code fragments across basic-block boundaries.

2.2.3.10 Comparison of code generation for conventional compilers and microcode synthesis. This section compares the different types of research issues involved in conventional compiler- and microcode-synthesis technology. Consider the research in automated code generator creation for conventional compilers. This research assumes that the primary problem in code generation is creating a tree of instruction-set actions that semantically matches that of a high-level language parse-tree fragment (which has been translated into an internal intermediate-level language). Although this is certainly problematical for conventional compilers, solving this problem leaves many related and interrelated problems for microcode compilation.

Compared to microengine instruction sets, CISC instruction sets are highly regular and much easier to write code generators for. The trees of code semantics are relatively simple in microcode. However, controlflow in microengines has much more complex semantics [DAS79], [HASJ79], [ADA78],

[ARM77], [SAL76], [HUS70], and any operations on data in horizontal microcode must consider datapath and resource contention [VEG82B], [TOKTTT77]. Due to irregularities of the microengine, tradeoffs must occur between controlflow and dataflow (see Chapter VI for an example of modifying the dataflow input to a conditional jump condition). This type of code tradeoff analysis, which includes both controlflow and dataflow, is beyond the demonstrated capabilities of compiler-creation systems based on existing research. It is even unclear whether the techniques developed for conventional compilers could process these irregularities.

Context is important in horizontal microengines; alternative ways of performing an action must be considered based on any existing microengine resources used by other microoperations within the same microword [VEG82B]. In contrast, most code-generation research for traditional compilers has strived to find the one optimum expression for each type of source semantics, with the results typically expressed in tables. It is often believed that if all possible code-generation context was added to the code fragment to achieve table indexing then the tables would explode in size. Further, the computational cost of deriving such large tables would be prohibitive. Therefore, the search-based derivation of experimental code fragments, which begin to accumulate in table form, seems to be the best strategy in UMS. The recording of previous search by the RUT in UMS promotes additional computational efficiency.

Microprogramming tricks become more important to achieve speed with more irregular microarchitectures. Research to date on conventional code generation does not consider inserting an extra operation (such as X+0) into a fragment of code. However, this was necessary in an example of Chapter VII to synthesize a fragment of microcode. Although techniques that generate specific constants in microcode [VEG82B] have been explored, they are relatively limited. Such techniques do not perform counterintuitive transformations such as the one just described, which is based on deep analysis of the microarchitecture. Although the more formal methods of code generation described above are attractive for conventional compilers, they are not yet completely understood enough for widespread use. The knowledge-base approach used by UMS seems to be the only practical approach at this point in the evolution of the field.

2.2.4 Computer hardware description languages

This section discusses the UMS requirements for a hardware-description language and why ISPS was chosen as the vehicle to meet these requirements. UMS requires using a computer hardware-description language to specify the target microarchitecture and the microcode intended to run on it. To constrain the UMS implementation effort, it was desirable that the language specify both the target hardware and the semantics of the desired microcode.

Instead of inventing a new language for UMS, an existing well-known, widely accepted, and easy-to-obtain language was needed.

Many hardware-description languages have been proposed and implemented in the literature (see [SINGT81], [SHI79], [LIP78], [SU74]). The parallelism implicit in hardware makes AHPL (A Hardware Programming Language, based on APL) [HILSMN81], [HILP73] attractive because of the simultaneity of its vector and array operations. It was unclear whether MIMOLA [ZIM80A], [ZIM80B], [ZIM79A], [ZIM79B] was widely accepted or rigorously defined. PMS [SIEBN82], [SIE74B], [BELN71] was rejected because it is at too high a level of abstraction; it is designed to describe configurations of computer processors, memories, and switches (without any executable semantics). CDL [CHU65] and DDL [SHIC82], [UEHMSK81], [DIE77], [CORDV79A], [CORDV79B] were not readily available. SLIDE [ALTP81], [WALP79], [WAL79A], [WAL79B], CONLAN [PILB82], [EVE81] and Edinburgh LCF [GOR81] were only beginning to be used for hardware description and were considered to be more risky choices.

ISPS is a well-known ([SIEBN82], [BARN78], [BAR79A], [BAR79B], [BAR79C], [BARBCS78], [PARTCC79], [BARS82], [SIE74A], [BELN71]) and widely accepted language that was readily available at the author's former place of employment. ISPS is a general-purpose programming language invented at Carnegie-Mellon University used for a wide range of research:

"...besides simulation and synthesis of hardware, software generation, program verification, and architecture evaluation and control are among the current applications bases on ISPS..." [BAR79B].

ISPS is particularly applicable for the needs of UMS (hardware and instruction-set descriptions) because it is excellent for creating register-transfer-level hardware descriptions and algorithm specifications. Many computer architecture descriptions are available in computer-readable form from the CMU. Most importantly, these published descriptions [SIEBN82] included both a microengine and an instruction-set description that were an ideal test for UMS. Wide acceptance and availability of ISPS, as well as use of previous work based on it, made it the strongest candidate for use with UMS.

2.2.5 Computer-aided design and engineering

The complexities of engineering have spawned a research area called computer-aided design or computer-aided engineering (or CAD/CAE). CAD programs are used in many engineering fields to help humans cope with a multitude of design details and tradeoffs. In CAE, computer-hardware description languages and computer databases are used to describe partial designs at almost every step in the engineering process. Too often, these CAD/CAE programs process a design at only one specific step, and data compatibility (both syntax and semantics) between programs has been a major

problem [FREG82]. This section compares other CAD/CAE systems for computers with the UMS design.

2.2.5.1 RT-CAD. The Register-Transfer-Computer-Aided Design (RT-CAD) project, which is part of the Carnegie-Mellon University Design-Automation system (CMU-DA), is an attempt to create an integrated system that synthesizes most aspects of computer hardware systems. The following discussion criticizes this work based on experience developing UMS.

Hafer and Parker [HAPA78] describe a hardware-allocation process that is the synthesis of a hardware datapath from a behavioral description. This work supports a fundamental separation between data-memory and control allocation, whereas UMS does not. Consider, for example, the action of multiplexer hardware. The value of one member of a set of data inputs is assigned to the output depending on the value of its control input. In other words, a multiplexer acts to switch one of many inputs to an output, and thus can be described as a case statement. It could also be classified as a controlflow action and implemented as a single microoperation in multiplexer hardware. Alternatively, it could be considered a dataflow action and implemented as a branch table in software. In specific contexts, either of these might be appropriate, and the appropriate cost-benefit tradeoff should be made. To ignore such alternatives invites inefficient synthesis. The UMS program-

transformation approach avoids the rigid classification of semantics inherent with this previous research.

Although the Hafer and Parker work does mention parallelism, it only exploits the parallelism made explicit in the ISPS [BARBCS78], [PARTCC79] language by the code author. When they use variables in their functional description of the hardware, they assume that the variables are "subject to random change." This is too conservative for high-quality code, and they admit that this strategy has a "strong influence on temporary storage allocation" [HAPA78]. The global control-and-data-dependency analysis performed by UMS would be an important addition to this work.

Although microcoded central processing elements are a popular implementation style, CMUDA, its successors, and competitive systems [WALT85], [SHI83] to date have not achieved the goal of this thesis: synthesis of microcode. Instead, the high-level design techniques [NAGLE81], [KOWT83] have been of the "hardwired" type. This research sometimes uses a "design style selector" [THOMAS77], [SNSITH78] to choose between "central accumulator allocation," "pipeline allocation," and "distributed memory and data elements." Comparison of the datapath synthesis for the PDP-8 effective address calculation in [KOWT85] or the design of a PDP-8/E datapath in [LEIT81], [PTSBHLK79] with its microcode-based counterpart in Chapter VII raises concerns about the practicality and quality of rigid design styles. Nagle

and Parker reflect on previous research using strongly styled design:

"Most approaches have involved stylized designs and restricted sets of digital components, single goals of either price or performance optimization, or optimization of one aspect of the hardware independent from other design dimensions." [NAGP81]

Other research has concentrated primarily on either the datapath [HITT83], [SHRO82] or control [CLOUT80], [NAGLE81], [NAGP81], [NAGLE78] but as yet not both [HAPA81] to the extent shown in UMS. Much of this can be explained as a tendency to interpret the distinction between control and data operations in ISPS description very literally:

"Digital system design has normally been separated into two main tasks, data path design and control design..." [CLOUT80],

without considering the types of transformations between controlflow and dataflow that UMS can perform.

Although the RT-CAD does not seem to have any formal connection to the PQCC project (described above) other than using the same ISPS description language, early RT-CAD work did use tree-like internal representations of descriptions similar to the work of Cattell [CATT80], [CATT79], [CAT78]. Later research has moved toward the graph-like representation required by UMS [HITT83], [SNSITH78].

2.2.5.2 MIMOLA. MIMOLA (Machine Independent Microprogramming Language) [MAR82], [MAR81A], [MAR80B], [MARZ79], [MAR79], [ZIM80A],

[ZIM80B], [ZIM79A], and [ZIM79B], developed in West Germany, is a design-automation and microcode-production system that performs some hardware "synthesis" in a top-down manner (the microcode production aspect was discussed earlier, see [ZIM80B] for an overview of its design-automation aspects). This section compares MIMOLA's computer-aided-design features to UMS.

MIMOLA takes a functional description of what the hardware is to do and a set of declarations of the maximum number of particular types of hardware that may be used. The output of the system is a description of the synthesized hardware and is similar to the CMUDA "distributed" design style.

The MIMOLA top-down hardware-synthesis strategy may overlook fortunate synergies in the way that related parts of the hardware are designed. The control of design-space exploration is performed by hand via declarations in the initial source code. No attempt to automate the design-space exploration is made "because we believe in the great worth of human intuition and experience" [ZIM80]. The system initially receives declarations for relatively large amounts of hardware. After the design is made, a statistical analyzer determines how much each piece of hardware is used. This information indicates to the user how to refine the initial hardware declarations for a subsequent design pass. This approach is in strong contrast to the highly automated UMS techniques.

In [MAR79], the part of the system that creates a mapping between the functional description and hardware is called the allocator. No mention is made of using transformations on the functional description to allow running the description on the hardware if a particular type of functionality is missing. Such transformations are a basic part of the UMS system. It seems to assume that no such transformations are necessary or that they are performed by hand. If transformations may be made to make the design more efficient, then these also seem to be done by hand, as prompted by analysis of the statistical analyzer output. Like UMS, their work with a hardware description does not require any specific microassembly language, but directly uses the microword field descriptions.

Initially, the fragments of functionality (described internally by trees) given to the MIMOLA allocator are quite large. If the large fragment does not map directly to hardware, then the fragment is divided into a small first part and a remainder. The allocator is recursively called on these two subfragments. In contrast to UMS, this top-down strategy ignores the possibilities of time-saving interactions between two subfragments. There is mention of the microprogram-compaction problem literature only in a much later paper [MAR81A]. The compaction algorithm described there is strongly biased by the top-down flavor of the rest of the MIMOLA system and is very unlikely to be near optimal. Although this later research does describe a retargetable microcode

compiler, the code-generation tables are written by hand, not synthesized as in UMS.

In [ZIM79A], the system takes microoperations divided into microwords and generates a description of the necessary hardware. Because there is no reported use of artificial intelligence techniques to scan the search space, it is assumed that the user of the system must make all these decisions. However, work with MIMOLA [ZIM79B] has shown that once Kuck type [GAJPKK82], [CASJK80], [ABUKL79], [BANCKT79], [KUC78], [SAMK78], [KUC77], [CHENK75], [KUCBC74], [KUCMC72] transformations are automated, significant speed increases can be found [MAR81B]. In contrast to the degree of automation shown by UMS, MIMOLA is a manually controlled microcode-development system.

2.3 Logic Programming

This thesis has been completely implemented in the Prolog (PROgramming in LOGic) language [VANW86], [BALL85], [POENPS84], [CLOM81]. When planning UMS, the three primary requirements for its implementation language were that it be 1), based on unification, 2) support search and backtracking, and 3) possess good database capabilities. It was quite a surprise to this author to learn that those requirements make an

excellent definition of the Prolog language. Prolog is also a good language for implementing compilers [WAR80], [PERW80] and expert systems [MIZ83]. Although Prolog does have a built-in inference engine, major extensions to its search strategy [POE84] were necessary to make it useful for UMS.

2.4 Artificial Intelligence and Heuristic Search

As mentioned in the earlier section on compiler technology and computer-aided design, the idea of using computers to automate some of the program-development task has been around for decades. This section discusses some of the artificial intelligence origins of this technology and how the classic artificial intelligence search problems relate to the novel UMS microcode-synthesis problem domain.

2.4.1 Automatic programming

This section discusses automatic Programming (AP), which is the automation of various parts of the programming activity. This thesis is an attempt to automate the microprogramming task. The Handbook of Artificial Intelligence [BARR82] reminds us that the first compiler was billed as "automatic programming," and that:

"in a sense, all of Artificial Intelligence is a search for appropriate methods of automatic programming..." ([BARR82] page 297).

2.4.1.1 Program specification. AP contains three main methods to specify desired program behavior: 1), a formal specification (such as a very high-level programming language, sometimes called a program schema [SMI83], [GRE83]); 2), specification by examples, or 3) natural language. When a program specification is written in a very high-level AP language, the target language is typically a conventional programming language, such as Lisp or Fortran. This section discusses how these research directions relate to UMS.

In UMS, a formal description defines precisely and unambiguously instruction sets and hardware behavior. Neither of the other two specification alternatives is appropriate for microcode. UMS does not attempt to determine the meaning of this description or to analyze algorithm-design choices. In contrast, some AP systems attempt to understand the "meaning" of the program description [SHRO79] or even redesign algorithms [KAN85], [KANN83], [BARS80B] while attempting to process incomplete or inconsistent descriptions [BARRB2].

The UMS description language (ISPS [BARN78], [BAR79B], [BAR79C], [BARBCS78], [BAR79], [PARTCC79], [BARS82]) is at a relatively conventional level. However, the destination language is at a much lower level: microcode. Therefore, UMS spans a similar semantic distance compared to the conventional AP paradigm. The data types used by UMS are only the bitfield; it

does not use abstract data types [GUI83], either alone or in conjunction with a hierarchy of data-refinement classifications [BARS79].

The handbook further suggests that AP research uses four main code-generation techniques: the theorem-proving approach, the program-transformation approach, knowledge engineering, and traditional problem solving. Each of these techniques is discussed below in relation to UMS.

2.4.1.2 Theorem-proving approach. Most closely related to axiomatic program semantics, the theorem-proving approach specifies a program by a description of its inputs and outputs. The theorem prover proves that a program can exist to create the specified outputs from its inputs. The required program itself is a by-product of a constructive existence proof. Similarly, the action of the UMS inference engine [POE84] proves that a mapping can exist between each part of the instruction-set description and the hardware description. The details of this (existence) proof describe the microcode in detail. The resolution theorem prover [KOW79], [ROB79], [NIL80], [COHF82] power implicit in the Prolog language ([BALL85], [POENPS84], [WOSOLB84], [MIZ83]) is used to implement the UMS inference engine. Also similar to axiomatic semantics, the code transformations used by UMS sometimes contain partial specifications (as constraints, also see Ganapathi

[GANA80]) of code inputs and outputs in a way reminiscent of axiomatic program semantics.

2.4.1.3 Program-transformation approach. The program-transformation approach to automatic programming has strongly influenced UMS, as described in this section. UMS uses rules to transform the instruction-set description into an equivalent program that may run on the target microengine. The resulting program is so changed that it has different intermediate variables, controlflow constructs, and selection of data operations. Further, subroutine calls may be expanded in-line. Often, these transformations make the resulting object code harder to read. Although UMS currently does not perform recursion removal ([BURD77], also see [FEA82], [DAR81], and [DARB73]), there is no reason to preclude this feature.

Both the transformational and knowledge-engineering approaches to automatic programming have influenced the design of UMS. The transformation describing the commutation of operands for addition is directly embedded in the inference engine of UMS and is merely a syntactic transformation. The bitfield movement used in an example of Chapter I is a nonsyntactic transformation triggered by specific code analysis. This second example, which is based on expert insight, would perhaps be best classified as knowledge engineering.

2.4.1.4 Knowledge-engineering approach. The automatic programming knowledge-engineering approach [HAYWL83], [DAVL82], [DAV84], [WINP84], [BARS79] discussed here attempts to codify human microprogrammer insight as a set of rules. Whereas the problem-decomposition approach ([SMI83], [WAL77]) of some AP systems relies on gradual refinement of the entire program specification into lower-level code [BARS79], [KANB81], [KANN83], UMS uses two steps: first, each code fragment is processed into microoperations and, later, into microcode.

UMS typically is used in a "cut-and-try" exploratory programming technique [BAL85A]. It is run until it cannot complete synthesis, the failure manually analyzed, any required rules are added, and UMS is run again. As with previous AP systems (see [BARS85A], [BARS79] pages 225 to 229, or [FRASER77] page 40), UMS has required fewer new rules, transformations, or manual intervention as the system has been tried on new examples.

There are two types of information in the knowledge-engineering approach: domain knowledge and programming knowledge.

Domain knowledge describes information about a particular problem domain, such as procedural computer hardware descriptions. With the exceptions of hardware description analysis for read/write use, indexing of variables, bus analysis, and use of the RUT, and microprogram controlflow, no other knowledge in UMS is specific to the microprogramming problem domain.

Programming knowledge ([KIBNS77], [SMIS79], [STAHKN76], [STAKN76]) describes most of the remaining knowledge used by the synthesis system. Some of this knowledge includes program transformations such as strength reduction, arithmetic identities, constant folding, and controlflow manipulation often found in the optimization phase of conventional compilers ([AHOSU86], [WULFWGH75]). However, most of this knowledge is more specialized and detailed in that it describes alternative ways of implementing some operations, such as the logical exclusive OR, often found in conventional architectures.

2.4.1.5 Traditional problem-solving approach. This section discusses the traditional problem solving approach, which decomposes a problem into successive sets subgoals. Prolog, the language used to implement UMS, implicitly uses this style of programming. This style of problem decomposition also describes the UMS inference engine strategy of trying to map a set of instruction-set nodes to hardware. The microcode synthesizer is given the goal of creating a mapping for the single top-level graph controlflow node of the instruction-set description. This node has connections to a hierarchy of descendant controlflow nodes, each of which operationally requires a mapping subgoal. This is relatively simple problem decomposition, compared to the elaborate planning necessary to achieve the open-ended

goals of robot planning (see Sacerdoti [SAC77]). Chapter VI describes in detail the heuristics that create this mapping.

2.4.1.6 Efficiency analysis. The only "efficiency expert" (see [KANB81], [KAN79A], [KAN79B], [KAN77]) in UMS is the RUT, which directs search to find "locally optimum" code within the search space by using techniques that avoid combinatorial explosion while postponing choice. Because UMS is based in a different application area than the conventional domain of automatic programming research, much of the work on choice of data structures (such as single- or double-linked lists [KAN77], [BARS79]) using algebraic program analysis is not relevant. UMS does not try to understand deeply the algorithm implicit in the instruction-set description or provide a way to refine algorithms [KAN85], [KANN83] by representing them abstractly or canonically. As a result, UMS is affected by the way the instruction-set description is written. This in turn affects which neighborhoods of the design space are chosen for investigation by the heuristics of the UMS inference engine. Efficient search for globally optimum code is an open synthesis-research area and is beyond the scope of this thesis.

2.4.2 Search

Later chapters of this thesis discuss the UMS inference engine [POE84] and its microprogramming- and microarchitecture-inspired heuristics without reference to artificial intelligence search techniques. The artificial intelligence

origins of these heuristics are described here, with emphasis on the way UMS uses them to decrease search time. In particular, UMS tries to postpone decision making until it gathers information about as many related constraints as possible. This is sometimes called the least-commitment principle [STEABBBHS83]. Barstow reflects:

"Most work in automatic programming has focused primarily on the roles of deduction and programming knowledge. However, the rule played by knowledge of the task domain seems to be at least as important, both for the usability of an automatic programming system and for the feasibility of building one which works on non-trivial problems." [BARS83].

This bias is reflected in the final design of the UMS inference engine, which is many orders of magnitude more efficient than the initial design based on straightforward search.

To discuss search in this chapter, a simplified view of the UMS inference engine is used. From this point of view, the engine tries to match operations and operands of the instruction-set description to those of the hardware description for each code fragment. These are respectively the arcs and nodes of the controlflow and dataflow graphs discussed in the rest of the thesis. In the internal form used by UMS, the simple expression "cpage = PC<0:4>" from Figure 1.3 contains four operations (such as assignment and bitfield extraction nodes) with a total of 11 operands (such as the arcs for cpage and PC, two separate arcs make up a connection between two nodes).

The set of all possible operation and operand matches covers an extremely large state-space ([BARR81] page 21) of inference engine activity on a large database. Like many artificial intelligence problem domains, it is not economically feasible to use blind search with generate and test techniques. Many search techniques are discussed in the literature [NIL80], [BARR81], [RIC83], [PEA84] from a game-playing point of view; their aim is to maximize the score of one player while minimizing the score of the opponent. In UMS, similar techniques maximize inference engine productivity (the number of code fragments synthesized) while minimizing computational expense. This is like the use of search in theorem-proving programs, in which the state-space is so large that suboptimal solutions are readily accepted (this rules out strategies like A* [BARR81], [PEA84]). Heuristic methods are used by UMS [POE84] as a control strategy to guide the inference engine between the states most likely to perform useful work. Note that to reduce the state-space, the order in which code fragments of the instruction-set description are processed is arbitrarily fixed (to reflect top-down controlflow and source-code order, irrespective of subroutine calls).

2.4.2.1 Simultaneity. Because a code fragment typically contains multiple operations, each with potentially multiple operands, UMS match processing occurs on sets of operations and operands collectively called candidate sets. Satisfactory matching of each operation and operand requires

successful completion of many tests. All operations and operands of a candidate set must simultaneously (see [STEABBBHS83], [WAL77]) pass the tests to complete the processing of a code fragment. This requirement of simultaneous satisfaction of interrelated constraints makes the search both computationally expensive and its strategy difficult to classify. UMS uses a "common sense" approach to this problem; it solves carefully picked subproblems first while modifying previously solved subproblems to reflect newly discovered constraints [WAL77].

2.4.2.2 Scheduling interacting tests. This section compares the UMS approach to dealing with interacting factors to other artificial intelligence research. Since complete synthesis of a fragment is computationally expensive, the UMS inference engine tries to determine quickly when processing a set will be obviously unsuccessful. This strategy is necessary because UMS can trigger many expensive heuristics that may overcome a small number of local difficulties. If the required instruction-set operations cannot be found in the hardware, then any further expensive search for the same operations and operands would be wasteful (this is an example of informed search [BARR81]). For computational economy, expensive heuristics should be applied only when they are most likely to contribute to successful synthesis of a complete fragment.

UMS may perform some tests on each operation in isolation (these are generally the least computationally expensive); other tests may be performed only in relationship to other operations of the set (these are more expensive and are usually based on their corresponding operand types). Search is biased first toward satisfying operation search before testing operands, since operation matching is a coarse and efficient search filter.

Those operations that are likely to cause the most difficulty are chosen first for the operations of the candidate set. If one operation fails its tests, then it is removed from the candidate set and another operation is chosen. After one operation match fails, a new operation is chosen for the set. However, this new operation possibly may be for a different role. If previous synthesis experience has used successfully matched hardware operators of a given type, then these operators are picked as candidates first, and search cost is not uniform.

Basing synthesis search on operation type is an example of ordered or best-first search [BARR81], [PEA84]. The evaluation function for search orders operators that perform semantic transformations (such as addition or subtraction) for matching first, because they are absolutely necessary for synthesis of the fragment and are rare in hardware descriptions. Match failure here would immediately cause finding a new candidate set. Array or bitfield-access operators are of second priority. Last are the extremely common assignment operators. In some special cases, requiring assignment operators

to match may even be overlooked. This occurs when it is decided that costly special graph-manipulation techniques should be used.

It was observed early in the development of UMS that, although some operand tests are much more expensive than others, the operand match failure rate was almost uniform across operand type. Therefore, the inexpensive operand tests are performed first. In conjunction with the previously discussed heuristic, the UMS implementation processes first the easiest operands of the hardest operators. This is just one of many examples in which experience with the UMS problem domain has refined the inference engine design and significantly improved search speed.

It is not efficient to choose all operations before any operands are tested. One operation often is related to another through the shared use of an operand. Unification-based ([CLOM81], [NIL80]) operation search can be much more selective if this shared operand is part of the search pattern. Because it is less expensive to test one inexpensive operand than to search for one operation, UMS search order is based primarily on operand type and secondarily on operation type. UMS creates a list of operand tests for all operations of the candidate list, which begins with the least expensive. Before each operand test is performed, a check is made to see if all associated operations have been chosen. If the match for any associated operations is unspecified, candidates are chosen before the operand test is performed.

After each hardware operation is chosen to match part of the instruction-set description, two levels of tests are performed. Ordered or best-first search [BARR81], [PEA84] is implemented directly in these two inference engine test levels. The first test level checks a candidate operation without reference to any other operations of the candidate set. This mainly determines suitability of the operator and takes place immediately after it is chosen. If the candidate is still suitable, then a second level of tests determines if the choice is compatible with all operators already existing in the candidate set. Note that at this point the candidate set may be only partly specified. These second-level tests mainly determine suitability of operands, and the schedule of their tests is intermixed with tests for other operators.

Many test categories exist within the second-test level; this stratification is based on computational cost to implement a search agenda [RIC83]. Multiple test categories determine alternate ways the same operand may be suitable for matching. Also different types of operands are associated with different kinds of applicable tests. If an inexpensive operand test fails, then a more expensive test might be performed later. If the inexpensive operand test succeeds, then no other tests are performed on that operand. However, if the last test for a specific operand fails, then the entire candidate set fails, and a new set is chosen.

When a transition occurs between processing of different test types, the success rate is measured by UMS (since failure of inexpensive tests tentatively schedules more expensive tests for later). Even though some operators and operands may have been successfully matched but the success rate is below a threshold value, the candidate set collectively fails and a new set is generated. This is an example of partially expanded or partially developed search [BARR81], even through the evaluation function may prune essential parts of the search space. The success-rate threshold increases at each transition (for more expensive tests). This is reminiscent of the cutoff values of Alpha-Beta pruning [BARR81], and implements an agenda [RIC83] of carefully directed search. Each threshold rate was empirically determined, and indeed "It often requires some experimentation to get it right." ([RIC83], page 86). These thresholds are expected to be valid across many examples of the same general type. Microcode for radically different problem domains, such as numerical analysis or FFT, perhaps would require threshold modifications.

As discussed so far, a large number of tests must be satisfied simultaneously, and the test schedule is based first on easiest operand type, then by increasing degree of interaction with other operators, then on operator type with decreasing test failure probability, then best successful synthesis experience with specific hardware operators, and last by success threshold. Only after applying this medley of problem-reduction transformations [BARR81]

applied to the original set-matching problem does each more primitive subproblem require exhaustive uninformed or random search [NIL80].

Even with involvement of all the heuristics just described, the search is unsuccessful sometimes. During the tests described above, records are kept of which candidate set was the most successful and which operation and operator caused the failure in the best search. Search resumes after some semantics-preserving transformations replace some of the original instruction-set operations with a new set. Only those transformations that modify the failure-causing operator and operand are applied in the programming-knowledge base. Search initially was attempted without use of transformations. After each failure to find a candidate set, one more transformation application is allowed. This strategy implements breadth-first search within a given number of transformation applications, although once a candidate set is picked, the search is depth-first.

Expert systems with interacting subproblems are among the most difficult design problems in artificial intelligence [STEABBBHS83] and require making decisions that impose the fewest arbitrary constraints. These difficulties are summarized:

"Reasoning based on the least commitment principle requires the following abilities: • The ability to know when there is enough information to make a decision • The ability to suspend problem-solving activity on a subproblem when information is not available • The ability to move

between subproblems, restarting work as information becomes available • The ability to combine information from different subproblems..." ([STEABBBHS83] page 107).

In UMS, the test for operator membership in the candidate set satisfies the first criteria. The suspension of simple operand matching (based on inexpensive information access while satisfying a cost threshold) and the potential matching of the same operands later (using more expensive to obtain information) represents the next two characteristics. The interaction of operators and operands in the candidate set is an example of the fourth feature.

The preceding discussion shows that the UMS inference engine uses a variety of artificial intelligence search techniques to find a practical solution to a set of interacting subproblems in the microcode synthesis problem domain.

2.4.3 Knuth-Bendix systems

Although not a direct part of this thesis, the UMS approach to microcode synthesis is influenced by the Knuth-Bendix algorithm [KNUB70] and the research that it triggered (such as [DER85], see [HUEO80], [HUL80], [PETS81] for a review). The Handbook of Artificial Intelligence [COHF82] describes the Knuth-Bendix algorithm as an efficient theorem-proving heuristic. It analyzes a set of directional transformation rules for completeness and consistency.

Knowledge-base completeness and consistency are important prerequisites for efficient rule-based systems [STEABBBHS82]. The algorithm's key idea is to measure with a single metric every sentence in the

transformation grammar. The transformations are designed so that according to the metric, each transformation results in a less-expensive sentence. A grammar of transformations is complete if there is a complete lattice (similar to mathematical duals) for a set of transformation applications. A grammar of transformations is consistent if there is no way to generate sentences that contradict each other from the same starting sentence. The key idea from this research is that after exhaustively applying a set of transformations in the Knuth-Bendix method, it can then be proven mathematically that the results of the transformation are the least expensive possible.

Instead of performing the complete exhaustive search of the Knuth-Bendix algorithm, UMS performs exhaustive search only to a specific depth. Proposed future research would preprocess the transformation set according to the Knuth-Bendix algorithm to decompose the effects of any overlapping rules and complete some gaps in the knowledge base [DER85]. The metric used would be the Resource Utilization Template, described in Chapter IV and [POE81B]. This metric is unique because it takes into account both the structure of the microword and the variability of microoperation placement. Thus, the synthesizer could find the provably best microcode (as constrained by a knowledge base) for a bounded amount of computational effort. This bound could be set for exhaustive search if needed.

2.5 Early Microcode Synthesis

This section discusses how early microcode-synthesis efforts not covered earlier in the chapter relate to the design decisions used in UMS.

2.5.1 MIXER

Previous work with the MIXER system [SHI83] has shown that a manually guided transformation-based system can produce microcode. MIXER is guided by three types of transformation rules (called macroknowledge, semantic knowledge, and microknowledge) manually supplied by a user for a specific microengine and microprogram. These explicitly describe translating the source microprogram into microcode. Whereas MIXER is designed for use with just one microengine, UMS is retargetable and automatically discovers the same types knowledge by inspection of the hardware description.

2.5.2 Mueller microcode synthesis

An earlier thesis by Mueller [MUE80A] (also discussed in [MUE80B]), published well after this thesis was under way, is the theoretical work most similar to this thesis. The research is based on direct matching of hardware microoperations with similar fragments of the instruction-set description. Extensive experience developing the UMS system [POE84] has shown that a synthesis-system design based only on preliminary design intuitions is not sufficient to achieve results on real examples. This system was intended to

achieve the same goals as UMS. This section discusses many aspects of its design and use. The next section compares how this work differs from UMS.

2.5.2.1 Test microengine. As a test example, the author of the synthesis system designed an idealized microengine with highly regular architecture (a "real" microengine was not used). Issues such as commutativity, operand routing, and temporary variable creation, which make synthesis search more difficult, are not addressed by this testing methodology to a degree necessary for practical tool building.

2.5.2.2 Variable bindings. To support instruction set to hardware variable bindings in this earlier research:

"manual assignment through common identifiers in both the instruction set requirement specification and host microprogrammable machine specification is assumed..." [MUE80A]. This requires not only close cooperation between the authors of each description but results in an unnatural programming style. Consider using common variable names to bind the instruction-set program counter to a member of a hardware-register array. The common variable name between the two descriptions prevents binding other elements of the hardware-register array to other instruction-set description variables.

2.5.2.3 Dataflow analysis. The author of a hardware description is required to make a distinction manually between state and nonstate variables (for instance, registers that can permanently store values). When an

assignment is made to a nonstate variable, the author of the source code is required to use a special ":=:" operator. Variables that store values for intermediate intervals of time are not considered.

2.5.2.4 Controlflow analysis. This research approach treats controlflow and dataflow separately, and they are represented in different types of tables. Hardware controlflow-sequencing operations are considered "typically simple and limited in number" [MUE80A] and specify a set in common between the instruction set and hardware.

"Each basic [instruction set] specification is either a microinstruction word encoding resulting from the direct translation of a control sequencing operation, or a functional specification which dictates a desired state transformation." [MUE80A].

This use of common control primitives between the two descriptions reduces the synthesis search difficulty to a relatively simple level. However, it is unclear whether all common control structures could be processed by the set supported by this research. Unfortunately, multiway branches, span-dependent jumps, and subroutines are not discussed.

A recognized simplification in this research is the assignment of microoperations to sequential time steps at description-translation time, instead of a design that directly supports microword compaction for horizontal microarchitectures. Because the Mueller research makes simplifying assumptions regarding analysis of ISPS sequential/parallel sequencing

operators, only very simple hardware timing design is mentioned. Since the assumption:

"The microprogram counter register initially addresses the first micro-instruction of any microprogram satisfying the symbolic assertion, and following the execution of such a microprogram, that the microprogram counter register addresses the micro-instruction which sequentially follows the end of that microprogram..." [MUE80A]

is made, the research does not address many common controlflow-synthesis problems. Note that controlflow in microengines has much more complex semantics [DAS79], [HASJ79], [ADA78], [ARM77], [SAL76], [HUS70], and any operations on data in horizontal microcode must consider datapath and resource contention [VEG82B], [TOKTTT77]. A mechanism such as described in [MUE80A]:

"At the start of each microinstruction synthesis phase, the micro-instruction register field(s) which determine the control sequencing function are initialized to the encoding pertaining to sequential execution..."

may not be compatible with some optimal microinstruction encodings, and is likely to sacrifice code quality.

2.5.2.5 Semantic analysis. The semantics used by a description is required to belong to a fixed set of idealized microoperations. This set may not be complete or appropriate for commonly used microengines. Extensions to the system for instruction-set semantics (either controlflow or dataflow, beyond

those available directly in the hardware may not be possible. When multiple microoperations perform the same operation in the microengine, only a single representative is included in the synthesis process-tables. In contrast, most microengines contain multiple ways to perform the same operation. It is unclear whether an operation such as "X = X+0" would be considered a redundant state transformation or not.

2.5.2.6 Code representation. Mueller's research is based on the popular "weakest precondition" theoretical paradigm (see [BAK80] for a tutorial), which specifies code-fragment semantics by description of the "before" and "after" states of any affected variables. Specifically, given a computation description, the weakest preconditions specify any necessary control or data requirements. Both the instruction-set and hardware descriptions are processed into tables of weakest preconditions (also see [MUEV82]) before performing matching to produce microcode. Subroutines are not discussed. The work on UMS has shown that this level of description granularity is too coarse for many microcode optimizations.

2.5.2.7 Search strategies. "Breadth first," "depth first," and other simple search techniques are discussed to match instruction-set fragments to the hardware description tables. Search is either completely successful or fails; partial successes are not analyzed and continued.

2.5.3 Advantages of UMS over traditional weakest-precondition based microcode synthesis

The most important distinction between UMS and traditional weakest-precondition based approaches is UMS has evolved from actual experience. Barstow argues:

"There is a growing belief in the software engineering community that formalization in the absence of implementation is not possible because of the difficulty in formalizing the right set of requirements." [BARS85C].

Perhaps the most important evaluation of previous approach research comes from one author:

"A working implementation of a basic automated microprogram synthesis system as proposed herein is suggested as the primary sequel to this work, to uncover weaknesses..." [MUE80A].

This section begins such an evaluation.

2.5.3.1 Test microengine. The microengine used to test UMS is based on a real bitslice chip architecture with typical irregularities. As a test of the power to use UMS with arbitrary hardware, this test microengine was designed and published by a different author prior to the development of UMS. Work previous to UMS contained simplified microengines designed by the authors of the prototype microcode tools.

2.5.3.2 Variable bindings. Instead of the approach taken in Mueller's work, which requires a user to determine the bindings between hardware and

instruction-set variables manually, UMS performs this function automatically. Chapters I and VII even describe examples with conditions necessary to change previous bindings when additional constraints are discovered.

2.5.3.3 Dataflow analysis. Major parts of Chapters IV and VI discuss how complex controlflow in real descriptions is, and how UMS can process it effectively. In contrast to earlier synthesis research, UMS uses the analysis techniques to automatically determine the valid lifetimes of variable values (state, nonstate, or intermediate). The inference engine of UMS [POE84] is based on these analyzed variable value lifetimes.

2.5.3.4 Controlflow analysis. UMS represents and processes controlflow and dataflow dependencies in similar ways, in contrast to earlier research. Instead of representing hardware controlflow in a simplified and idealized way (incurring the problems of artificial machine models, described earlier in this chapter), UMS bases synthesis on whatever controlflow techniques the hardware description uses. In an example described in Chapter VI, rules from a programming-trick knowledge base bridges a mismatch of semantic level between the instruction-set and hardware descriptions. This is achieved by modifying dataflow to overcome a controlflow mismatch between hardware and the instruction set. This author's experience with real microcode and microarchitectures has found these mismatches (not discussed in earlier

research) to be common. UMS is designed to support microcode compaction as well as multiway branches, span-dependent jumps, and subroutines.

The description techniques described in Chapter IV allow using hardware with asynchronous- and variable-timing constraints with UMS. UMS can detect the microprogram counter and, when called for, can appropriately synthesize microcode to manipulate it. In particular, sequential controlflow execution is not required. Earlier research has required the user to point out the microprogram counter and use it in restricted ways.

2.5.3.5 Semantic analysis. The graph-oriented structure of code within UMS does not require determining where one microoperation stops and another begins (this has traditionally been an important research issue see [MUE80A], [OAK79] and discussion earlier in this chapter). The knowledge base of UMS allows translation from instruction-set semantics that are not directly available in hardware to sets of operations the microengine can perform. When multiple microoperations perform the same operation in the microengine, each one can be considered in the synthesis process. Since typical microengines have irregular architectures, only a subset of all similar microoperations can actually be used in a specific microcode context. The ability to choose efficiently between two equivalent operations (such as "A+B" and "B+A") was found with experience to be so important (initially, it was a program-transformation rule in UMS) that it was directly implemented within the

inference engine (and called operand commutativity). The example in Chapter VI uses the code transformation "X = X+0" to route a (Boolean) data value through the ALU when a more direct technique is not possible. These types of synthesis flexibility have not appeared elsewhere in the literature.

2.5.3.6 Code representation. UMS is based on the program-transformation paradigm in which the instruction-set description is repeatedly modified until it becomes microcode. Code is represented in UMS as a control-and-data-dependency graph that implicitly contains understanding of subroutine calls and returns. Code is initially synthesized as if subroutines are expanded in line. Optionally, subroutines could be created out of commonly used microcode sequences. Subroutines have seldom been discussed elsewhere in the microcode synthesis literature.

2.5.3.7 Search strategies. Search in UMS is based on carefully scheduling the solution of subproblems. The reasons for failure within a partially successful search are analyzed in UMS to guide problem (and program) modification. These search techniques compare favorably with other advanced techniques found in the artificial intelligence literature.

These last two sections have described many differences between microcode synthesis systems, such as Mueller's, and the UMS design. The UMS design experience suggests that the issues described in this chapter are of critical importance to successful microcode synthesis.

CHAPTER III

DESIGN OF A RETARGETABLE MICROCODE COMPILER TO USE SYNTHESIS TECHNIQUES

3.1 The V-Compiler

This chapter describes a retargetable microcode compiler called the V-Compiler (the V is for Virtual). The V-Compiler development at Digital Equipment Corporation was a team effort (which included this author [PATGPS81]) that was canceled before completion. This research was intended to derive automatically code generation and other required tables, based on analysis of a machine description. After project cancellation, the author implemented and extended the V-Compiler ideas to develop UMS, described in the remainder of this thesis. Experience gained in the V-Compiler project inspired many of the ideas described in this thesis.

The V-Compiler is table driven (the tables are in the form of productions [DAVK77]). Each table entry described one irrevocable step in the compilation process. This chapter describes how the V-Compiler ideas, which are based on more familiar compiler technology, have evolved to become the UMS design. Such a comparison assists understanding how synthesis technology differs from traditional compilation techniques.

3.2 The Compilation-Synthesis Spectrum

This thesis views compilation and synthesis as different code-translation methodologies. Compilation conventionally transforms input source-code fragments into object code in a fixed and predefined way. Any analysis of the microengine or input source-code semantics required for the transformation is performed prior to compiler implementation. In synthesis, this transformation is not specified ahead of time and is based on search. Further, previously specified knowledge fragments may be assembled in unexpected ways (not counting bugs) during synthesis. In other words, compilers are based primarily on algorithmically understood techniques, whereas UMS depends on heuristic search.

The intended use of microcode suggests which microprogram-development methodology is appropriate. During evolution of the microengine design, it is more appropriate to synthesize microcode for each new version of the microengine. When new user instructions are being designed for a fixed and completely understood microengine, compilation is appropriate for the translation to microcode. In practice, the extremes of this spectrum are seldom observed. Register-allocation algorithms and peephole-optimization techniques in compilers (see [AHOU77], [WULFWGH75], [GRI71]) apply some context information and a small amount of algorithmically guided search to

modify selectively the transformation of all but the most trivial programs. In the programming knowledge base of UMS there is only one relevant transformation for some high-level source constructs, so that its use can be predicted and little search is appropriate. Nevertheless, this spectrum is useful for understanding the relative degrees of search or determinancy in UMS behavior and how it differs from conventional compiler technology.

In practice, UMS with both compilation and synthesis techniques creates microcode. Specifically, UMS can use information obtained from previous synthesis effort to either reduce search or avoid it altogether. The most obvious example is the catalog, which can store the "before" and "after" code configurations that have undergone transformations. At a more subtle level, when the inference engine tries to find a match for instruction-set code, it first tries those similar parts of the hardware that were used successfully in previous matches.

3.3 Design Goals for Automated Microcode Development

The motivation for creating the V-Compiler was to decrease programmer-time costs (and therefore time to market) for developing microcoded emulators of CPU families with large instruction sets. It was considered necessary to be able to produce code for large microprograms

within 15% of the size and speed of microcode produced by hand. Patterson [PAT76] demonstrated that machine-dependent, high-level microprogramming languages can produce efficient microcode. This goal was also achieved by the V-Compiler. Chapter VII shows UMS achieving this performance goal for machine-independent code.

Microcode should be created in a form that is easy to write and maintain. This goal was not satisfied by the V-Compiler. In contrast, UMS achieves this goal with the same ISPS language ([BARS82], [BAR79], [PARTCC79], [BARBCS78]) description used by engineers to determine the behavior of the instruction set.

The long-term goal of the V-Compiler was to reduce compiler retargeting effort, in part by using transportable microcode, to one month for a CPU. Experience with transportable BLISS programs at Digital indicated that highly efficient programs can be transported, although they must be written more carefully than nontransportable programs. Unfortunately, neither the V-Compiler nor UMS was used with a large instruction set. However, the example in Chapter VII shows that UMS does not require retargeting (other than the nonrecurring effort required to augment the programming knowledge base) to synthesize microcode for a small instruction set.

3.4 V-Compiler Components and their UMS Counterparts

Figure 3.1 shows the components of the intended V-Compiler, which is discussed in dataflow order. The design philosophy made each step or phase of the translation process correspond to one V-Compiler component. A shared common intermediate data language based on expression trees interconnected the components of the V-Compiler. This language consists of arithmetic logical and field operators, simple-assignment statements, conditional and unconditional jumps, IF-THEN, IF-THEN-ELSE, and WHILE statements. It also allows defining of code blocks and procedures. The literature review discusses the reason UMS uses graphs instead of expression trees to represent intermediate code.

The V-Compiler high-level language front end (shown in Figure 3.1) was never created; it was intended to be a compiler to translate a high-level language into an intermediate-level language. UMS contains a compiler to translate the ISPS language into the UMS internal graph-structured language.

Instead of implementing most of the V-Compiler as in-line code or using conventional compiler tables, productions were used extensively (see [NIL80] or [HAY84] for an introduction, or [DAVK77], [BARR81], or [COHF82] for more discussion). In the V-Compiler context, productions are interactively applied and described before and after states of phrases in the intermediate language.

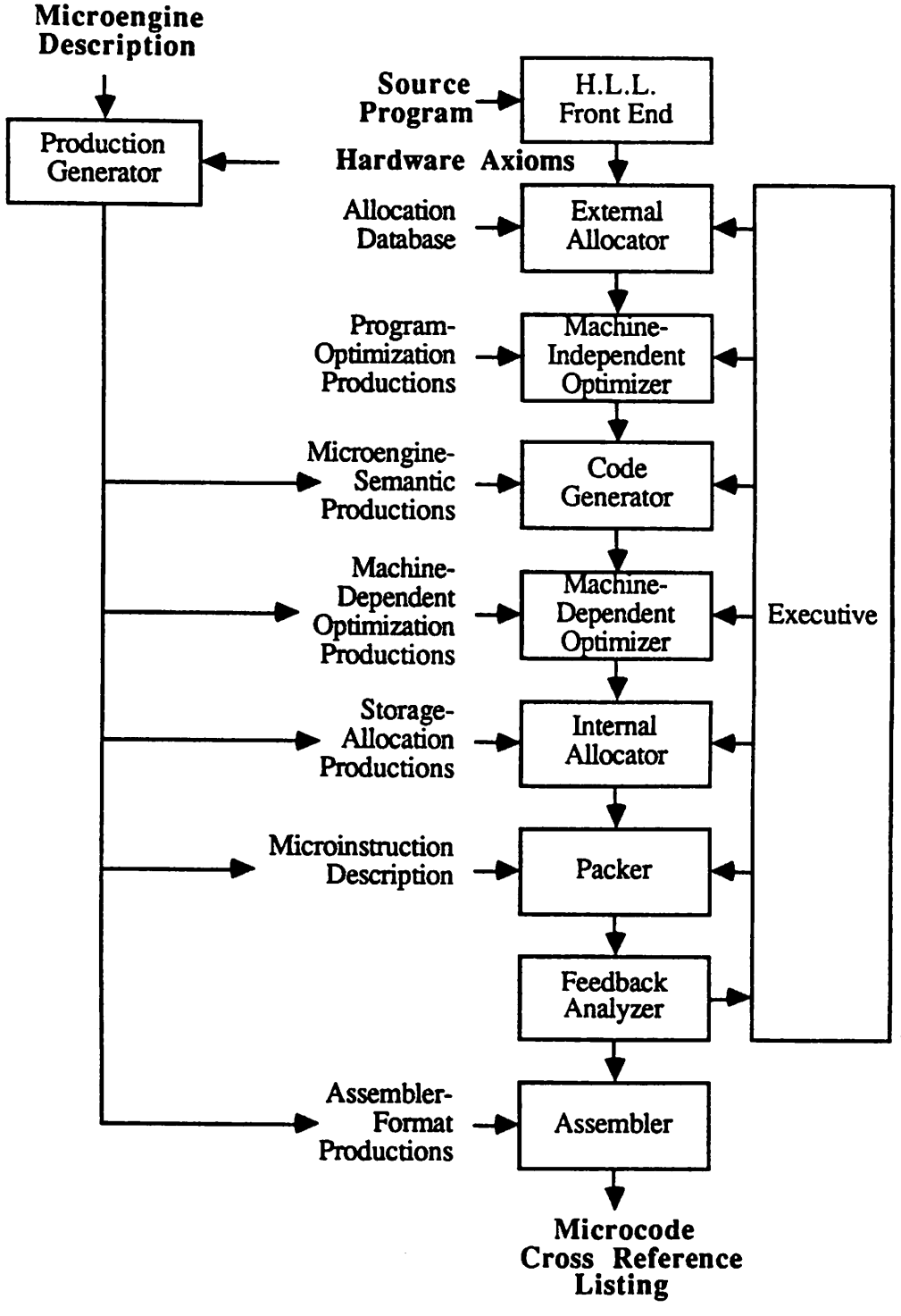


Figure 3.1: Components of the V-Compiler (from [PATGPS81]). This is the design of a microcode-compilation system that preceeded the design and implementation of UMS. The strategies of the two systems are compared in the text.

Similarly, UMS uses productions to describe transformations on the intermediate microcode representation.

The external allocator was needed, but never created for the V-Compiler, to allocate resources to the external variables of separately compiled microroutines. In UMS, resource allocation based on the RUT and global control-and-data-dependency analysis is intended to perform this function for the complete microprogram. At present, UMS does not support separate compilation or separate synthesis of code modules.

The machine-independent optimizer was intended to perform classic compiler optimizations, such as constant folding, removing common subexpressions, and eliminating chained jumps (described in [AHOU77], [WULFWGH75], [GRI71]). These were not implemented in the V-Compiler, but are part of the UMS design. With a small amount of extra implementation, the inference engine could determine where constant folding is appropriate. It would check to see if all inputs to an operation are already constants, then calculate the result. At present, UMS assumed that code processed is clean enough to not require this optimization. Removing common subexpressions is already part of the UMS design; this occurs when the RUT allocates microoperations with the same arguments and control-and-data dependencies to the same microwords. Chained jumps, considered to be an artifact in conventional compilers, do not occur in UMS. Chained jumps occur when local

context only is considered during code generation or peephole optimization; a jump is created to an address that later processing will fill with another jump. It is assumed that UMS's instruction-set description code, which is at a high semantic level, would not contain obvious inefficiencies such as these, and the UMS synthesis process does not create them. If these inefficiencies were present in the input code, then UMS could be augmented to correct them.

The next V-Compiler component in Figure 3.1 is the code generator, which receives the intermediate language and translates it into microoperations using microengine specific productions. These productions were created by hand before using the V-Compiler and remain unchanged during its operation. To reduce the number of production applications per code fragment, the level of semantic granularity in the V-Compiler productions is as large as possible. This is feasible because these productions implicitly contain knowledge about the target microengine. In contrast, UMS has finer semantic-transformation steps in its knowledge base than is typical in V-Compiler code generation. Initially, UMS requires multiple transformations to synthesize microcode, but to improve efficiency, UMS stores the results of these transformation sequences in the catalog.

The machine-dependent optimizer takes advantage of microarchitecture idioms to produce better code (for example, incrementing a counter rather than using an adder). This is similar to peephole optimization, but was not

implemented in the V-Compiler. Such a component is needed in the V-Compiler because the V-Compiler's code-generation process does not consider context. Code generation in UMS, but not in the V-Compiler, could be affected by neighboring control or data dependencies or microword-field availability in the target microwords. The UMS code-generation and compaction techniques do take these issues into account, so this type of optimization is not needed.

The V-Compiler's internal allocator (Figure 3.1) function, which was intended to bind any local variables to fixed registers, was not implemented in UMS. UMS views register allocation and microword-field allocation to be essentially the same problem; these are addressed with the RUT.

An initial version of the packer component of Figure 3.1 was implemented by this author in 35,000 lines of Bliss code. The packer first reads the microoperation sequences to form a control-and-data-dependency graph. This graph is used to schedule parallel microoperations into microinstructions as constrained by control and data dependencies.

This packer differed from the w y cited approach by Fisher [FIS79], which first packs complete execution traces through the microprogram. The V-Compiler method [POE80], [POE81A] packs blocks of code in order of global execution frequency. This packer carefully weighed microoperation placement tradeoffs for the multiway branches common in microcode. The packer could

reverse the order of controlflow fork/join pairs, although the importance of this optimization was not measured. The Fisher approach might move an operation from the trunk of a control join into every branch if such a movement would speed up the most commonly used path. For example, if the most commonly used path was used 40% of the time and the three remaining paths were used 20% of the time, the movement described by Fisher would be performed. Unfortunately, the movement could slow down each of the lesser-used paths, although they collectively represent a majority (60%) of the execution time. The V-Compiler packer did not make this type of judgment error.

A problem of most microcode-improvement systems is facilitating microarchitecture-independent compaction. The V-Compiler approach was to require writing two custom subroutines called FIT and COMBINE during retargeting. FIT tests to see if a microoperation may be placed into a microword. COMBINE modifies the representation of a microword contents to include the new microoperation. This approach allowed machine-independent design of the packer without introduction of an elaborate notation. These subroutines are not needed in UMS, where the RUT is intended to perform compaction based on analysis of the instruction-set and hardware descriptions.

The next component of the V-Compiler was the unimplemented analyzer. This analyzer digests information from the functions above and feeds it back to the (unimplemented) executive in the form of directives for a more

optimal compilation of the same code. Given the design of the V-Compiler, this was the most direct feedback path possible. In contrast, UMS has much shorter and more direct feedback paths. For example, if the hardware allocation for a global variable or a code-generation technique was infeasible or inefficient, then the resource analysis of the UMS RUT could potentially trigger an immediate search for alternatives. In contrast, the V-Compiler would wait until compilation completion before responding. If the V-Compiler was able to use more direct feedback, as does UMS, it would prevent potentially very expensive machine dependent-optimization and packing (the subsequent stages of compilation) of otherwise poor code.

The conceptually simple assembler of the V-Compiler takes the intermediate code form of the microinstructions and the format-production table to create the final microinstructions. It also links microaddresses from separately compiled microroutines. At this point, UMS does not support a microassembler format output; the research-level implementation allows comparing the microword field values directly to only existing microcode. UMS also does not support separate compilation; the complete instruction-set description is synthesized at one time. The V-Compiler was designed to work with an existing microassembler, which allocated microinstruction addresses based on span-dependent jump constraints ([LEV80], [ROBE79A], [WILL79],

[SZY78], [FRIS76]). UMS currently does not address these issues, but could process them similarly at a later date.

The V-Compiler production generator, which was intended to run at compiler-creation time, is in the upper left corner of Figure 3.1. It was conceived of as reading a formal machine description and using a set of axioms to derive the machine-dependent productions for the compiler. The key interest of this author is synthesis of microcode based on information extracted from a machine description. As such, the configuration of the system changes when the emphasis switched from relying on an existing compiler to being centered around synthesis. Furthermore, because this author could not rely on use of the V-Compiler, much of its functionality was duplicated in the creation of UMS.

Similarly, the parameter setting and reprocessing functions originally considered for control by the V-Compiler executive are distributed throughout UMS for prompt feedback. Chapters V and VI discuss how UMS carefully measures the amount of computational effort expended to solve specific synthesis problems and determines whether the current branch of the search tree should be continued, suspended, or abandoned. Ultimately, the V-Compiler considered many fewer code-generation alternatives than UMS, which is to be expected from its compiler technology genealogy.

CHAPTER IV

ROUTING AND PARALLELISM ANALYSIS OF MACHINE DESCRIPTIONS

4.1 Introduction

UMS requires two descriptions before it can synthesize microcode: 1) A description of the desired instruction set to be realized in microcode and 2) A description of the target hardware (see [LIP78]) that will run the microcode. In UMS, these descriptions are programs (written in the ISPS language [BARN78], [BAR79B], [BAR79C], [BARBCS78], [BAR79], [PARTCC79], [BARS82]) used as input data to UMS. This chapter discusses techniques to specify common microprogram and hardware concepts in these descriptions for later exploitation by the synthesis process.

Little of the information needed for synthesis is explicit in either the hardware or instruction-set descriptions when they are written in the procedural description style. Before synthesis can begin and proceed efficiently, a great deal of processing must be performed to extract and make explicit meaning from the descriptions.

The four topics discussed in this chapter are: 1) the machine descriptions used as input for UMS, 2) the compiler technology used to extract the

description meanings, 3) interpretation of the description semantics, and 4) the UMS internal representation of machine descriptions.

4.2 Machine Descriptions

This section discusses syntactic and semantic machine (both instruction set and hardware) description techniques. Subsequent sections describe how UMS analyzes, internally stores, and uses these descriptions.

4.2.1 Machine-description levels

Machines can be described at many different levels of detail. An influential book on computer design [SIEBN82] (which is a revised version of [BELN71]) suggests a nine-level hierarchy of computer description: electronic circuit (at the lowest level), combinatorial logic, sequential logic, datapath register transfer, control register transfer, instruction set, operating system, run-time routines, and application programs (at the highest level). The instruction-set description used by UMS is represented by the instruction-set level and a microarchitecture is represented by combining the datapath and control register-transfer levels. Later discussion describes the meaning of each description level used by UMS.

The literature review describes related synthesis work at lower levels (such as described in [LIP83], [SHI83], [SHI79]) or higher levels (for an

example, see [BARS80A], [BARS79], [DAR78]). One published example (see [PTSBHLK79]) bridged more levels than described in this thesis, but by using very different heuristics. That research addressed issues different from this thesis; it synthesized only a portion of a complete computer implementation without approaching the issue of microprogramming. Although synthesis is in general a hard problem, limiting the range of detail that synthesis must bridge is one of the reasons why it is successful here.

4.2.2 Description style

This section discusses how description style affects the way a description can be analyzed. The semantics implied by the instruction-set description code must be understood completely, otherwise the duplication of the functionality in microcode may be erroneous. One code statement may affect the actions of another code statement through data dependencies; the data value written by one statement is "depended on" when read by another statement. Similarly, one statement (such as an "IF" statement) can control the action of other statements (such as the "THEN" or "ELSE" parts) using control dependencies.

It is quite important to base the description on structured controlflow (no "GOTOs") that allows easy and complete analysis of the control-and-data dependencies. Without structured controlflow, analysis by the highly conservative and computationally expensive "case splitting" (see [HEC77]) would be required. Case splitting precludes many types of microoperation

movements described in the literature (see [LANDSM80] for review). Recursion is also not supported by the UMS controlflow analyzer. If UMS needed to support recursion in an instruction-set description, then code-transformation techniques [BURD77] could derive iterative code from many types of recursive controlflow. Adherence to these guidelines is not only good programming practice but also ensures proper UMS description analysis.

4.2.3 Instruction-set description

The instruction-set description describes the functionality of the computer from the point of view of the human assembly-language programmer. There are two parts of the instruction-set description: state and program execution semantics. This section describes how both of the parts are represented in an instruction-set description.

In the ISPS language used by UMS, state information such as specialized or general-purpose registers is specified as variable declarations at the beginning of the instruction-set simulation program. The characteristics of the primary memory array, such as the number of bits per word (width) and the number of words, are similarly specified.

The instruction-set description is intended to describe unambiguously the semantics of the computer architecture. The difference in semantic levels between the instruction-set description and the target microengine may be arbitrarily large if no ambiguity is present. For example, the instruction-set

description may describe floating-point operations, character-string manipulations, or even parts of the operating system that manipulates complex data structures. Most types of commonly used semantics, such as floating-point operations, may be described in terms of much simpler operations (such as integer additions, subtractions, and shifts). If floating-point operations are unspecified in the instruction-set description, then definitions from the knowledge base of microcode semantics could be substituted. Other types of semantics, such as memory management or interrupt processing are less standardized and techniques used to describe them will be presented next.

4.2.3.1 Memory management. For those computer architectures with virtual memory completely processed by the hardware, mention of memory management (see [SHA74] or [GRA75]) is unnecessary in the instruction-set description. This, however, is unlikely, and for most of the newer computer architectures (particularly those with 32-bit wide memories and addresses) some memory-management processing occurs in microcode. In this case, the instruction-set description must refer to part of the hardware description; this may be as minimal as knowledge about a special register, flag, or microword field, or knowledge that a special memory-management routine address is located in a special memory location. The PDP-8 example in Chapter VII lacks virtual memory. This section describes an approach to employ UMS for virtual-memory-based computers.

The instruction-set description for any computer describes explicitly the processor's fetch-execute loop (shown in Figure 1.2) along with the reads and writes of the input and output operands. Typically, a test (sometimes called a memory probe) executes before an instruction or operand is read or written. This test determines whether the relevant fragment of virtual memory is present or not in physical memory. If the virtual-memory fragment is not present, an exception called a page fault usually occurs. By employing structured programming techniques and subroutine calls, the portion of the instruction-set description involved in probing memory and reading or writing operands is typically very small compared to the rest of the microcode.

In the design of some of the newer 32-bit microcomputer chips, an optional chip (MMU, for Memory-Management Unit) performs the virtual-to-physical memory mapping. Without the MMU unit, the processor ignores the values of some of the address bits and directly accesses primary memory. If the MMU is present in the system when the virtual-memory data is present in primary memory, the MMU transparently processes memory traffic. If the virtual-memory image is not present, then the MMU triggers the previously mentioned page-fault exception. The description of an interrupt or exception in the hardware and instruction-set descriptions is discussed after pseudomicroword fields.

4.2.3.2 Use of pseudomicroword fields. Techniques described in [POE81A] allow using a pseudomicroword field to make explicit a relationship between the instruction-set description and the hardware description (such as information about memory management described previously). A pseudomicroword field is a portion of the value of the microcode in the hardware description that does not actually exist in the hardware. However, its value can be declared as unspecifiable or it can contain a value used during the synthesis of each microword. Before synthesis of microcode for a virtual-memory-based computer begins, a declaration is made to the inference engine that a specific instruction-set variable maps to a specifiable hardware pseudomicroword field. In the instruction-set description, ISPS source-code statements could be inserted before accessing a memory operand; insertion at this point would explicitly set the variable of the mapped pseudomicroword field to "true" instead of the default "false". This use of a pseudomicroword field would make explicit the implicit hardware actions of a memory access. The use of pseudomicroword fields require very minor changes to existing descriptions.

Pseudomicroword fields can be used to represent the nondeterminacy of some types of data processing. In the hardware description, when the pseudomicroword-field value is true, the hardware performs the required probe to memory and a conditional branch based on the probe outcome. If the virtual-memory fragment is present, then processing continues normally. If the

virtual-memory fragment is not present, then the hardware description describes the READ of the dedicated memory address, which points to the operating-system page-fault-exception handler and the subsequent unconditional branch. This process is similar to the nondeterminant interrupt processing described later. This use of the pseudomicroword fields allows implicit control and data dependencies to be expressed explicitly so they may be processed by UMS.

4.2.3.3 Interrupts, exceptions, restartable instructions, and resets. An important characteristic of a CPU's or operating system's performance is its interrupt processing capabilities. Although computer interrupts occur automatically from the assembly language programmer's point of view, since their processing affects the microengine, their processing must be made explicit in the microcode.

For simple instruction sets, interrupt processing is not a problem at the instruction-set description level. Here every instruction completes in the span of a few memory cycles. Very few simple microengines can be interrupted while running microcode; the microengine usually must test explicitly the value of a special interrupt-pending register before, during, or after the assembly-language instruction execution. After an interrupt is noticed by the microengine, microcode initially processes it to detect what type of interrupt has occurred, and a machine language routine processes the interrupt for the

specific interrupting device. The connection between interrupt semantics of the instruction-set description and the hardware can be made explicitly with the pseudomicroword-field technique described earlier.

When more-complex instruction sets are designed, each instruction execution may take an arbitrary (and potentially lengthy) amount of time. For example, the Digital Equipment VAX can move blocks of data up to 64K bytes long (which would require 16K memory cycles of a 4 byte wide data bus) or perform polynomial-series evaluation. To ensure acceptable interrupt performance, lengthy instructions may be interrupted and then restarted. The measure of the shortest amount of time possible between two consecutive interrupts is called the processor's interrupt latency. The following discussion describes the design of code to accomplish efficient interrupt processing and restarting of instructions.

It is impractical to restart most instructions from the beginning, particularly such lengthy instructions as the 64K-byte moves mentioned earlier. If several interrupts were processed during a period of time and such an expensive instruction were restarted from the beginning, the CPU's net instruction execution speed would be very low. Instead, when lengthy instructions are interrupted, they typically save enough state to resume processing. If the instruction is to be restarted, then careful analysis must determine: 1) What values in the current portion of the microprogram represent its state other than

the current microprogram and must be stored, 2) What parts of the microengine represent its state and should not be modified and 3) What parts of the microengine or process will not be modified during any external sequence of events and thus may be used to store the current state of the microprogram. The state is often stored in general-purpose registers or memory locations that are output operands of the instruction. Ensuring that these values are not modified, or if modified returned to their original state, requires verification (see [MARCL84], [CROMV80], [CARJB78], and [PAT76]) of the behavior of the operating system. Automatically ensuring that all these conditions are satisfied requires program-analysis techniques beyond the state of the art.

The general and complete solution to the interruptible-instruction problem is beyond the scope of this thesis, but a limited solution is presented later. Interruptible instructions are difficult to implement automatically if a maximum interrupt latency is specified. The synthesized microcode and the microengine must be analyzed very carefully to satisfy this design requirement. Special microoperations executed in the course of instruction emulation check for pending interrupts and perform a conditional branch to a microcoded interrupt handler. Determining where and how often these microoperations must be placed in each possible controlflow path requires careful controlflow analysis. Since microcode contains a high percentage of conditional branches, this analysis would be extremely complex.

Although it is not yet possible to verify correct interrupt processing, it is possible to design microcode based on unverified standards. It can be assumed that an operating system's interrupt handler would not invalidate the state of the general-purpose registers or the system stack within main memory. It also can be assumed that the assembler instructions within the interrupt handler do not corrupt the state of the microengine. The implementer of a new computer would quickly observe if such assumptions of correct implementation were invalid.

Most lengthy assembly language instructions contain an inner loop. When interrupt latencies are specified in the instruction-set description, tests for pending interrupts can be inserted in the microcoded loops. If the test determines that an interrupt was pending, then the state of the loop could be stored in the operands of general-purpose registers or main memory and the process suspended (see [SHA74] or [GRA75]). The interrupt handler would then process the interrupt, the process would resume, and the interrupted instruction restored and continued. To facilitate this type of interrupt handling, computer architectures such as the VAX have an "instruction not completed" bit within the processor-status word. When a process resumes, the microcode checks the value of this bit to determine whether its operands should be interpreted normally or as the intermediate state of microcode loop.

Although many computer architectures do not posses complex interrupt structures, most architectures do support some type of processor-state reset feature. Some similarity exists between reset processing and an interrupt. The special hardware-reset signal's function places the processor into a predefined internal state so that processing may begin. It is the responsibility of the microcode to set the state. Typically, the reset signal is given to the processor when its power is turned on. Even older computer architectures such as the PDP-8 have the reset function.

Using techniques described earlier, the hardware description's reset signal can be mapped manually at the beginning of synthesis to the reset variable of the instruction-set description. A reset is described in the instruction-set description by a small fragment of code that assigns fixed values to the processor's state variables. One or more tests within the main fetch-execute loop check to see if the reset signal is true. When the reset signal is true, then the instruction-set description simply branches to the code that sets the necessary values and continues processing based on those values.

The description techniques discussed in this section make it easy to describe how external events can affect instruction-set processing by a microengine.

4.2.4 Register-transfer hardware description

The datapath-and-control-register hardware description is mainly at the register-transfer [SIEBN82] level of detail. This functional description describes the routing of value movements and modifications of values between hardware registers with specific names. The description does not describe hardware-implementation technology; instead, it describes the microengine behavior for all potentially usable microword values. For example, details of memory-controller design are unimportant if the interface between the microengine and memory is well described. Similarly, if two different microword-field values can cause the microengine to perform integer addition, it is unnecessary to determine whether this is done on the same hardware or on two physically different adders. Thus, the microengine description must describe only the potential actions from the microword point of view, and any irrelevant details may be suppressed. This section describes how a microengine description specifies the semantics of commonly used hardware technology.

4.2.4.1 Tristate buses. Many examples of digital logic use tristate buses. This allows the hardware to transmit and receive information on the same bus. Whereas conventional digital logic has only two states, 0 (or low) and 1 (or high), tristate logic has an additional state: floating, which is shown in ISPS code as -1. Electrically, this means that the connection may "float"

between low or high. Any connection to the bus may transmit (write) a value of 0 or 1 by electrically forcing the connection to one of these values. Alternatively, the connection may receive (read) information by not forcing the connection to a particular value, instead sensing the value forced by other parts of the hardware. No ill effects occur when more than one connection to the bus tries to write the same value. Simultaneous attempts to force the bus to different values cause a "race" condition with an undefined outcome; this should never happen in properly designed hardware.

The following text discusses procedural hardware description techniques for tristate logic. The emphasis is on description, but simulation is also possible. The techniques are similar to those used in an event-driven simulator. Alice Parker's research group [PARW81], [ALP81], [HAPA81], [WAL79A], [WAL79B], [PAR75] has performed this type of research. During a bus cycle, one part of the hardware often writes a value on the bus that is read by another portion of the hardware; the hardware in turn uses this value to calculate a value written to the bus to be read by still another part of the hardware, and so forth. Well-designed hardware does not allow this cause-and-effect chain to go on indefinitely, instead it settles down to a fixed state in a finite period. For any valid set of control signals, the maximum amount of time (including an appropriate safety factor) required for this process is called the cycle time of the bus. To describe this behavior in the hardware, it

is necessary to express the idea that individual pieces of hardware may potentially respond in any time-sequence order and that one piece of hardware may affect any other during the cycle time of the bus.

Figure 4.1 shows the general code format for describing a tristate bus. Although the bus shown in this figure is only one bit wide, a wider bus follows the same general pattern. The label (line 3) and RESTART statements (line 23) form a loop, allowing multiple passes through a section of code (lines 4 to 22) that describes the behavior of all hardware (lines 7 to 12 and 16 to 21) connected to the bus. Multiple passes are necessary to allow the pieces of hardware to respond in any temporal order. A condition statement (lines 5 and 14) determines if each portion of the hardware is to respond during a pass through the code. The RESTART statement (line 23) determines whether the next pass should begin. This technique requires only a minor amount of additional programming effort.

At the beginning of each bus cycle, assignment statements assert every connection (bit) of the tristate bus to the value floating, represented by the -1 value in line 1 of Figure 4.1. Whether a piece of hardware responds during a specific pass through the code is determined by a function based on a pseudomicroword field, the pass number through the code, and time (see lines 5 and 14 of Figure 4.1). Only the pseudomicroword field is necessary to create the required data dependencies; the other variables could be used in a realistic

model of the bus behavior. The beginning of each code section (lines 7 to 12, and 16 to 21) contains a DECODE statement (lines 7 and 16) that is indexed on the possible bus values (0, 1, and -1, which are low, high, and floating respectively). Each branch of the DECODE statement allows the hardware to react differently depending on any previous information on the bus. Although this is not the most straightforward hardware description, it does adequately describe the semantics of tristate buses so that UMS can exploit its potential.

```
bus := -1;                                          ! line 1
N = 0 next                                          ! line 2
mainloop :=                                         ! line 3
begin                                               ! line 4
   IF timefunc(pseudofield[μPC]<1>,N,bus) =>        ! line 5
   begin                                            ! line 6
      DECODE bus =>                                 ! line 7
      begin                                         ! line 8
          0 := ~~~                                  ! line 9
          1 := ~~~                                  ! line 10
         -1 := ~~~                                  ! line 11
      end next N = N+1 next                         ! line 12
   end next                                         ! line 13
   IF timefunc(pseudofield[μPC]<2>,N,bus) =>        ! line 14
   begin                                            ! line 15
      DECODE bus =>                                 ! line 16
      begin                                         ! line 17
          0 := ~~~                                  ! line 18
          1 := ~~~                                  ! line 19
         -1 := ~~~                                  ! line 20
      end next N = N+1 next                         ! line 21
   end next                                         ! line 22
   IF (notend(N,bus)) then RESTART mainloop         ! line 23
   end                                              ! line 24
```

Figure 4.1: Creating the proper data dependencies for tristate hardware busses. The symbols "~~~" mean an individual sequence of code for that case is not shown.

4.2.4.2 Pipelines. Many microengines use a pipelined design style [PRO84], [SIEBN82], [KOG81], [RAML77] to achieve greater performance. This section describes what pipelines are and their specification in the hardware description.

A pipeline contains multiple stages of processing activity in which the results of the processing at one stage are passed sequentially to subsequent stages. Registers between each stage alternatively store the results of the previous stage and then pass these results to the subsequent stage; these two activities occur during alternate time cycles. An example of a human pipeline is a chain of people sequentially passing buckets of water to each other so that the one at the end may put out a fire (ignore the problem of empty buckets at the end). The simplest pipeline consists of one processing stage and a register at its input or output. Here, after calculating the next microinstruction address, a register holds its value until the next microinstruction execution period. Such a pipeline can be observed in most microprogrammed microengines [SIEBN82], [AMD81].

Pipelined microengines present no difficulties, other than increased complexity, to either UMS or the writing of the hardware description. The key to writing a pipelined hardware description is to specify registers that store the intermediate values of the pipeline and pass these values to the separate processing stages.

Figure 4.2 shows the outline of a two-stage pipeline hardware description. One step in the pipelined processing cycle contains three phases. First, the input values are loaded from the pipeline input (line 4) or the register from the preceding stage (5). Then the inputs at each stage is processed (8, 10). Finally, the processing outputs are stored in the register prior to the subsequent stage (13) or as the pipeline output (14). The first time the "onepipecycle" code executes, the input value "PipeInput" (line 1) is processed (lines 4 and 8) and stored for later use (line 13) in "Register1". The second time "onepipecycle" executes, the stored value from "Register1" is processed (lines 5 and 10) to become the output value of the code (lines 14 and 1). Thus in Figure 4.2 it requires two executions of the "onepipecycle" code to completely process a piece of input data.

```
onepipecycle(PipeInput, PipeOutput) :=          ! line 1
begin                                           ! line 2
      ! read Pipeline Registers                   line 3
   Input1 = PipeInput;                          ! line 4
   Input2 = Register1                           ! line 5
   next                                         ! line 6
      ! Stage 1 Processing                       line 7
   Output1 = f1(Input1);                        ! line 8
      ! Stage 2 Processing                       line 9
   Output2 = f2(Input2)                         ! line 10
   next                                         ! line 11
      ! Write New Values in Pipeline Registers   line 12
   Register1 = Output1;                         ! line 13
   PipeOutput = Output2                         ! line 14
end;                                            ! line 15
```

Figure 4.2: Example of a two-stage hardware pipeline described in ISPS. Not shown are the data declarations or semantics of the pipeline steps f1 or f2 (see text).

4.2.4.3 Variable time-delay activities. The following text describes how hardware-related timing constraints may be described in the hardware description. The two subsumptive issues are variable time-duration activities and asynchronous events. In the inference engine hardware with timing constraints is viewed merely as a more-complicated form of data dependencies, which UMS can already process.

When different parts of the engine can operate semiautonomously, varying intervals of time (or windows) are needed for information transfer. In some minicomputers, for example, data is available three, four, or five microcycles after the microengine issues a memory-read request to a memory controller. The memory data can therefore be used as a variable time delay after executing the memory-read-request microoperation. Instead of assuming that the memory-read activity always takes five microcycles, a conditional branch in the microcode can test if the memory data is ready. If the memory data is ready after only three microcycles, then such a microprogram can run up to two microcycles faster. Because microcode speed is so important, a great deal of effort is typically put into these small optimizations.

Variable time-delay hardware events may be modeled on a modified pipeline structure in which no actual processing is performed at each step in the pipeline. In the memory controller example described in Figure 4.3, each time the memory-access code executes, a pipeline of assignment statements

```
memoryportread(readdata) :=                    ! line 1
begin                                          ! line 2
   begin                                       ! line 3
   time5data = time4data next                  ! line 4
   time4data = time3data next                  ! line 5
   time3data = time2data next                  ! line 6
   time2data = time1data next                  ! line 7
   time1data = memoryreadport next             ! line 8
   DECODE pseudofield[μPC]<0:2> =>             ! line 9
      begin                                    ! line 10
      #3\ 3_cycles_later :=                    ! line 11
          readdata = time3data                 ! line 12
      #4\ 4_cycles_later :=                    ! line 13
          readdata = time4data                 ! line 14
      #5\ 5_cycles_later :=                    ! line 15
          readdata = time5data                 ! line 16
      end                                      ! line 17
   end                                         ! line 18
end                                            ! line 19
```

Figure 4.3: Description of a variable time-delay memory-controller read.

(lines 4 through 8 in Figure 4.3) records the value of the memory-read port for up to five time units in the past. A programming trick called a pseudomicroword field (discussed earlier and also used in Figure 4.3) creates a logical extension to the microword's width by associating an unspecifiable variable with each microword. This new variable is never part of the microcode output but is used to help control this type of timing problem. A field in the logical microword extension (pseudomicroword field) determines which of the subsequent microcycles (in this example cycles 3 through 5) may access the memory-read value. This determination depends on the value of the pseudomicroword field; this value cannot be specified through the control dependency of the DECODE statement index. Therefore, all three permutations must be considered

possible. By applying the pipeline data dependencies across multiple microcycles, variable delay and duration events such as this one may be described for the microcode synthesizer.

4.2.4.4 Asynchronous events. When hardware reacts to external stimuli by causing a sudden transition to a known state, it is called an asynchronous hardware event. A common example is an interrupt from an input or output device. This section discusses how to describe such behavior in the hardware description.

Figure 4.4 shows the general form for code describing asynchronous events. When an event is not present, the hardware processing begins in line 8 and ends in M-1. These lines describe the processing that can occur when it is impossible to interrupt the hardware. In a microengine, this code structure typically would be the body of the fetch-execute loop. When an asynchronous event of the highest priority occurs, lines M through M+5 execute. Events with lesser priority are described in priority order in the code between lines M and N-1.

The behavior of the code of Figure 4.4 is described as follows. The variable describing which asynchronous event has occurred most recently is initialized in line 1. The main loop of the code consists of the endless REPEAT loop described in lines 2, 3, 4, and N+2. The contents of the loop (lines 7 to N) are isolated (lines 5 and N+1) within the code so that they may have a labeled

name (line 6). Any normal processing is located in lines 8 to M-1. Until an event that satisfies code in the form of line M occurs, the code of lines 8 to M-1 is REPEATedly run.

```
event = 0 next                                  ! line 1
mainloop :=                                     ! line 2
begin                                           ! line 3
   REPEAT                                       ! line 4
   begin                                        ! line 5
      interruptedloop :=                        ! line 6
      begin                                     ! line 7

         ... hardware processing without event ...

         IF event1() =>                         ! line M
         begin                                  ! line M+1
            event = 1 ;                         ! line M+2
            processevent1() next                ! line M+3
            RESTART interruptedloop             ! line M+4
         end                                    ! line M+5

         ... more event handlers ...

      end                                       ! line N
   end                                          ! line N+1
end                                             ! line N+2
```

Figure 4.4: Modeling asynchronous hardware events.

Lines M through M+5 describe an event handler for asynchronous events. When more than one event is possible, these lines would be replicated and the character "1" appropriately changed in lines M, M+2, and M+3. These event handlers use a function of the form shown in line M to determine whether the event has occurred. This function may have additional input parameters. The code (shown in line M+3) describing the change of state due to the asynchronous event (recognition of the event) may also contain input and

output parameters. Once the change of state has been recognized (as described in code in the form of line M+3), the code is RESTARTed by the implied GOTO of line M+4. The data dependencies generated by this hardware-description technique allow UMS to understand the implications of the occurrence and absence of an external asynchronous event.

Four commonly used hardware-design techniques and ways to represent them in a hardware description have been discussed. The next section discusses how these hardware descriptions are analyzed to reflect their original meanings.

4.3 Computer-Description Compilation Techniques

Control and data dependencies express the meaning of the hardware and instruction-set descriptions. These dependencies are used by the UMS inference engine during synthesis and are expressed as a data structure called a graph. Graphs are used commonly as an internal form in compilers ([ANKCHM82], [AHOU77], [WULFWGH75], [AHOU72], [GRI71]) to represent the control and data dependencies of program code. Since these dependencies are only implicit in description source code, the compiler built into UMS creates a graph to make them explicit from the ISPS ([BARBCS78], [BAR79], [PARTCC79], [BARS82]) source code. This compiler is described below. The

output of the compiler is annotated with additional information by the postprocessor described later.

4.3.1 Parsing and metamorphosis grammars

The ISPS compiler implemented for UMS extensively uses metamorphosis-grammar technology [COL78], [WAR80] (which is very closely related to definite-clause grammars [PERW80]). The postprocessor described later in this chapter does not use metamorphosis-grammar technology. Metamorphosis grammars are an extension to the Prolog language [CLOM81]. This type of grammar, which is a syntactic variant of Backus-Naur Form (BNF), allows automatic translations of parser descriptions into a Prolog implementation of the parser. Ease of parser implementation was one of the Prolog designer's goals [COL78].

The first phase of the UMS compiler reads source-code characters from the disk and performs lexical analysis to partition the character stream into tokens. The second phase arranges these tokens into a parse tree. This parsing is based on the grammar described by Barbacci [BARBCS78] but is greatly modified to correct errors and improve efficiency. The third phase converts the parse tree to graph form, which is stored as Prolog source text in a temporary disk file. Each of these phases is described as a grammar and implemented with metamorphosis-grammar techniques. These phases are of conventional Prolog-based design and are not discussed further. The fourth

phase, called the postprocessor, reads the temporary disk file, annotates each node with additional control-and-data-dependency information, and creates a final disk file.

4.3.2 Compiler-output data structures

There is a direct relationship between the ISPS source code, which may describe either an instruction set or the hardware that implements it, and the arcs and nodes of the control-and-data-dependency graph (see Figure 4.5). A node represents an operation within a fragment of source code, and the arcs represent control or data dependencies.

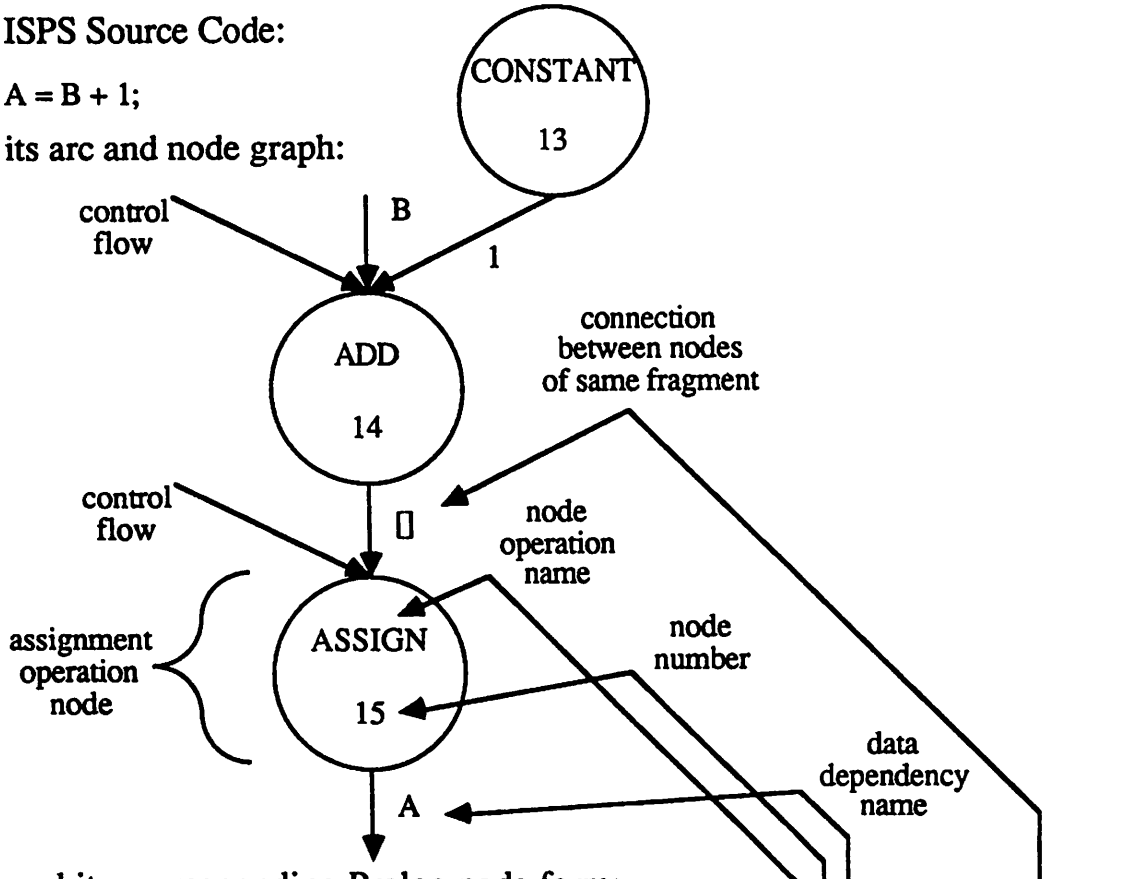
One or many nodes may represent a single microoperation. Figure 4.5 shows that the ISPS compiler translates the statement "A = B+1;" into three nodes, which are shown diagrammatically in the middle of the figure. The nodes, in order of their use, are the "CONSTANT" node (in which a constant with a value of one is created), the addition node ("ADD" with the operands "B" and "1"), and the assignment node ("ASSIGN" which writes to the variable "A"). Although this translation style may generate more nodes than may appear to be necessary, later discussion shows why the semantics are decomposed to this degree of resolution.

The pretty printed Prolog nodes representing the statement "A = B+1;" are also shown in the bottom of Figure 4.5. A comment in Prolog begins with "/*" and ends with "*/". Each node is a 4-tuple, with a head (or datatype name)

ISPS Source Code:

A = B + 1;

its arc and node graph:



and its corresponding Prolog node form:

```
hwnode('CONSTANT'                                          /* line 1  */
       [],                                                 /* line 2  */
       [[1,[],[14]]],                                      /* line 3  */
       13).                                                /* line 4  */
hwnode('ADD',                                              /* line 5  */
       [                                                   /* line 6  */
         [_,[['CHOICE',_,'SUBRDEF']],[12]],                /* line 7  */
         ['B',[],[6,11]],                                  /* line 8  */
         [1,[],[13]]]                                      /* line 9  */
       ],                                                  /* line 10 */
       [[[],[],[15]]],                                     /* line 11 */
       14).                                                /* line 12 */
hwnode('ASSIGN',                                           /* line 13 */
       [                                                   /* line 14 */
         [_,[['CHOICE',_,'SUBRDEF']],[12]],                /* line 15 */
         [[],[['FunctionName',_,'ADD']],[14]]              /* line 16 */
       ],                                                  /* line 17 */
       [['A',[['WIDTH',_,[0,3]]],[23]]]                    /* line 18 */
       15)                                                 /* line 19 */
```

Figure 4.5: The relationship between a fragment (single statement) of ISPS source code (top), its arc and node graph (middle), and its Prolog node form (bottom, see text).

termed "hwnode" for hardware (shown here) or alternatively "instnode" for the instruction-set description.

The first parameter of the 4-tuple is a description of the node operation, which is "CONSTANT" for the first node. The second and third parameters are lists of inputs and outputs, respectively. The fourth parameter is a unique node number, which is 13 for the first node of Figure 4.5. As is shown at the end of line 4, a tuple or clause in Prolog always ends with a period. An input or output list can be empty (described by the nil character "[]") or composed of a list of as many entries as there are variables.

The first node has no inputs (indicated by the first "[]") and one output (shown as "[[1,[],[14]]]"). Each input or output entry has three parts: first a variable name, then a list of attributes, and last a list of arcs to other nodes (these are "1", "[]", and "[14]" respectively in node 13). If a variable name is nil (shown as "[]"in lines 11 or 16), then the arc connection is part of the same individual operation. If a variable name is itself a Prolog variable (described by "_" in Prolog, and shown in lines 7 and 15) instead of a Prolog constant (such as 'B' in line 8), then a special dependency representing controlflow is described. If a variable is an integer (shown in lines 3 or 9), then a constant is used. A variable name may employ an ISPS source-code variable (lines 8 and 18) to describe a specific named data dependency. Each attribute is a three-element list composed of an attribute name, the anonymous Prolog

variable "_", and an attribute value. These Prolog data structures directly represent the original description's source code.

Now that the structure of the compiler and its output has been described, techniques for the controlflow and dataflow analysis will be discussed next.

4.3.3 Controlflow and dataflow postprocessor

The compiler produces graph arcs and nodes which represent only local control-and-data dependencies. Any required global dependency information could be searched for during synthesis. Since multiple, repetitious searches at synthesis time is inefficient, a postprocessor phase analyses the graph and adds any additional global dependency information. The analysis involves a single exhaustive search through a code graph at the time of compilation and postprocessing.

The inference engine modifies the instruction-set description section by section, without regard to instruction-set context, to make it compatible with hardware design details. However, hardware context often contains a considerable amount of detail. The most important context information describes how each part of the hardware connect along buses to other parts of the hardware. Because the amount of context information used by UMS is much greater for the hardware description, some of the preprocessor's global analysis, such as bus analysis, is not required for the instruction-set description. This section describes types of search required for this analysis as well as

information about the semantic interpretation of the hardware and instruction-set descriptions.

The first type of analysis is complete control-and-data-dependency trace analysis, to be described in detail later. All other types of analysis are based on this information.

The second type of postprocessor analysis determines if each variable is read, written, or both, anywhere in the code. Because the synthesis system is not given prior information about the intended use of hardware components, this analysis allows the detection of input ports, output ports, or state storage in the hardware description. The analysis increases the likelihood of a correct mapping between a hardware component and a variable of the instruction-set description, so that search is more efficient.

The third type of analysis merely records which variables are used with indices. For example, this aids the search for hardware memory arrays to map with the primary memory array of the instruction-set processor.

The last type of analysis searches for hardware buses (this analysis is switched off for the instruction-set description). A bus may be a special cable, with plugs and sockets, or merely a set of connections on a printed circuit board. In a hardware description, a bus may be represented by a series of simple assignment statements (with a single variable as the value source) and parameter passing through subroutine calls. A bus represents the paths that a

variable value can move through unmodified. Instead of embedding bus information in the hardware graph, a separate data structure is created to contain this information efficiently and succinctly. Now that the four functions of the postprocessor have been described, control-and-data dependency analysis will be described in detail.

4.3.3.1 Controlflow and data-dependency tracing. To determine global data dependencies, a postprocessing phase of the compiler walks through, in controlflow order, the code statements of the hardware and instruction-set description. The exact execution order may be dependent on input data, and is discussed in more detail shortly. A symbol table of data dependencies is updated at each step in the walk-through. This symbol table is used to annotate each node with data-dependency information. The postprocessor also analyzes variable name "alias" information because in ISPS a single bit of storage can be described with many names, and a single name can refer to many locally scoped variables in different parts of the code. When subroutines are called to or returned from, a form of translation occurs for parameter names, so the correct data dependencies are determined across the call boundary. This code walk through is roughly the order in which they would execute in a simulation.

The instruction-set description and hardware description are in procedural form; this means they can act as simulators. Each of these

descriptions contain many subroutines. When either of these two simulations are run, a function call must be entered into an ISPS interpreter to begin execution of the top-level code. UMS requires this same information so it may analyze the global controlflow and dataflow from the same point of view. This information is currently supplied manually to the UMS compiler. This is the only additional information the compiler and postprocessor receive about the code they process.

The ISPS language allows many types of conditional branches, and the postprocessor determines the resulting data dependencies from all possible branch choice permutations. The postprocessor starts at the beginning controlflow point in each description and traces every controlflow path through the nodes.

While the tracing progresses through a description, a symbol table for each bit is maintained by inspecting all variable reads and writes. This is necessary because ISPS may allow modification of fewer than all bits of a variable. The conditional branch choices that lead to the execution of a statement describe position in the code controlflow. The symbol table is organized hierarchically for each variable based on controlflow. When controlflow may alternatively flow down two or more paths (this is called a fork), separate symbol tables are maintained for each path branch. When controlflow paths join, the symbol tables are also joined. When subroutines are called,

controlflow is traced through them. The necessary name translations are made for the subroutine parameter list to maintain the symbol tables accurately. These symbol tables describe exactly the program's data dependencies.

Controlflow nodes partition the source code into a hierarchy of blocks (or regions) of code. Each block boundary is the source or destination of a conditional branch (see [HEC77] or [FIS79] for further discussion about this type of analysis). To implement the type of controlflow tracing just described, a stack is kept of as-yet-unanalyzed nodes. A controlflow node (such as a node describing an IF statement, shown in lines 1 to 7 of Figure 4.6) may require the evaluation of a set of other control nodes (such as a block of code for the "then" and "else" parts). These control nodes may in turn specify the evaluation of other control nodes (possibly an embedded DECODE statement within the "then" part). The block nesting depth at a given point in the evaluation is called the controlflow level. These partitions describe the alternative ways the code may be used.

Controlflow processing begins by putting the node number of the simulation's subroutine call (described earlier) on a stack of controlflow nodes. This first node is initially the only element of the stack. The controlflow-processing cycle begins by popping the next node for analysis from the top of the stack. If the current node is a controlflow fork, then a special symbol and all embedded nodes (such as the "then" and "else" nodes of an IF

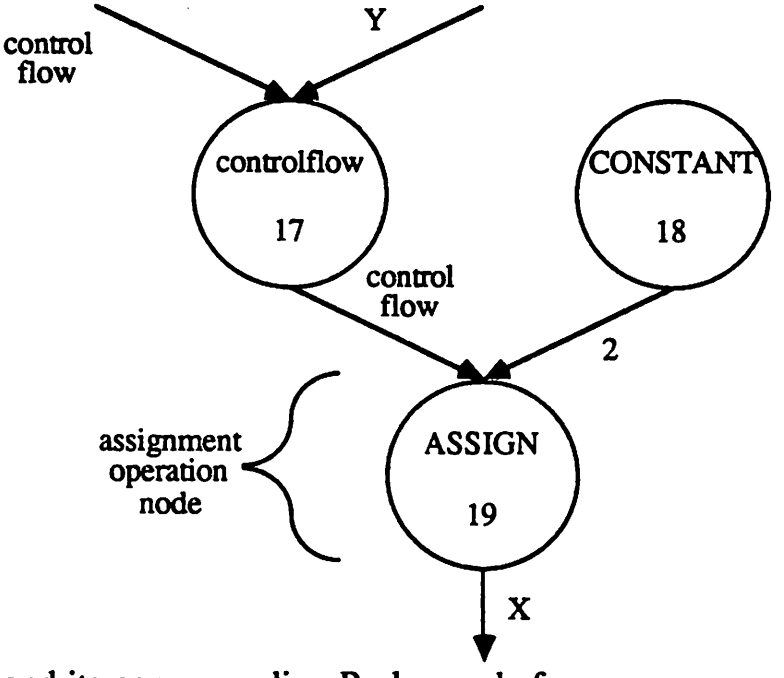
ISPS Source Code:

IF Y => X = 2;

- *or alternatively* -

DECODE Y<0> =>
begin
1 := X = 2
otherwise := no.op()
end;

its arc and node graph:



and its corresponding Prolog node form:

```
hwnode('controlflow'                                    /* line 1 */
       [                                                /* line 2 */
        [_,[['CHOICE',_,'SUBRDEF']],[14]],              /* line 3 */
        ['Y',[],[8]]                                    /* line 4 */
       ],                                               /* line 5 */
       [[_,[['CHOICE',_,'TRUE']],[19]]],                /* line 6 */
       17).                                             /* line 7 */
hwnode('CONSTANT'                                       /* line 8 */
       [],                                              /* line 9 */
       [[2,[],[19]]],                                   /* line 10 */
       18).                                             /* line 11 */
hwnode('ASSIGN',                                        /* line 12 */
       [                                                /* line 13 */
        [_,[['CHOICE',_,'TRUE']],[17]],                 /* line 14 */
        [2,[],[18]]                                     /* line 15 */
       ],                                               /* line 16 */
       [['X',[['WIDTH',_,[0,3]]],[23]]]                 /* line 17 */
       19).                                             /* line 18 */
```

Figure 4.6: The relationship between a conditional branch in ISPS source code (top), its arc and node graph (middle), and its Prolog node form (bottom, see text). Note that for the second source code alternative, the choice described within the Prolog code would be '1' instead of 'TRUE.'

statement) at the next deeper level are pushed onto the top of the unprocessed-node stack. Later, when this symbol is popped and given to the postprocessor to evaluate, it triggers the necessary data dependency merge from the symbol tables for all lower controlflow levels. This analysis of controlflow and dataflow semantics detects exactly what the code means. Now that the techniques used to order and control processing of each code fragment in a description has been discussed, the actual processing of each fragment type will be described in detail.

4.3.3.2 DECODE and IF semantics. DECODE and IF statements internal to the postprocessor have identical forms and are labeled as nodes of type "controlflow" (see the node described by lines 1 to 7 of Figure 4.6). These nodes receive a single input value (see line 4), which may come from a simple variable or from a value calculated from an arbitrarily complex expression. The outputs are of a special type of data dependency called "controlflow", which is named in the Prolog node form as the anonymous variable "_" (see line 6).

The Prolog node notation in UMS does not allow placing inputs or outputs with the same ISPS source-code names in a node. However, a node may use the anonymous "_" Prolog variable name with one or more controlflow outputs. Each controlflow output for the same value of the input variable describes those nodes that receive controlflow from the statement. In the case of an IF statement, there is only one output-node entry, which represents the "then" part

of the IF statement. The "controlflow" input variable value is described in the arc's "CHOICE" attribute field (shown as "TRUE" in line 6 of Figure 4.6 for the IF statement). For DECODE statements, each of these entries represents a different value of the DECODEd variable.

Two types of data-dependency semantics can be associated with a DECODE statement: the contents of the statement can conditionally replace (called merge) the previous values or they can completely replace (called join) any previous values (see Figure 4.7). Since ISPS does not have an IF-THEN-ELSE construct, ISPS IF statements always perform a merge (conditional replacement).

To understand what conditional replacement means, consider the ISPS fragments of Figure 4.7. After line 3 of the first fragment, the value of X could be either 1 or 2, depending on the value of Y. When the postprocessor analyzes such a fragment, it cannot determine the value of Y because the value may depend on input data. When line 3 of this fragment is processed, the updated symbol table indicates that the value of X may come from either line 1 (prior to the conditional branch) or line 2 (from the conditional branch). This is called conditional replacement because the controlflow statement might (conditionally) write a new value of the variable instead of always writing a new value (although in some DECODE statements it may write one from a choice of many values). Conditional replacement within a controlflow statement of a

Fragment #1, Data Dependency Merge:

```
X = 1 next          ! line 1
IF Y => X = 2       ! line 2
next                ! line 3
```

Fragment #2, Data Dependency Join:

```
X = 1 next          ! line 1
DECODE Y<3>         ! line 2
   begin            ! line 3
   0 := X = 2,      ! line 4
   1 := X = 3       ! line 5
   end              ! line 6
next                ! line 7
```

Fragment #3, Data Dependency Join:

```
X = 1 next                    ! line 1
DECODE Y<0:11>                ! line 2
   begin                      ! line 3
   0 := X = 2,                ! line 4
   otherwise := X = 3         ! line 5
   end                        ! line 6
next                          ! line 7
```

Fragment #4, Data Dependency Merge:

```
X = 1 next          ! line 1
DECODE Y<0:11>      ! line 2
   begin            ! line 3
   0 := X = 2,      ! line 4
   1 := X = 3       ! line 5
   end              ! line 6
next                ! line 7
```

Fragment #5, Data Dependency Merge:

```
X = 1 next          ! line 1
DECODE Y<3>         ! line 2
   begin            ! line 3
   0 := X = 2,      ! line 4
   1 := Z = 3       ! line 5
   end              ! line 6
next                ! line 7
```

Figure 4.7: Examples of data-dependency merges and joins. In fragments 1, 4, and 5, the IF or DECODE statements can possibly modify the value of X or leave it unchanged. This is called a data-dependency merge. In fragments 2 or 3, any path through the IF or DECODE statements changes the value of X. This is called a data-dependency join.

variable value requires the symbol table to carry along data-dependency information from two locations: 1) the value originating from a location prior to a controlflow statement, and 2) the value written inside the control flow statement. This differs from the usual action of completely replacing the data-dependency record with the new assignment-statement location whenever the variable is written.

In some cases, complete replacement (also called a join) can occur, as shown in fragments 2 and 3 of Figure 4.7. Fragments 2 and 3 show the value X to be reassigned in all possible controlflow paths; this is called complete replacement. The use of all bit permutations of the DECODEd variable is shown in fragment 2, while the use of the OTHERWISE alternative is shown in fragment 3. The postprocessor calculates the number of possible values the "condition choice" part of the DECODE statement may have by determining the number of variable bits used in the condition variable. Inspecting the optional bitfield of the variable (found in line 2 of fragment 2, but not in line 2 of fragment 1) determines this number. The number of possible values also may be determined by inspecting the variable definition found elsewhere in the code.

Fragment 4 of Figure 4.7 shows another example of conditional replacement. In this case, the postprocessor has determined that complete replacement (join) of the X value cannot occur because the 12 bits of Y used to determine the condition represent more permutations than the two choices

found within the DECODE statement. The postprocessor also determines that fragment 5 of Figure 4.7 also shows conditional replacement because not all possible DECODE choices modify the variable X. These alternatives describe the ways that controlflow and dataflow may interact in a description. All of these alternatives are usually expressed with simple conditional branches in the synthesized microcode. Now that analysis of the most common controlflow-operation type has been discussed, the more specialized forms will be described next.

4.3.3.3 RESTART semantics. The ISPS RESTART statement is a restricted form of a GOTO statement, which terminates the current controlflow path so that processing may resume at the code described by a label. The label associated with a RESTART is assumed to be bound to code previously encountered in the execution path. Otherwise, RESTART would not have a well-defined semantics (and would not be performing the action implied by its name). A RESTART forms the main fetch-execute loop of the PDP-8 hardware description example of Chapter VII. This code also includes a RESTART statement that forces immediate processing of input or output in the PDP-8 instruction-set description.

Figure 4.8 shows a sample RESTART statement. After running the condition statement of line 5, the execution can continue with line 6 or jump back to line 2. Therefore, line 2 can execute from two contexts: immediately

after the execution of line 1 or after the execution of line 5. Note that lines 2 through 5 can form a loop. The data dependencies also reflect these two controlflow possibilities; the value of X read in line 4 could have been written in line 1 (from the first time through) or in line 4 (after looping through a RESTART).

```
X = 0 next                                  ! line 1
labelname :=                                ! line 2
begin                                       ! line 3
   X = X + 1 next                           ! line 4
   IF Y => RESTART labelname next           ! line 5
   X = X + 2                                ! line 6
end                                         ! line 7
next                                        ! line 8
```

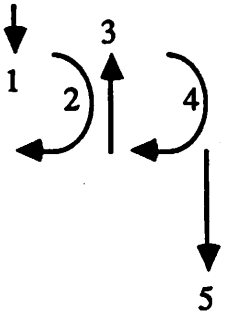


Figure 4.8: Example of controlflow through a RESTART statement. The numbered arrows on the right show the way the postprocessor traces the controlflow through the fragment.. The value of X read in line 4 could come from lines 1 or 4.

During the postprocessor's controlflow tracing (the trace is shown on the right of Figure 4.8 as arrows 1 and 2), the RESTART statement branch is performed the first time it is encountered (arrow 3), and the tracing continues from the labeled code branch destination (arrow 4). The first time each RESTART is encountered, UMS records what code caused it to be performed. The record also contains the controlflow-choice history leading up to the RESTART. The second time UMS encounters the RESTART statement with the same controlflow choices (tip of arrow 4), data dependencies are merged, and processing continues (arrow 5) from the statement after the RESTART. Treating the RESTART statement differently the first and second times it is encountered

avoids an infinite loop within the postprocessor. Successive encounters with the same RESTART statement alternate between these two processing techniques.

This approach considers both sources of data dependencies for X in line 4; X can receive a value from line 1 (arrows 1 and part of 2) or line 4 (arrows 1, 2, 3, and part of 4). Note that although more than one RESTART statement may point to the same label, these statements are processed sequentially. These data dependencies describe all possible RESTART invocation permutations. The RESTART statement, and its relative the REPEAT described next, represent the most difficult controlflow-choice history processing (and its corresponding data-dependency processing) that UMS performs.

4.3.3.4 REPEAT semantics. The data-dependency analysis of REPEAT and RESTART statements is the same. Whereas there is an implicit infinite loop between the label and the RESTART statement (described earlier), the REPEAT explicitly describes an infinite loop. An example of this is found in the PDP-8 instruction-set synthesis of Chapter VII.

4.3.3.5 Subroutine CALL & END semantics. When the postprocessor encounters subroutine CALLs, it creates a special type of symbol-table entry called a "submap" (for subroutine map) and allocates a hierarchical controlflow level just for the subroutine call. This entry lists each variable's name in the subroutine call along with the variable name that it is

matched to in the body of the subroutine. In the routines that access the symbol tables, these special symbol-table entries are searched before any others. If a match is found, the search continues, but instead of using the original variable name and controlflow level, the name associated within the "submap" is used at a prior controlflow level. Using this technique, the data-dependency analysis of subroutine calls is equivalent to that which would have occurred if the body of the subroutine had been expanded in-line. Further, this dataflow analysis across subroutine boundaries regularizes synthesis regardless of how the source code was modularized.

4.3.3.6 Alias analysis. This section describes how the postprocessor deals with the potential confusion of multiple names for variables and bitfields within a variable typically found within an ISPS program's data-declaration statements. For example, a microstore memory called "uMem" may have variable names associated with each of its fields and subfields. This "part of" relationship may be chained to any depth. Although this ISPS alias feature makes analysis more difficult [PARTCC79], it greatly improves the legibility of the source code.

The postprocessor determines the largest (by width in bits) variable of a set of variables that shares some of the same bits; this is the parent variable. When more than one variable within a set is of the maximum width, the first one of these encountered is labeled the parent. All other variable names from the

set are considered aliases. The case in which the two largest bitfields do not contain the same bits was not observed in the inspected ISPS source code and is not handled by the postprocessor. It would require little additional effort to add this feature to UMS.

Each entry in the symbol table has pointers to its parent variable and describes which bits within the parent variable the entry name describes. A parent variable's pointer fields are blank. Each ISPS source-code variable name has its own symbol-table entry. When the postprocessor uses data-dependency information only the data dependencies of the parent variable are read or modified. This is because processing proceeds on the indicated symbol-table entry after a variable name is matched in a symbol table when the parent pointer field is not blank. This strategy assures that the proper data dependencies are recorded.

4.3.3.7 Hierarchical bitwise symbol table. The postprocessor creates the symbol table to define where each variable value is written in the source code. The postprocessor then annotates the arcs of the control-and-data-dependency graph with this information to speed up later synthesis search. This section describes the design and use of the symbol table.

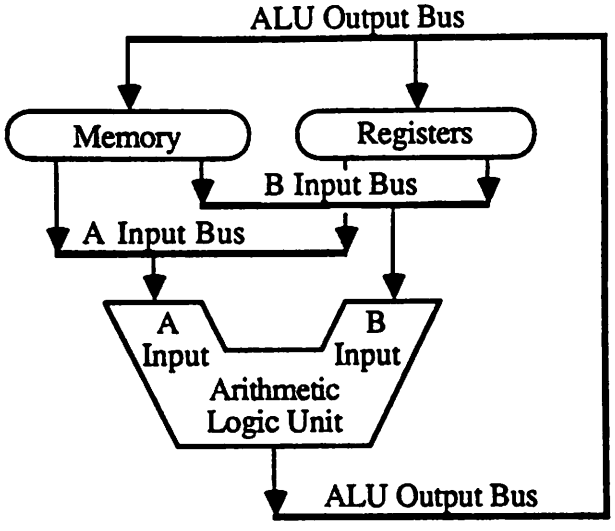
Each symbol-table entry contains a field that describes the parent variable name (described earlier in the section on aliases). When a symbol-table entry

is to be accessed and the symbol is not a parent, then the parent's entry is manipulated instead. Previous discussion described how multiple nodes may write a variable in more than one controlflow path. Since the multiple variable names may refer to different bitfields of the parent variable, each bit of the parent variable contains its own data dependencies. Specifically, each symbol-table entry of a parent variable contains a list of nodes that could have written each bit.

Different subroutines may use the same name locally for logically different variables. The symbol table is organized hierarchically to provide access to the correct data dependencies. Each symbol-table entry contains both a controlflow-level number and a controlflow-path description. These two additional symbol-table fields disambiguate the occurrences of the same symbol name. Each section of the controlflow path can be thought of as locally maintaining its own symbol table since the entire symbol table is partitioned by the value of its controlflow path field. When a symbol is not found in a current controlflow partition, then consecutively higher partitions are searched for a match. When the symbol is found, the symbol-table entry is copied into all intermediate partitions, with the appropriate modifications of the controlflow path field. After the data dependencies of different controlflow paths are merged or joined (described earlier), the unneeded symbol-table entries for these levels are deleted.

A subroutine CALL statement causes definition of its own controlflow level. Symbol-table entries are made for each variable used in the call. Special fields in these symbol-table entries specify which bits of the variable in the subroutine call match which bits in the subroutine-definition header. The data dependencies of the subroutine CALL control level are copied into the symbol-table entry of the subroutine-header level. After the postprocessor finishes analyzing the subroutine body, the data dependencies are merged or joined with those of the call level. Therefore, the postprocessor considers variable names within a subroutine to be entirely local. This data-dependency analysis has a similar effect to expanding all variable names from the subroutine CALL into the body of the subroutine. These two techniques for analyzing ISPS variable references, alias disambiguation and hierarchical symbol tables, solve a common criticism with the language.

4.3.3.8 State-variable analysis. State variables of the hardware description can communicate values between execution cycles of the microengine. The hardware description does not specify explicitly which source-code variables can store state information (these variables are called memory or registers, see Figure 4.9). Similarly, the hardware description does not make explicit which variables merely temporarily store values (and do not correspond to a specific part of the hardware) but are used simply to make the hardware description easier to write. Analysis of state variables is necessary




```
DECODE uwrd[uPC]<0>                              ! line 1
   begin                                         ! line 2
   0 := Abus = Memory[MAR],                      ! line 3
   1 := Abus = Reg[uwrd[uPC]<1:3>]               ! line 4
   end next                                      ! line 5
DECODE uwrd[uPC]<4>                              ! line 6
   begin                                         ! line 7
   0 := Bbus = Memory[MAR],                      ! line 8
   1 := Bbus = Reg[uwrd[uPC]<5:7>]               ! line 9
   end next                                      ! line 10
DECODE uwrd[uPC]<8>                              ! line 11
   begin                                         ! line 12
   0 := ALUbus = Abus + Bbus,                    ! line 13
   1 := ALUbus = Abus - Bbus,                    ! line 14
   end next                                      ! line 15
DECODE uwrd[uPC]<9>                              ! line 16
   begin                                         ! line 17
   0 := Memory[MAR] = ALUbus,                    ! line 18
   1 := Reg[uwrd[uPC]<10:11>] = ALUbus           ! line 19
   end next                                      ! line 20
```

Figure 4.9: General structure of a simple microengine. Controlflow and data-dependency analysis can determine (see text) that the variable Memory is a state variable because it can be read at the beginning of the code sequence (line 3) and written at the end of the sequence (line 13). Conversely, Abus is first written (lines 3 and 4) and subsequently read (lines 13 or 14), suggesting that its use is simply of a temporary variable or bus. Figure 5.2 shows the complete controlflow for this microengine.

for UMS to store and retrieve values in the hardware correctly. To ensure that every potentially available part of hardware state is used in the microcode, UMS deduces these variable-use distinctions as described in this section.

A very simple analysis technique can detect registers in a hardware description. The analysis assumes that the hardware and its description are designed intelligently (compare to [PAHA81]) so that undefined data values are not read or written. Specifically, the key assumption is that hardware reads data (at the beginning of a microcycle) only from parts of the hardware with information stored in previous microcycles. There is no sensible reason to read from a piece of hardware with undefined data values. Similarly, it is assumed that the hardware performs writes at the end of a microcycle only for a practical purpose: to store values that can be read during subsequent microcycles.

These assumptions are justified when the hardware description specifies the semantics of what the microengine can do, but not necessarily its exact design. To write a hardware description with other than the above constraints is considered unreasonable. Although it cannot be proved that the analysis techniques are sufficient in all cases, the hardware descriptions inspected to date have met these constraints.

State-variable analysis is illustrated with the simplified microengine diagramed in Figure 4.9. The code in Figure 4.9 shows the body of the fetch-execute loop. At the beginning of a microword-execution cycle, the ALU

(lines 13 and 14) may receive values from the registers or memory via the A Input Bus (lines 3 and 4) or the B Input Bus (lines 8 and 9). Similarly, the output of the ALU is written to the ALU Output Bus (lines 13 and 14) and then to either memory or the registers (lines 18 and 19).

UMS's control-and-data-dependency postprocessor does not assume that the same variable names in lines 3 and 4 can necessarily read the memory and register variable values written in lines 18 and 19 in a subsequent microcycle. This additional data dependency is not determined by the postprocessor, even when these lines of code are within a loop (not shown in this figure).

Unless variables with indices have exactly the same index (such as in the statement "X = Y[I] + Y[I]"), they are conservatively assumed to refer to different values. In the case of Figure 4.9, the value of MAR in line 18 is likely to change before line 3 executes again; certainly the value of uPC (the microprogram counter) in line 19 is likely to change before executing line 4. This assumption avoids using expensive analysis techniques such as symbolic execution (see [OAK79]) to determine conditions under which the values of the indices would be the same. In practice, this is a reasonable assumption.

Read/write analysis (described in the next section) determines that the memory and registers variables of Figure 4.9 are both read and written in the code. Since the data-dependency arcs determined by the postprocessor do

not show the values written in lines 18 and 19 being read by lines 3 and 4 (and these variables are assumed to be used rationally), they are therefore assumed to be state variables. Although cases that invalidate this type of simple analysis can be fabricated, they have not yet been found by inspection of ISPS source code. This observation has therefore been assumed to be true for this thesis. More advanced work may require special annotations to the hardware ISP to specify more subtle storage capabilities.

4.3.3.9 Read/write analysis. A significant number of the variables in a hardware description is used in special purpose ways: variables that only can be written are output ports, variables that are only read are input ports or constant ROMs (read-only memories). Variables of the instruction-set description perform similar purposes. However, most variables (or parts of the parent variable, described earlier) are both read and written. Variables therefore can be categorized as being read, written, or both (except those variables that erroneously are not used at all). This section describes how this information is used during synthesis.

When UMS attempts to map an instruction-set variable to a hardware variable a great deal of computational effort is made to verify the choice. Experience with an early version of UMS determined the usefulness of the read/write/both categorization. Comparing the characteristics between instruction-set and hardware variables can help avoid many synthesis-mapping

mistakes. When the postprocessor traces control and data dependencies, it posts whether the variable is read or written in the parent variable's symbol table (described earlier). This information necessary for synthesis is added to the variable-definition nodes of the graph just before the results of the postprocessor are written to disk.

4.3.3.10 Absence of side effects. The hardware description is intended to describe completely, with no hidden side effects, the semantics of the target microengine. This use of the term "side effect" is different than it would be in common reference to destructive assignment. A side effect occurs when a program action causes modifications other than those specified. In some Fortran implementations, a side effect occurs when a number is stored in an array when a negative-array index is used. In this case, storing a number into a named array could modify a variable with some other name. If the index value is beyond the bounds specified by the Fortran array description, the value sometimes can be stored within the user code area or even the operating system code, with correspondingly hard-to-predict consequences. Side effects such as these would invalidate the hardware and instruction-set description program analysis, which in turn would make synthesis impossible. UMS's computer-description approach requires that all effects be specified.

Condition codes, which are often considered a side effect, are characteristic of conventional hardware. When a computer's arithmetic logic

unit (ALU) performs an operation such as addition, its primary computational result is a written sum. In many central processing units (both at the microengine and instruction-set processor level) one or more one-bit condition codes are also stored in the state. Typically, these condition codes are the arithmetic-carry bit and bits that describe if the value of the previous ALU action was negative or zero. When the value of the sum is the only result desired by the programmer, the writing of condition codes is sometimes seen as an unwelcome side effect. In this case, the carry bit would have to be explicitly set to a known value before the arithmetic operation. When an ALU is specified in either a hardware or instruction-set description, it therefore contains at least two outputs: the arithmetic value and any condition codes. The postprocessor of UMS processes the data dependencies of this type of description in a straightforward manner. When potential side effects such as these are described and analyzed in this manner, during synthesis UMS does not have to consider the possibility of side effects.

4.3.3.11 Array handling. Two "side effects" that UMS does not consider in detail are reads and writes to memory arrays. For data-dependency analysis by the postprocessor, writes to a memory array are considered to alter potentially all cells of that variable. This means that two functionally independent array cells are considered to be the same variable in data-dependency analysis. This is similar to believing a whole integer has

been altered if only one bit of its value is modified (such as when an even number is incremented). The UMS variable-mapping technique distinguishes between memory array elements (such as registers) even though use of these arrays is based on this more conservative analysis. This greatly simplifies the analysis of memory arrays, but it comes at the cost of resolution of detail and, ultimately, at the cost of some potential code-generation optimizations. Inspection of typical source code has determined that in most cases, more-detailed analysis is unnecessary (such as discussed in [MUCJ81] and [HEC77]).

4.3.3.12 Bus analysis. Early experience with UMS determined that tracing the movement of variable values through the hardware description expended a great deal of computational effort. Typically, these searches answer questions such as, "Can this value be moved to one side of the arithmetic-logic unit?" Analysis of UMS behavior at the level of its Prolog source code and Prolog runtime environment determined that answering this type of question required repetitive and costly serial search and node-data-structure decomposition. This section first justifies the need for bus analysis, then defines a bus, describes how its use streamlines this type of search, and last discusses the design of the postprocessor's bus analysis.

There were two inefficiencies in tracing the flow of values through the hardware description: 1) the same trace was often repeated at synthesis time

and 2) much of the computational effort went into extracting connection information from the node data structure. To solve the first problem, this trace search was performed only once (while data dependencies were analyzed) during postprocessing of the hardware description and posted in an easy-to-access form in the database instead of being repeated during synthesis. The second problem was solved by creating a new data structure (called a bus) that is optimized for value-tracing search.

Conceptually, UMS calls a bus any part of the hardware that moves a value without modification. The UMS concept of a bus is more general than is commonly accepted in hardware design. A bus in UMS is specified in the hardware description as a chain of assignment statements and subroutine linkages. Conventionally, a bus represents a set of wires; from the UMS point of view, it also includes multiplexers and some other gates. In UMS, all the different buses are interconnected conceptually by the parts of the hardware that modify data values.

All nodes except for ASSIGN, VARDEF, SUBCALL, ARRAYSTORE, and STOREFIELD are ignored by the bus analyzer because these are the only node types that do not modify variable values. Typically, the UMS inference engine manipulates bus nodes even more often than it does those that perform operations (such as addition or subtraction). In the example of Chapter VII, the bus nodes account for 43% of either description type (see Figure 4.10).

Because these nodes make up a large percentage of a hardware or instruction-set description, bus analysis significantly speeds up later synthesis.

Instruction-Set Description	Hardware Description	Node Type
45	184	ASSIGN
44	130	VARDEF
20	24	SUBCALL
5	8	ARRAYSTORE
2	1	STOREFIELD
116	347	Total Bus Nodes
269	804	Total Nodes
43%	43%	Percent of Total

Figure 4.10: Proportion of semantically transparent nodes in the example instruction-set and hardware descriptions of Chapter VII.

The bus analyzer scans the nodes sequentially based on their node number. In this context, the ARRAYSTORE and STOREFIELD nodes are considered identical to ASSIGN nodes except that they are stored in memory arrays and variable subfields, respectively. When the bus analysis encounters an ASSIGNment node, it searches the existing bus-fragment database to determine if it is connected to this node. If the ASSIGNment node is part of an existing bus, this fragment is added to the database. If a bus fragment is part of two other buses, then they are all joined together to form a single bus. Otherwise, a unique bus number is generated and that node begins analyzing a new bus fragment in the database. Since the bus analyzer works from data dependencies based on the parent-variable analysis described above, it is not

fooled when a small bitfield value is assigned to a large variable, which in turn is assigned to another variable. Similar processing occurs for VARDEF nodes and subroutine CALLs.

In the ISPS language, a bit in a variable value can have many different alias names and be a part of a hierarchy of different variables (see [BARBCS78], [BAR79], [PARTCC79], and earlier discussion). Due to these characteristics of the source language, only parent variables are described in the bus data structure.

The Prolog form of the bus data structure is bus(N, INAME, INODE, ONAME, ONODE). Here N is a unique integer for each separate bus, INAME and ONAME are the parent (defined earlier) variable names for the input and outputs of a node, INODE is the node number from which the value was received, and ONODE is the node number of one of the above types. For example, bus(5,X,14,Y,22) means that part of bus number 5 receives the value called X from node 14 and is renamed Y by node 22.

The current implementation of bus analysis requires about 10% extra postprocessing time and programming effort, but sometimes improves synthesis speed by almost an order of magnitude. This experience suggests the importance of this often neglected aspect of synthesis systems.

4.3.3.13 Summary of machine description analysis. The previous discussion described the actions of the UMS postprocessor from a

functional point of view. The following summary describes the postprocessor's seven phases from an implementation point of view:

1) Read into the Prolog workspace the nodes that were stored on disk in Prolog source code by the output of the compiler.

2) Trace through the nodes of all controlflow paths, in the order of potential execution, while keeping a hierarchical symbol table and noting when each bit was defined (written). Annotate the inputs of each node as it is encountered during the tracing with the node numbers of potential input sources.

3) Move in sequence through the nodes based on node number. Each input source of every node input is processed. The processing annotates the output arcs of their node destinations. The processing also records whether each variable is read, written, or both, in the node that writes the variable.

4) Since subroutine calls have separate input and output lists, prune these lists to indicate variables that are only read or only written.

5) Move in sequence through the nodes based on node number. For each assignment node, determine if it connects to any previously created bus data structure and write the connection; otherwise, specify the beginning of a new bus.

6) Write (as Prolog source code) all annotated nodes to a new disk file.

7) Write (as Prolog source code) all bus descriptions to the same disk file.

4.4 Parallelism Analysis using the Resource Utilization Template (RUT)

The rest of this chapter describes a machine-description analysis technique [POE81B] called a resource utilization template (RUT). The RUT is designed to help maximize the potential parallelism in the final microcode while minimizing synthesis search effort. Although the RUT is not yet implemented as part of UMS, it was based on previous implementation experience [POE80], [POE81A] with V-Compiler [PATGPS81] optimization techniques. Part of this section is derived from [POE81B].

4.1 Motivation

UMS considers microcode synthesis to be a heuristic search (see [STEABBBHS82], [NIL80]) through a large space of microprograms. The search consists of three main parts: 1) Finding ways to perform the semantics of the instruction-set description with the operations available in the microengine, 2) Determining a relatively efficient family of alternative microprograms that perform the desired semantics, and 3) Choosing the most efficient member of that family. Chapters V and VI describe in detail the techniques that find a way for the instruction-set semantics to run on the microengine. This section discusses how the RUT applies to the later two issues.

Synthesis search can be guided by measurements of code quality. The search strategy should also delay the computationally costly assignment of microoperations to microwords (see [AGE76] or [LANDSM80] for a review) until as late as possible. This allows for economical production and evaluation of alternative microcode translations. Generation of high-quality microcode depends particularly on context (most importantly, potential parallelism exploitation). An accurate measurement of code quality must reflect this potential and not rely on code-use context assumptions. The primary motivation behind developing the RUT was the need to express microoperation-placement variability and its potential parallelism exploitation in a microengine. Neither the required number of microwords (or microengine execution cycles) nor the number of microoperations measure potential parallelism in synthesized code fragments.

It would be quite useful to have a measurement of microengine-resource use; such a measurement could guide the search-path choice during microcode compilation. This information may be obtained by translating microcode into hardware microoperations, scheduling these microoperations into microwords, and then measuring the obtained microwords. Traditional microoperation scheduling is computationally expensive. When the number of alternative code-translation search trees increases combinatorically, scheduling each alternative becomes prohibitively expensive.

A minimally useful measurement of resource use would describe the smallest number of microwords needed to contain a microprogram for a machine with maximum potential parallelism. The DAG described earlier would provide such information. Second, it is important to measure resource use in microcode fragments for realistic machines with limited resources. The DAG does not provide this information. The measurement should include an understanding of microoperation-placement variability. This would be used to represent families of microprograms that contain the same microoperations but vary in their microoperation-placement order. Further, this allows isolating the placement-variability dimension from the other dimensions of the microcode-synthesis search space, and considerably simplifies the search. Third, it would be helpful to estimate if two semantically parallelizable microcode fragments can be contained (or overlayed) in the same sequence of microwords. Fourth, if resource constraints prevent placing an additional set of microoperations completely in a sequence of microwords, it would be advantageous to add a microword or two, locally modify our measurements, and continue the analysis. Most importantly, this measurement should be computationally inexpensive compared to complete compilation and scheduling of microcode. This measurement can be used as an estimate, even if it is not completely accurate (accuracy could be greatly improved by

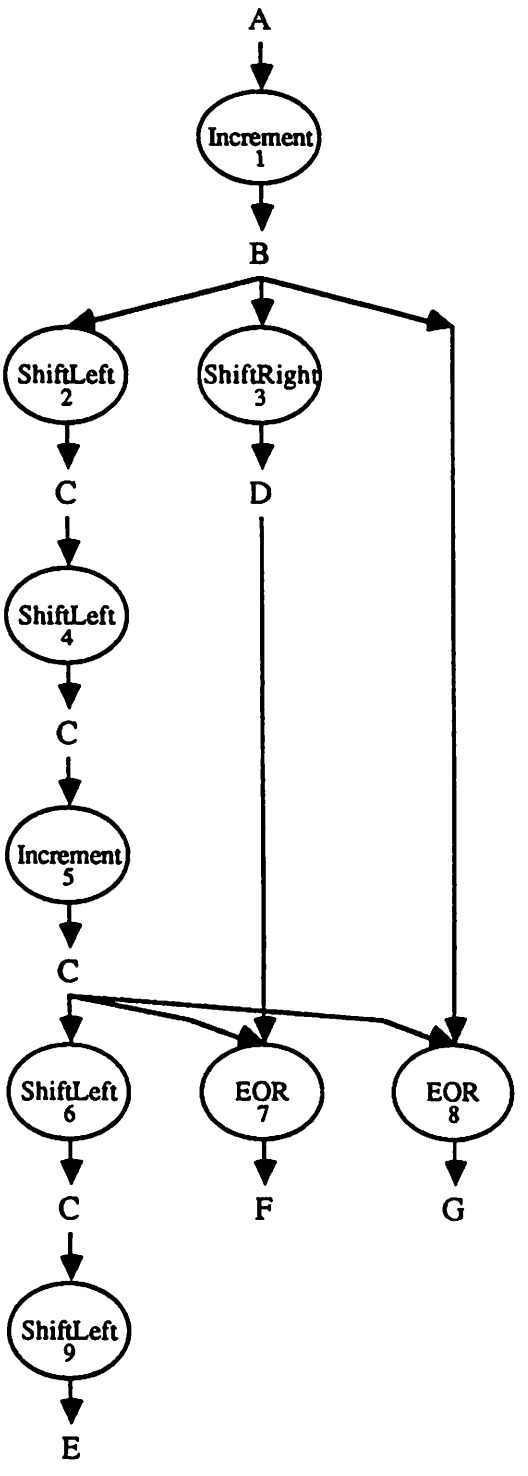
rechecking the microoperation partial ordering). The RUT fulfills all these requirements.

The RUT is a metric-based technique for analysis of the machine description, used to aid selecting the most-efficient microprogram. Referring to a catalog of successful code-generation experiments in RUT form allows applying the experience gained from one part of a microprogram to assist processing other parts. This section first gives examples of placement variability and potential parallelism. The RUT algorithm is described next. Finally, the usefulness of the RUT is demonstrated.

4.4.1.1 Microoperation-placement variability. Microoperation-placement variability is based on two sources of potential parallelism. In horizontal microengines, many microoperations may potentially perform in parallel (simultaneously) in a single time unit, microword, or microcycle. This is potential hardware parallelism. Single operations or statements of the high-level language that describe the instruction set may be implemented as sequences of microoperations. The control-and-data dependencies in these microoperation sequences specify only a partial ordering. The partial ordering often allows arranging the same microoperations into many different valid sequences, some of which may include performing operations in parallel (simultaneously). This is potential microcode parallelism.

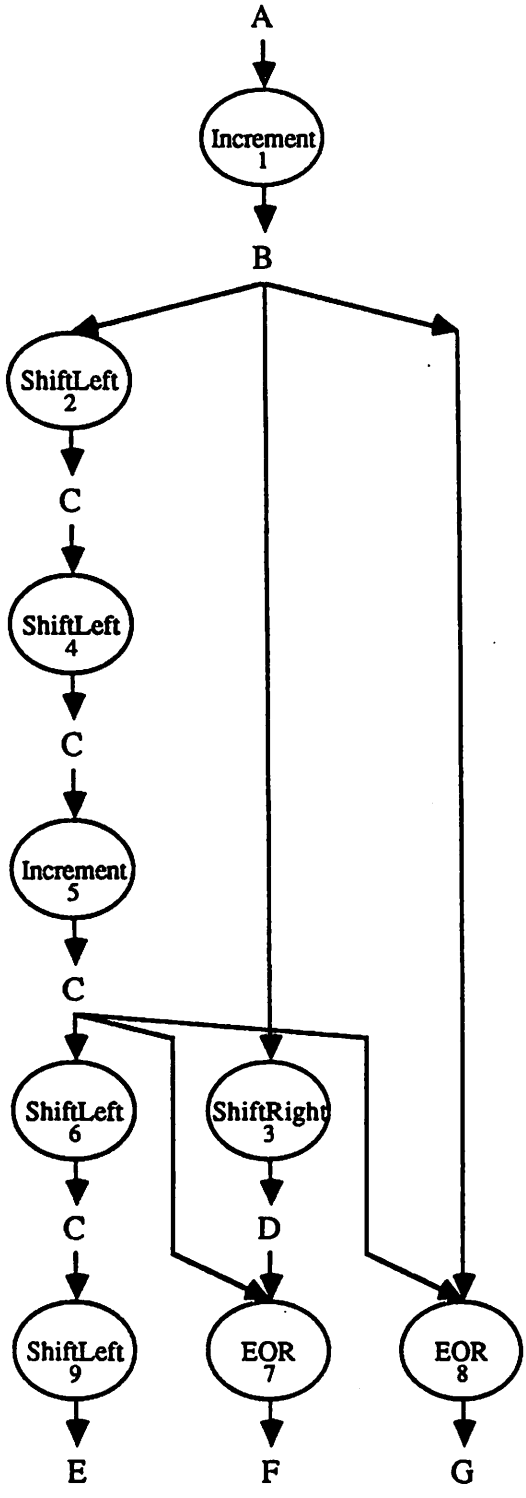
Figures 4.11 and 4.12 show in data-dependency graph form the microoperation-placement variability of the same microprogram. Figure 4.11 shows microoperations 3, 7, and 8 placed as early in the code as data dependencies allow (in levels 1 and 4), while these microoperations are placed as late as possible in Figure 4.12 (levels 4 and 5). The RUT was designed to express variability of placement of nodes based on their data dependencies, such as the extremes shown in Figures 4.11 and 4.12.

4.4.1.2 Microcode-fragment interleaving. A microprogram may contain many unused microoperation "holes." These occur when some of the microword fields and their corresponding resources of the microengine are unused during a cycle. Figure 4.13 shows the same microprogram of Figures 4.11 and 4.12, with the additional constraint that each level of the graph represents a microword that can control independently a single incrementer, shifter, or logic unit. If this code was placed into microwords, the first microword would contain the contents of DAG level 0: an increment operation. The first microword of Figure 4.13 could contain potentially two other types of microoperations simultaneously: a shift and a logic microoperation. However, due to the data dependencies within the code fragment, it is not possible to perform either of these microoperations in the first microword to speed up the microprogram. Similarly, the second microword could contain potentially an increment and a logic microoperation, but no microoperation in the micro-



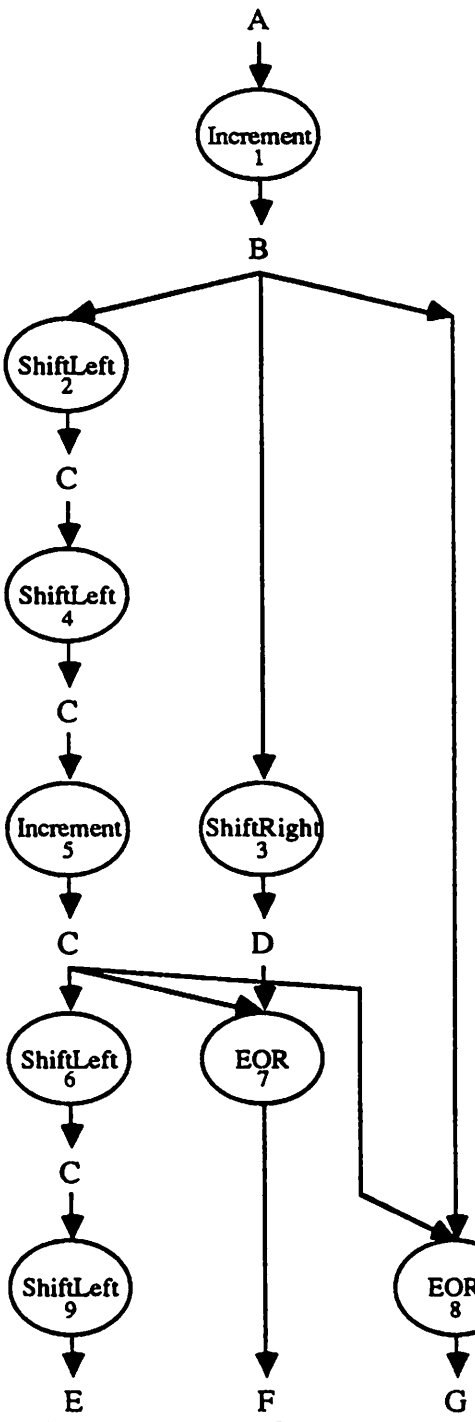
Level	Operation Number	Microoperation Source Code
0	1	B = Increment(A)
1	2	C = ShiftLeft(B)
	3	D = ShiftRight(B)
2	4	C = ShiftLeft (C)
3	5	C = Increment(C)
4	6	C = ShiftLeft(C)
	7	F = EOR(C,D)
	8	G = EOR(C,B)
5	9	E = ShiftLeft(C)

Figure 4.11: A DAG and its source code showing earliest (or highest) level placement. Compare to Figure 4.12 for latest placement.



Level	Operation Number	Microoperation Source Code
0	1	B = Increment(A)
1	2	C = ShiftLeft(B)
2	4	C = ShiftLeft (C)
3	5	C = Increment(C)
4	6	C = ShiftLeft(C)
	3	D = ShiftRight(B)
5	9	E = ShiftLeft(C)
	7	F = EOR(C,D)
	8	G = EOR(C,B)

Figure 4.12: A DAG and its source code showing latest (or lowest) level placement. Compare to Figure 4.11 for earliest placement.



Level or Microword	Operation Number	Microoperation Source Code
0	1	B = Increment(A)
1	2	C = ShiftLeft(B)
2	4	C = ShiftLeft (C)
3	5 3	C = Increment(C) D = ShiftRight(B)
4	6 7	C = ShiftLeft(C) F = EOR(C,D)
5	9 8	E = ShiftLeft(C) G = EOR(C,B)

Figure 4.13: A DAG and its source code, shown with the constraints of one available incrementer, shifter, and logic unit per microword (or level). The DAG shows one version of the earliest (or highest) level placement based on hardware availability constraints. Figures 4.11 and 4.12 contain the same microprogram without hardware constraints and show other placement possibilities.

program could access all necessary input data dependencies available at this point.

The microoperations 1, 2, 4, 5, 6, and 9 of Figures 4.11, 4.12, and 4.13 are connected with data-dependency arcs to form the longest node path chain, which reaches from the top to the bottom of the graph. This longest chain is called the critical path (the term comes from scheduling theory [COFF76]) because it describes the strongest constraints that prevent performing all of the nodes in parallel. Any analysis of the quality of the code produced should include measurements of the number of microoperation "holes" (such as discussed in Figure 4.13) microoperations and microwords. The RUT describes implicitly the constraints of the critical path of microoperations as well as width of the data-dependency graph.

When two microcode fragments are combined one after the other, there is potential to fill some of the microoperation holes from one fragment with microoperations from another fragment. The RUT also provides a way to estimate inexpensively how well two microcode fragments may be interleaved. This estimate can allow more insightful synthesis and improved microcode quality.

4.4.1.3 Microcode-compiler metrics. This section discusses how code quality analysis relates to search strategies in microcode compilers. Conventional microcode compilers (such as described in [PATGPS81],

[SINT80], [DAD80] or [DAVS80]) use an approach that leaves large regions of the microcode design search space unexplored. In general, these compilers apply a two-stage filtering mechanism to prune away search-space regions. First, they generate code sequences that require a minimum number of microoperations, irrespective of some microengine architectural details. Alternative microoperation sequences that are longer than the minimum are usually filtered out. If the compiler considers more than one way to perform a microoperation on a microengine, then these alternatives are called "or lists" [MAL78], although this is rare. Secondly, the compilers optimize these locally optimum sequences into microwords by a process called microcode compaction, or packing (see [LANDSM80] or [AGE76] for a review). This second filter is usually based on computationally economical heuristics. Since exhaustive search is not used, optimal code cannot be guaranteed for microcode compilation.

How the interaction between these two filters affects microcode quality is still relatively unexplored ([VEG82A,B]). In contrast, the unimplemented RUT is a method to estimate inexpensively the results of these two manipulations and their interactions.

Conventional microcode compilers make no attempt to consider systematically the serendipitous interactions (such as the interleaving described above) between microcode fragments. These interactions are

potentially very important, and could be exploited by UMS as described in the remainder of this chapter and Chapter VI. Code phrase evaluation during translation, such as the RUT may perform, has computational efficiency implications related to production design [KNUB70], optimal code-quality proofs [KNUB70], [PEA84], and search-control strategies [GEO79], [STEABBBHS82].

4.4.1.4 Microcode compiler resource models. The microarchitecture descriptions of most other microcode compilers are based on the idea of resources (see [DAS80], [SINT80], [MAL78], [AGE76]). A resource such as an ALU, bus, or register is a part of the microengine that may be used by only one microoperation at a time. Resource allocation involves mapping these resources to operations at particular times. Usually, the microcode compiler author designs the resource model of the microengine and determines which resources are important. In contrast, the UMS microcode synthesizer analyzes the microengine description resources without user assistance. It determines what parts (principally, microword fields) of the microengine may have different values and maps particular values to them. UMS-based analysis is typically at a much finer level of detail than a compiler implementer would attempt, with a corresponding potential increase in code quality.

The compiler within UMS must deal with many aspects of the hardware that conventional compilers do not ever consider. Mallett [MAL78] suggests

only a beginning point for full microarchitecture description for microcode optimization. His thesis specifies that each microoperation include lists of machine resources used for input, storage of intermediate values, and output; functional units required during execution; required clock phases; and the needed microword fields and their corresponding values. Dasgupta [DAS80] also uses a detailed microengine model. An optimizing microcode compiler must posses this detailed information to understand which microoperations cause conflicts when they are put in the same microword. The RUT, described next, automatically processes much of this complexity.

4.4.2 The resource utilization template algorithm

First, the unimplemented RUT is discussed intuitively in regard to conventional models of microengine resources. Next, the complete algorithm is given. Other uses of the RUT are described last.

A RUT-like diagram in Figure 4.14 shows resource use over time for three microengine resources. The resources (an incrementer, a logic unit, and a shifter) are for the microprogram of Figure 4.12. To simplify the discussion, each time-step is assumed to be represented by a microword, which also contains one DAG level (such as shown in Figure 4.13). Many microprograms require more than one microword to contain the contents of a DAG level. In this case, the convenient relationship between DAG level and microword would not

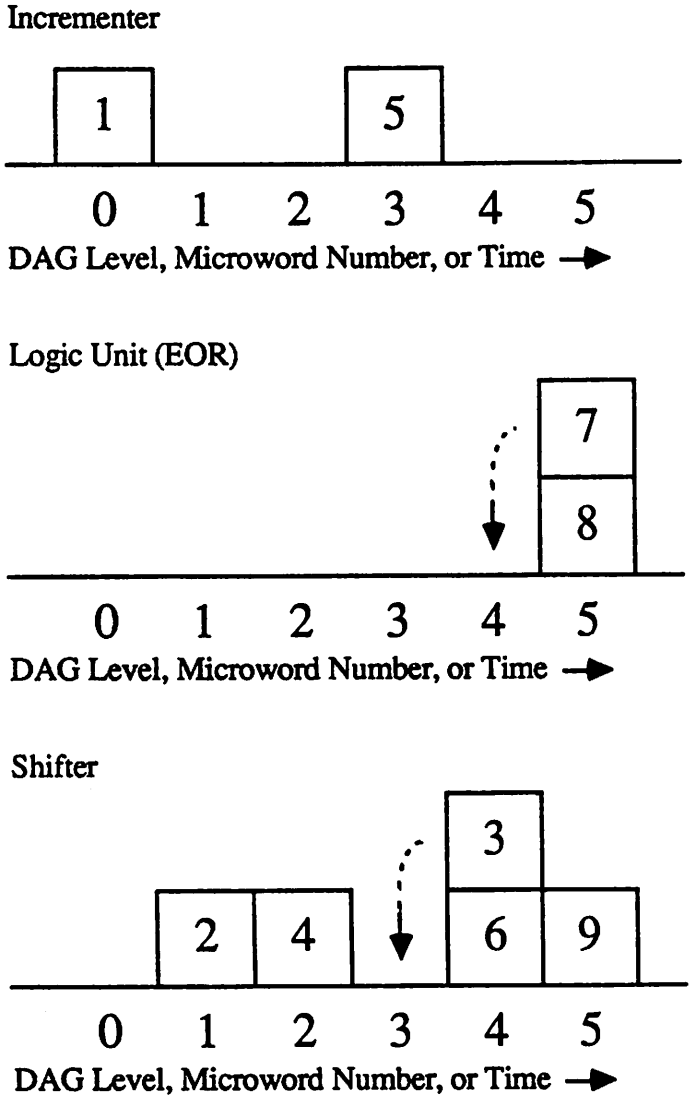


Figure 4.14: Resource use of the microprogram in Figure 4.12. Each time-line indicates when microoperations (shown as numbered squares) use a type of hardware resource. If only one logic unit or shifter is available per microword, then microoperation 7 may be moved to microword 4 and microoperation 3 to microword 3.

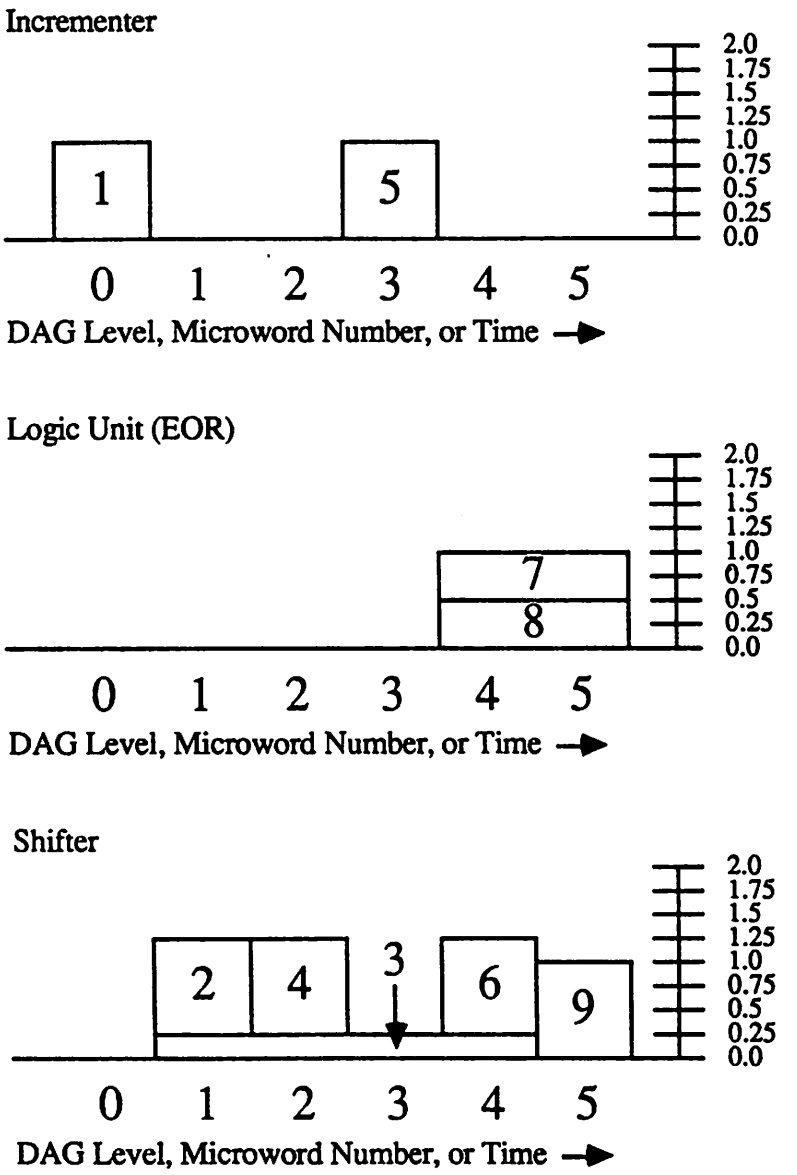


Figure 4.15: Resource usage template for the resource use shown in Figure 4.14 and for the microprogram in Figure 4.12. Each time-line indicates when microoperations (shown as numbered squares) use a type of hardware resource. Both microoperations 7 and 8 can be located in either microwords 4 or 5, so they are shown as using 1/2 of a unit of resource over the two microwords.

occur since more than one microword would be needed for each DAG level, resulting in less tidy diagrams.

By visual inspection of Figure 4.14, it is easy to observe that microword 4 uses the shifter twice while microword 3 does not use the shifter at all. If the shifter resource can be used only once per microword, then a natural idea is to move microoperation 3 from microword 4 to microword 3. Similarly, if the logic unit can be used only once per microword, it is a natural idea to move microoperation 7 from microword 5 to microword 4. The RUT automates this type of spatial reasoning. Although the type of diagram shown in Figure 4.14 is the origin of the RUT idea, it does not show how the shifter in microoperation 3 can be placed in microwords 1, 2, 3, or 4 (see Figures 4.11 and 4.12).

An actual RUT similar to Figure 4.14 is shown in Figure 4.15. When a resource can possibly be used in N different microwords (as constrained by data dependencies), it is represented in the RUT as using 1/N of a unit resource in each of those microwords. Inspection of the distance between the earliest placement (such as Figure 4.11) and the latest placement (such as Figure 4.12) determines into how many different microwords a microoperation may be placed. Note that in Figure 4.15, the placement variability of microoperation 3 is represented as requiring 1/4 unit of resource over microwords 1 through 4. Similarly microoperations 7 and 8 are shown as requiring 1/2 resource each over microwords 4 through 5. The length of the critical path is described

implicitly by the number of levels (and in this example, microwords) used in the RUT. The detailed algorithm for this spatial reasoning is discussed after illustrating how resource bottlenecks can be detected.

4.4.2.1 Using the RUT to find bottlenecks. One important use of the RUT is to determine where and when there are not enough microengine resources available to run a microprogram in a fixed number of words. The RUT in Figure 4.15 is shown with six available microwords. A scan of each resource determines how well the microengine satisfies the demands on each resource. Figure 4.14 suggests placing microoperation 3, which uses the shifter, in microword 3. Inspection of Figure 4.15 also supports this suggestion, as it also indicates that microoperation 3 can be placed in any of microwords 1, 2, 3, or 4. This intuitive reasoning is implicit in the RUT algorithm.

The following dialog discusses the action of the RUT bottleneck detector as it scans from left to right along the shifter-resource RUT at the bottom of Figure 4.15. A later section will formalize this process. Consider the shifter resource in Figure 4.15. There are no net accumulations of resource requirements and no delayed resource uses after microword 0 and before microword 1. There is a request to use 1.25 unit (1 unit from microoperation 2, .25 from microoperation 3) of the shifter within microword 1. Since only one unit of shifter use is available after microword 1, this leaves a "debt" resource use of .25. Similarly, after microword 2, the shifter resource debt has increased

to .5. Since microword 3 of Figure 4.15 uses only .25 of a shifter resource, the remaining .75 available more than satisfies the resource debt of .5, leaving a shifter resource surplus of .25. This surplus is balanced out after microword 4. Microword 5 causes no change from zero of the resource debt. As is described in detail below, a local bottleneck occurs whenever the resource debt is equal to or greater than twice the amount of available resource. A global bottleneck occurs when the resource debt is greater than zero at the end of a scan. A bottleneck indicates that at least one additional microword is required to place the microprogram into microwords. The incrementer is relatively uninteresting within the example of Figure 4.15 and is not discussed here.

Similar processing for logic-unit use in Figure 4.15 determines that a bottleneck does not exist. Note that compared to other constraint-based microcode-analysis techniques (such as [HENMA83]), the RUT does not decide whether microoperation 7 is in microword 4 or 5.

The RUT is only an estimate of microoperation placement and not guaranteed to work in all situations. However, it seems to be sufficiently accurate for much of the code-generation analysis needed for the UMS system.

4.4.2.2 Definition of the RUT. The definition of the RUT (from [POE81B]) does not require a specific resource model. Although the model used by the UMS system is that of microword fields, the definition described here is general enough to be compatible with historically used resource models

(see [AGE76]). Let the microoperations be labeled as M1, ... MZ. Resources are labeled R1, ... RY, which are available in amounts A1, ... AY. The level (or microword) currently under inspection is labeled L, ranging from zero at the beginning to N at the last level.

The RUT uses the following functions. FIRST(Mi) describes the first level that the microoperation can be placed into (see Figures 4.13 and 4.14). The earliest microword is number zero, and increasing integers label subsequent microwords. LAST(Mi) describes the last potential level that the microoperation can be placed into. These two functions are determined from inspection of the DAG (directed-acyclic graph) of the microprogram. Note that these placement variabilities would otherwise have to be measured for conventional microword packing and do not result in net extra computational effort by UMS.

CONDPLACE is the conditional placement of microoperation Mi in level L using resource Aj, and is defined as:

$$\text{CONDPLACE}(i, L, j) =$$

$$\begin{cases} 0 & \text{if } L < \text{FIRST}(Mi) \text{ or } \text{LAST}(Mi) < L; \\ (Aj) \,/\, (1 + \text{LAST}(Mi) - \text{FIRST}(Mi)) & \text{if } \text{FIRST}(Mi) \le L \text{ and } L \le \text{LAST}(Mi). \end{cases}$$

CONDPLACE's value is zero when the microoperation Mi is to be placed in a microword earlier than the first legal level, as determined by control and data

dependencies. CONDPLACE is similarly zero for attempted placement after the last legal level. Otherwise, the value of CONDPLACE is proportional to Aj (the amount of resource available) divided by the number of microwords the microoperation may be placed into.

SUMCOND is the summation of all conditional uses at a level L of a resource Rj, for all microoperations Mi. It is defined as:

$$\text{SUMCOND}(L, j) = \sum_{i=1}^{z} \text{CONDPLACE}(i, L, j).$$

LEVEL_OK determines whether microoperation Mi legally can be placed within level L while using resource Rj:

$$\text{LEVEL\_OK}(i, L, j) = \begin{cases} 1 & \text{if } \text{FIRST}(Mi) \le L \le \text{LAST}(Mi); \\ 0 & \text{otherwise.} \end{cases}$$

OPS_IN_LEVEL counts the number of microoperations using resource Rj that may be placed in level L:

$$\text{OPS\_IN\_LEVEL}(L, j) = \sum_{i=1}^{z} \text{LEVEL\_OK}(i, L, j).$$

PERMITTED determines if a microoperation Mi using resource Rj may be legally located within both the current level L and the previous level:

PERMITTED(i, L, j) =

1
  if (LEVEL_OK(i, L-1, j) = 1 and
    LEVEL_OK(i, L, j) = 1 and
    L > 0);

0
  otherwise.

FORWARD counts the number of microoperations that legally may be placed in the previous and current levels:

$$\text{FORWARD}(L, j) = \sum_{i=1}^{z} \text{PERMITTED}(i, L, j)$$

if $L > 0$.

To describe how the amount of resource Rj used within one level L affects that of another level L-1, the accumulation function ACCDEBT is used:

ACCDEBT(L, j) =

minimum(OPS_IN_LEVEL(i, L, j),
(SUMCOND(0, j) - Aj))
while L = 0, SUMCOND(0, j) > 0, and LOCALB(L, j) = FALSE;

maximum(-FORWARD(L, j),
minimum(FORWARD(L, j),
(ACCDEBT(L-1, j) + SUMCOND(L, j) - Aj)))
while L > 0, SUMCOND(0, j) > 0, and LOCALB(L, j) = FALSE;

0
otherwise.

If the value from this function is positive, then a resource debt has been incurred and use of the resource must be satisfied later (in the following levels

and microwords). If the value ACCDEBT function is negative, then there has been more opportunity to use the resource than was immediately needed.

Consider ACCDEBT for the first level of the RUT (where L = 0), where there are requests for use of the resource (SUMCOND > 0) but without a local bottleneck (LOCALB = FALSE). Resource-debt accumulation is the difference between the amount requested (SUMCOND) and the amount available (A). When the amount requested is more than the amount available, then ACCDEBT is positive. The amount of debt is limited to the amount of resource available, so the difference between SUMCOND and A is bounded above by OPS_IN_LEVEL.

ACCDEBT is also defined for other than the first level of the RUT (where L > 0). Here the accumulated debt ACCDEBT from the previous level is increased by the amount of resource required (SUMCOND) but decreased by the amount of resource available (A). The amount of resource debt or surplus that can be used at a different time is constrained by the FORWARD terms; ACCDEBT's new value is bounded above and below by the number of resource requests that can be satisfied between the current level and the previous one.

A local bottleneck LOCALB(L, j) for level L and resource Rj is defined as:

LOCALB(L, j) =

while L = 0,

TRUE if 2*Aj ≤ SUMCOND(0, j),

FALSE if 0 ≤ SUMCOND(0, j) < 2*Aj;

while L > 0,

TRUE if LOCALB(L-1, j) = FALSE and either:
2*Aj ≤ SUMCOND(L, j)
- or -
2*Aj ≤ ACCDEBT(L, j)
- or -
(FORWARD(L,j) = 0 and
0 < ACCDEBT(L, j)),

FALSE if:
LOCALB(L-1, j) = FALSE and
0 ≤ SUMCOND(L, j) < 2*Aj and
ACCDEBT(L,j) < 2*Aj.

When requests to use the resource (SUMCOND) for the first level are equal to or greater than twice the amount available, then a local bottleneck occurs and LOCALB is TRUE. If the requested amount (the amount that can be satisfied immediately plus the amount that can be carried over to later microwords) is below this limit, then a local bottleneck has not occurred (LOCALB = FALSE).

Consider LOCALB for other than the first level (L > 0). There must not be a local bottleneck (LOCALB must be FALSE) at every previous level because the RUT locates only the first bottleneck, and the RUT cannot detect any subsequent bottlenecks. LOCALB can be TRUE if the requested amount of resource is greater than or equal to twice the amount of available resource Aj. This is the amount that can be satisfied within the microword and can be subsequently satisfied. Secondly, if the accumulation of unsatisfied resource requests (ACCDEBT) is greater than or equal to the same boundary value (2*Aj), then LOCALB is TRUE for the same reason. Finally, LOCALB may be TRUE when no microoperation-placement opportunities are shared with the previous level (FORWARD = 0) although the accumulated resource debt requires such sharing (0 < ACCDEBT).

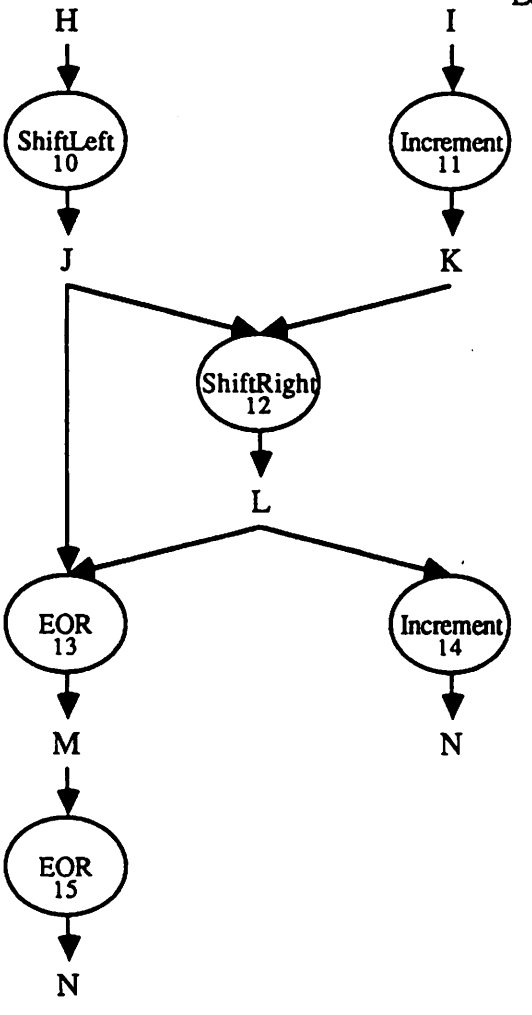
A local bottleneck has not occurred (LOCALB = FALSE) when the request for resources (SUBCOND) is within resource limits, a previous bottleneck has not occurred (LOCALB = FALSE), and the resource debt (ACCDEBT) is also within this limit. A global bottleneck occurs when, after the processing of the last level, some resource debt remains (ACCDEBT(L, j) > 0 for any j). Now that the RUT algorithm has been formally described, other uses for it will be discussed.

4.4.2.3 Using the RUT to test microprogram interleaving. Microprogram interleaving is a technique that tests the unused microword fields

of every other microword to determine if two microprogram fragments may overlap partially. To illustrate microword interleaving, consider in Figure 4.16 the microprogram designed to show this affect and its corresponding RUT shown in Figure 4.17. This microprogram is shorter (a critical path of only four levels or microwords and a total of six microoperations) than the first microprogram of Figure 4.15 (a critical path of six levels or microwords and a total of nine microoperations).

When the RUT of the second microprogram (Figure 4.17) is interleaved with (i.e., layed on top of) the RUT from the first microprogram (Figure 4.15), the RUT of Figure 4.18 results. A local bottleneck in this combined RUT becomes visible immediately in the top left of Figure 4.18. Here, two units of incrementer resource are required within the first microword. Since the threshold for a local bottleneck is a request for twice the available resource, this becomes the first local bottleneck. In the RUT algorithm, this bottleneck is detected within LOCALB because SUMCOND = 2*Aj while L = 0.

If the motivation to use the RUT is simply to test for bottlenecks, then processing may end at this point. However, the motivation often is to determine how many microwords are required to contain two interleaved microprogram fragments. In this case, the local bottleneck must be removed before further processing and measurements can occur.



Level	Operation Number	Microoperation Source Code
0	10	J = ShiftLeft(H)
	11	K = Increment(I)
1	12	L = EOR(J,K)
2	13	M = EOR(J,L)
	14	N = Increment(L)
3	15	P = Increment(M)

Figure 4.16: A DAG and its source code. The DAG shows a complementary microprogram to Figures 4.11, 4.12, and 4.13.

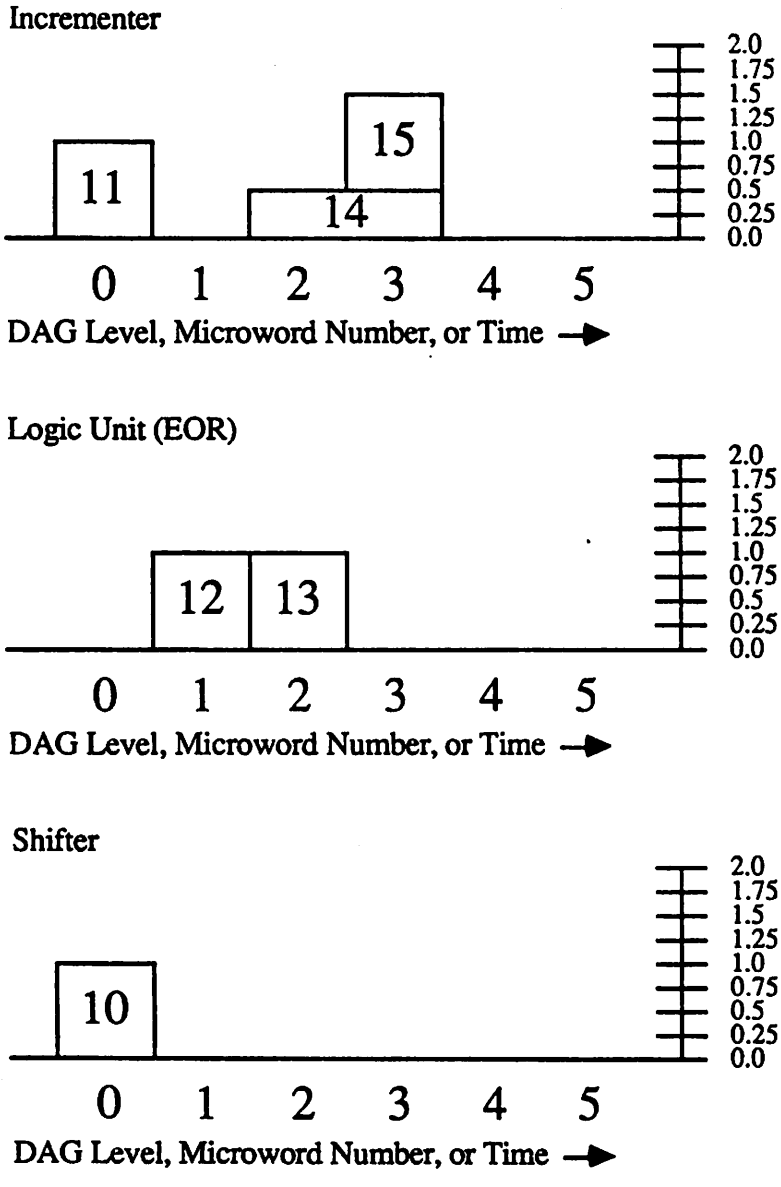


Figure 4.17: Resource usage templates of resource use in Figure 4.16. Each time-line indicates when microoperations (shown as numbered squares) use a type of hardware resource.

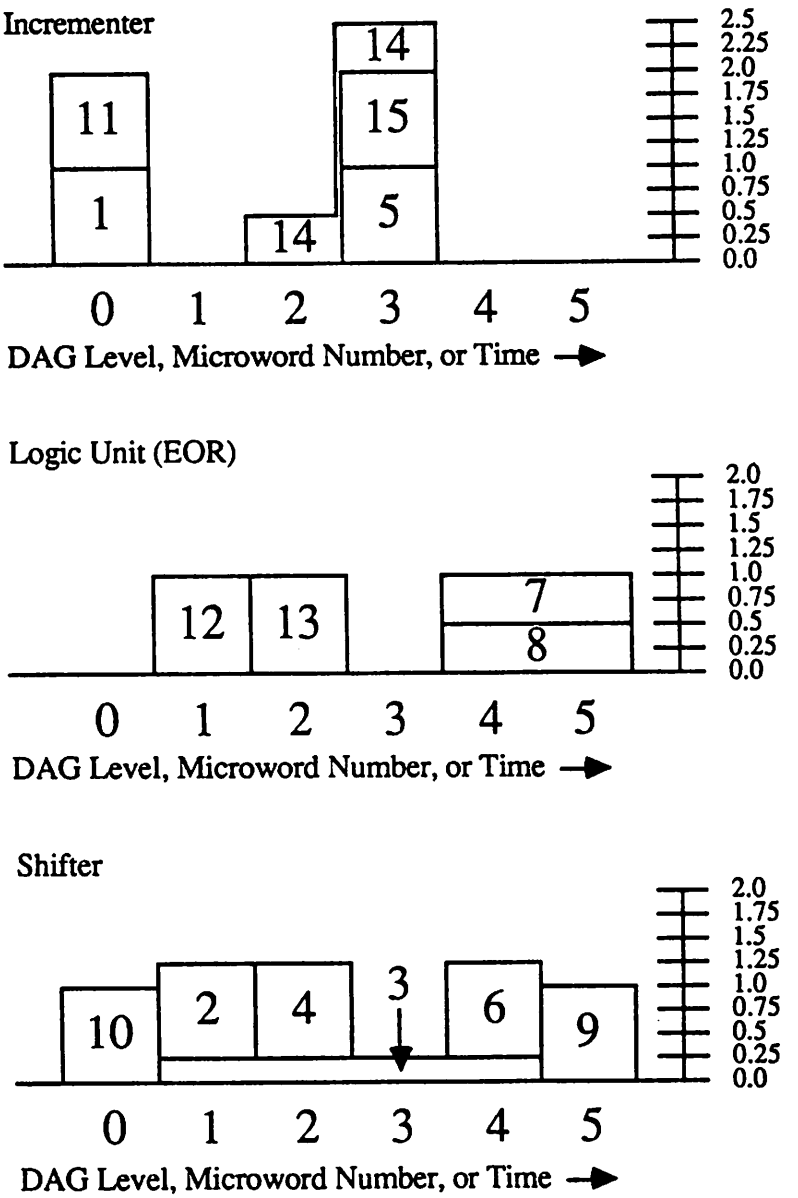


Figure 4.18: Overlayed resource usage templates of Figures 4.15 and 4.17. Each time-line indicates when microoperations (shown as numbered squares) use a type of hardware resource.

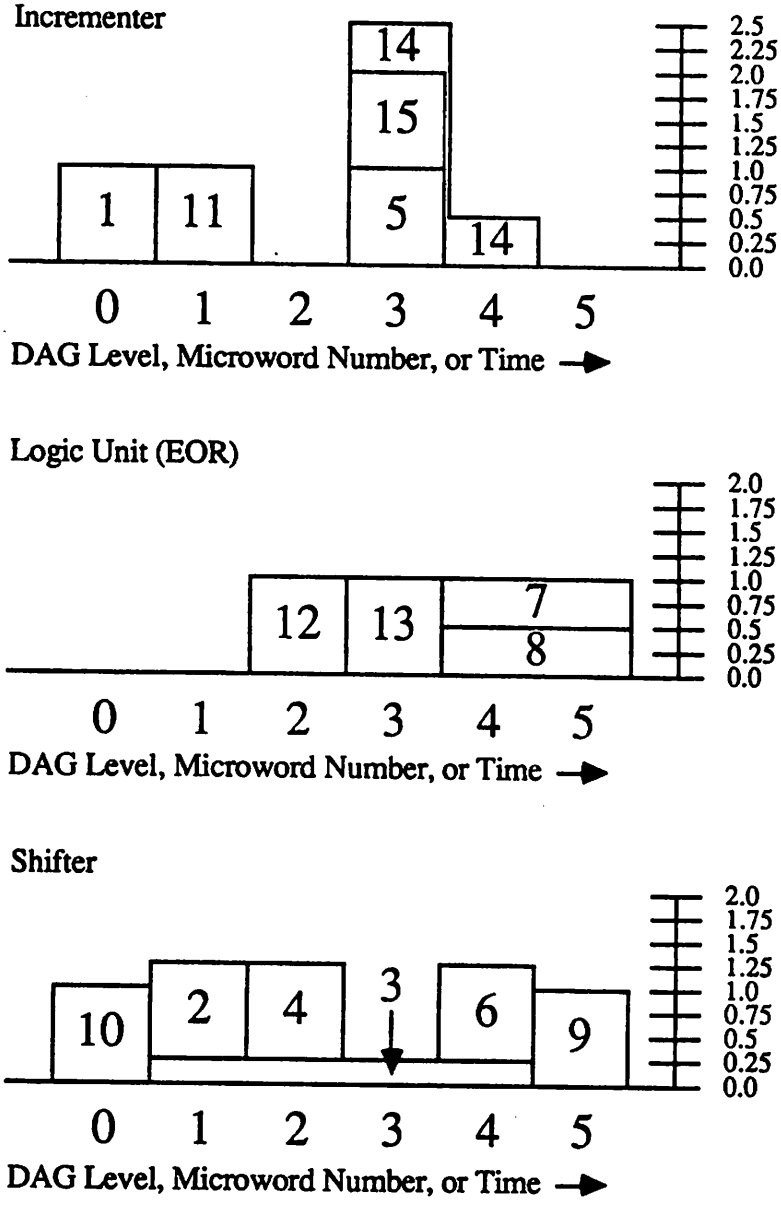


Figure 4.19: Overlayed resource usage templates of Figures 4.15 and 4.17, with an extra microword between levels zero and one of Figure 4.17 to overcome a local bottleneck. Each time-line indicates when microoperations (shown as numbered squares) use a type of hardware resource.

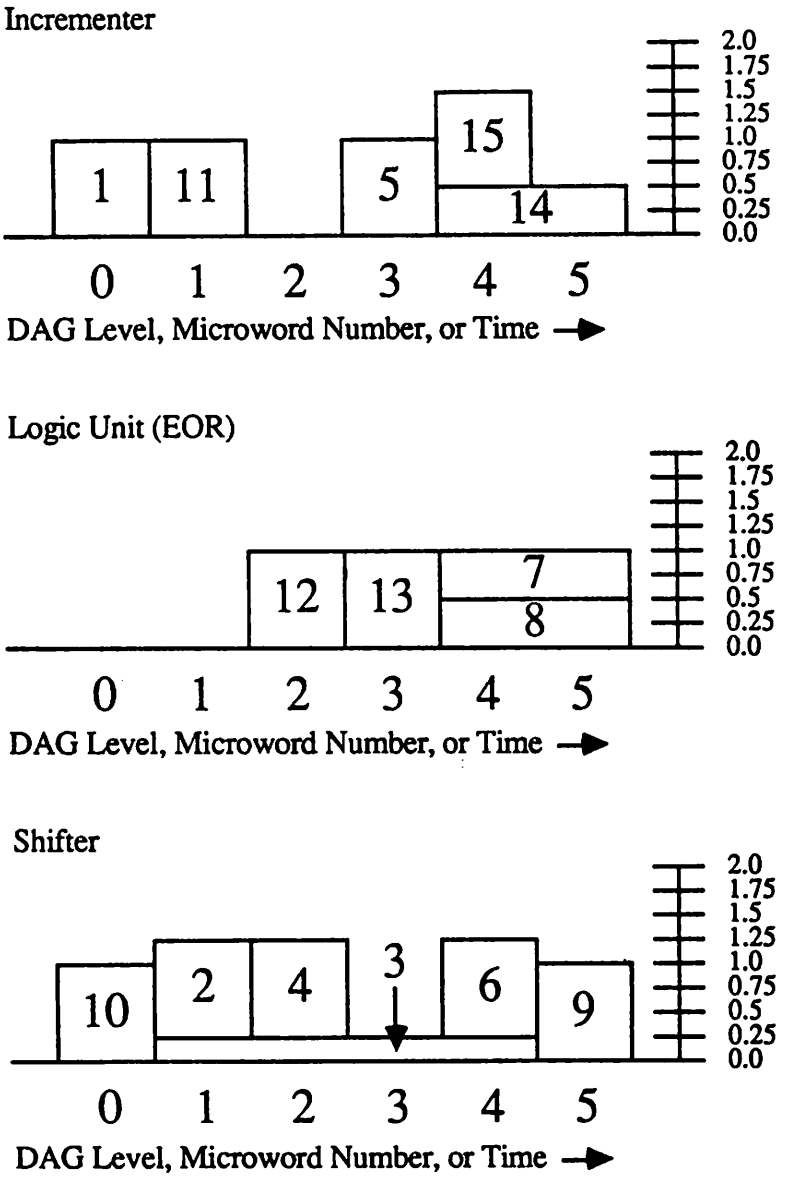


Figure 4.20: Overlayed resource usage templates of Figures 4.14 and 4.15, with an extra microword between both levels zero and one and between levels one and two of Figure 4.17 to overcome local bottlenecks. Each time-line indicates when microoperations (shown as numbered squares) use a type of hardware resource.

The heuristic used to rectify a local bottleneck during microprogram interleaving is to elongate the shortest fragment. This is done by not placing the bottleneck-causing microoperation of the shortest microprogram into its current microword or level. This relieves the bottleneck within the current microword. The bottleneck-causing microoperation and its subsequent microoperations are then placed into subsequent microwords. The net effect is to elongate the microprogram because it contains a "hole" where the "pressure" from the local bottleneck occurred.

In the context of Figure 4.18, the first local bottleneck is for incrementer use in microword 0. The heuristic solution for solving this bottleneck, which was found during microprogram interleaving, is shown in Figure 4.19. The RUT of microoperations 11 through 15 of the microprogram is elongated by delaying microoperation placement until microwords 1 through 4 (instead of in microwords 0 through 3). This has the effect of prefixing the microprogram with a microword containing only microoperation 10.

When bottleneck detection continues, the next one is found in microword 3 (see Figure 4.19). Similarly, moving microoperations 14 and 15 down one microword solves this local bottleneck, and the modified RUT looks like Figure 4.20. This final RUT of the combined microprogram fragments does not contain any additional local bottlenecks. This process allows the UMS synthesizer to determine (without computationally expensive microword packing) that

interleaving the six-level RUT of Figure 4.15 and the four-level RUT of Figure 4.17 probably will be successful. It also determines that when the two microprograms are combined, they would require no more microwords than for the first microprogram alone.

4.4.2.4 The RUT as a microarchitecture description. Up to now, the RUT has been discussed without regard to a specific formal machine model (see [DAS80], [MUE80A]). However, the RUT along with the original procedural ISPS language description of the hardware is sufficient for the UMS system. Whereas microcode-generation (especially compaction) systems conventionally specify the hardware resources (such as an incrementer, logic unit, or shifter, as shown in Figures 4.14 through 4.20) that perform the actual processing, the RUT of the UMS system is intended to use nodes of the hardware-description graph. By design ([BANR82], [AND80], [AGRR76], [HUS70]), these graph nodes nevertheless usually have strong correlations with both individual hardware units and their controlling microword fields.

4.4.2.5 Types of microarchitecture resources. There are two main types of resources within the nodes of the hardware description. These nodes specify 1) the types of information processing (such as of the ALU) to be done and 2) the types of state storage (such as memory and registers) to be used. Both of these resource types could be allocated within UMS by the RUT.

In practice, information-processing nodes do not become a scarce resource. Instead, the input or output assignment nodes or microword-field values become scarce. Consider an ALU that can perform both addition and subtraction. The nodes that can perform these actions could be allocated to two different microoperations simultaneously. However, the nodes that route the operands into the ALU are individually necessary for both of these arithmetic operations but are likely to generate a bottleneck. If the input or output nodes do not create a resource bottleneck, then a bottleneck would be detected in the attempt to assign different values to the microword field that specifies the type of ALU operation. Note that in this case, it is irrelevant for synthesis whether the same or entirely separate parts of the hardware perform the arithmetic operations. Since the hardware description describes exactly what behavior the microengine is capable of, these types of hardware-design details are unimportant. Since the RUT is based on use of a control-and-data-dependency graph, it automatically handles the "equal edges" (described by Fisher [FIS79]), in which two microoperations may be assigned to different levels and may be placed into the same microword.

4.4.2.6 Manipulation of RUT time scale. Typically, the use of a RUT is bound to a specific time scale. However, the time scale may be manipulated, as was illustrated in the section on microprogram interleaving. In that example, when a section of code needed the use of two incrementers per

DAG level, the DAG was manipulated for resource use on a microengine that had only one incrementer. The number of levels of the DAG was increased, and the nodes and arcs from the problem DAG level were placed in adjacent levels. This technique allows stretching or contracting an entire RUT or portions of it to meet resource constraints.

Examples of the RUT in this section suggest that it will be a resource-model independent technique to estimate the interactions of resource constraints and microcode data dependencies on individual and combined microcode fragments.

4.5 Code-Generation Catalog

Once the compiler has translated source code incrementally to the microoperations available in the microengine, a catalog of alternative translations and their costs may be produced. Use of alternative translations has been shown to improve microcode significantly ([TOKTTY78]). The current version of UMS uses only a very simple type of catalog. In the future, a RUT will be associated with each code-generation alternative, and its length will indicate the number of microwords needed. This future design of the catalog is described here.

The organization of the catalog is based on the node structure of the instruction-set description fragments. When an unmodified fragment of the instruction set can produce microcode, the index for that entry in the catalog is simply the source-code structure. Otherwise, the index for an entry is the set of productions used by UMS to generate the satisfactory microcode.

An instruction-set description fragment often can be modified into more than one form. For horizontal microengines, many microoperations may be placed into each microword. Furthermore, many permutations of placement of microoperations into microwords often preserve the partial ordering of the control and data dependencies. A RUT is generated for each alternative object-code sequence. However, only those alternatives different enough to be interesting need to be kept. The catalog is useful in deciding (by attempting microcode fragment-interleaving) which sequence of microoperations would be the most efficient, based on the sequence of neighboring source-code statements. For two adjacent phrases in the high-level language, translations may be chosen from the catalog so that their corresponding object code may be partially interleaved. The use of the RUT and catalog allows the generation of highly compact code by a process likely to be more efficient than other compilation techniques requiring extensive trial and error.

This chapter has discussed techniques to procedurally describe (in languages such as ISPS) and later extract information about hardware, the

instruction set, and microcode resource-use constraints. Subsequent chapters will describe how these descriptions are used for microcode synthesis.

CHAPTER V

MICROCODE DATAFLOW-SYNTHESIS HEURISTICS

5.1 Introduction

This chapter describes the software kernel of the UMS inference engine: the microcode dataflow-synthesis heuristics. Dataflow synthesis is a highly automated form of code generation that determines which microoperations a microprogram performs. Dataflow synthesis is similar to the controlflow synthesis described in chapter VI which determines the order of microoperations in the microprogram for the available hardware.

The scope of microcode-synthesis semantics is much narrower than that conventionally attempted with general program synthesis. The semantics of microcode is usually simple; often, the only data types are integers or bitfields, and the operations available in the ALU (arithmetic logic unit) are typically integer arithmetic and bitwise logical operations. Floating-point arithmetic operations, polynomial expansions, or character-string manipulations are specified algorithmically in the instruction-set description. Other common synthesizable target-program data structures, such as sets, queues, arrays, and graphs, are similarly beyond the scope of microcode semantics. However, these may be similarly algorithmically specified by using lower-level primitives

in the instruction-set description. The complexity of microcode comes from the large number of decisions needed to control typical microarchitecture datapaths and the potential parallelism.

Previous code-generation generators have been based on machine models designed specifically for code generation. The author of the model must carefully control the machine model style to ensure successful pattern-matching between the model and the generated code. In contrast, UMS accepts a machine model written in almost any style. Previous work in automatic parser generation, which is sometimes erroneously called compiler-compilers, is not relevant to dataflow synthesis because of the fixed input syntax of UMS. Most previous models of semantics in code-generation generators are based on parse trees, which obscure some important data dependencies (see [GANA80], [GANAF81A], [GANAFH82], [GANAF84]). These dependencies become obvious in the graph form that UMS uses. Using the techniques described in this chapter, UMS can also deal with bit-width mismatches (unlike most prior code generator generation efforts, see for example [CATT78], [CATNL79]).

The inference engine is written totally in Prolog ([CLOM81], [POENPS84]), which is based on unification and has an embedded relational database capability. It is widely argued [FEIM83], [VERI84] that the built-in search strategy of conventional Prolog implementations [CLAT82], [POENPS84]

makes artificial intelligence programming intrinsically inefficient or awkward. However, the results from this work are consistent with other expert system implementation efforts [MIZ83], [OKE83], which have shown Prolog to be as computationally efficient as other artificial intelligence programming languages. In fact, this experience has found Prolog's built-in search strategy to be very convenient for expert-system implementation [POE84] and algorithm development.

After a brief overview of related program-synthesis research, this chapter presents the general dataflow-synthesis algorithm. Pattern-matching based synthesis techniques for UMS's three graph structures (nodes, arcs, and variables) follows. UMS's graph-based program transformations and their implementation are described last.

5.2 Related Program-Synthesis Research

Synthesis of general-purpose programs has yet to become a useful tool. The specialized success shown here with microcode synthesis hopefully will help direct synthesis toward becoming a practical tool. Whereas all work in program synthesis generates code from complete specifications, some work tries to deduce algorithms from mathematical relationships (see [BARS79], [DAR78], [DAR81]). In UMS, microcode algorithms are completely supplied in

the instruction-set description, and it may be argued that UMS techniques are like goal-directed program transformation (see [STAHKN76], [STAKN76], [JET79]). UMS employs search to choose an appropriate expansion of an algorithm for a particular microarchitecture and is intended to be guided by the RUT.

Previous work with the MIXER system [SHIS83] (reviewed in Chapter II) has shown that a manually guided transformation-based system can produce microcode. MIXER uses three types of transformation rules (supplied by a user for a specific microengine) called macroknowledge, semantic knowledge, and microknowledge. These describe explicitly how to translate one source microprogram for a specific microengine. By contrast, UMS automatically discovers the same knowledge by inspecting the hardware description.

5.3 Synthesis Algorithm

This section overviews the various algorithms used by the inference engine for dataflow synthesis. After discussing the general strategy and data structures, the synthesis order for instruction-set description fragments is described. Then an outline that describes the actions performed during synthesis of an instruction-set fragment is presented.

5.3.1 General strategy

One main principle of the UMS design is that a procedural description of an instruction set (see [SIEBN82] page 125) is an accurate, but inappropriately implemented, microprogram. The challenge for UMS is to apply program transformations to modify this untargeted microprogram code for use on the target microengine. UMS employs a production system as the code-modification mechanism. This production system contains many rules, each consists of "before" and "after" code images. Insight about which transformations should be applied where is obtained from analysis of the target hardware description (see [SIEBN82] page 224 for the AM2901-based microengine description used to experiment with UMS). Because UMS accepts arbitrary hardware models, hardware description input may be presented at different levels of detail. A more detailed hardware model allows for more cleverly produced microcode, at the cost of extra synthesis effort.

The synthesis search space is characterized by a wide branching factor and long individual branches. A typical synthesis task requires many subtasks (code generation for the individual instruction-set description fragments) that are largely independent but have some constraints (such as variable bindings) in common. The microcode-synthesis search space can decomposed into three interrelated subtasks: 1) Determine how to change the functions in the instruction-set description to those available on the hardware, 2) Search the

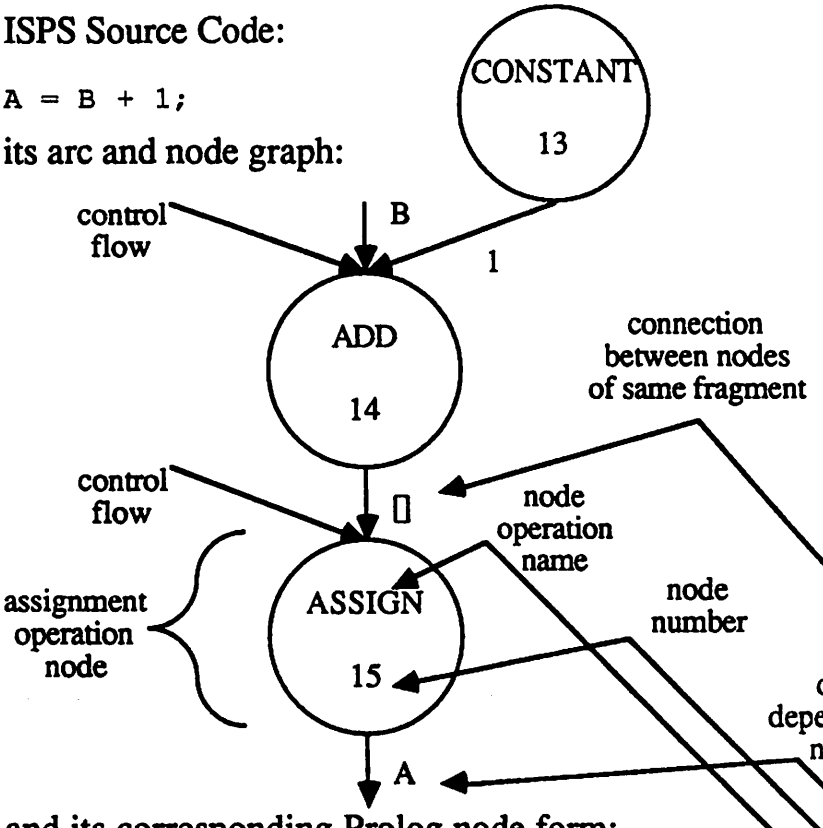
hardware for ways to realize these functions, and 3) Search the resulting microprogram to optimize for speed and space. The kernel of the UMS software, called the inference engine, manages search with cost analysis. This analysis can cause a search path to be abandoned and later possibly resumed. The inference engine tries to find (as soon as possible) a search path that will fail. Such a failure triggers a shift in search direction. A major thesis contribution is the design of an inference engine to make this search practical.

The output of the synthesizer's ISPS compiler is a set of Prolog 4 tuples called nodes. Figure 5.1 shows both the conceptual graph form and the actual data structures used by the synthesizer. The nodes of the hardware description and instruction-set description are of identical form, except for their respective main functor names: "hwnode" and "instnode." These nodes, shown in Figure 5.1, have four main parameters. These are a node operation type, then a list of inputs, a list of outputs, and last an unique node number. Each input or output list consists of a sublist for each operand. Each of the sublists consists of an operand name, a list of attributes, and a list of arc connections to other nodes. A compiler postprocessor creates the arc connections, which represent control and data dependencies. Conceptually, the combination of a single arc connection, the list of attributes, and the operand name is called an arc. Both the source and destination nodes of a data dependency in Prolog contain symmetric arcs, which redundantly specify the same data dependency. An

ISPS Source Code:

```
A = B + 1;
```

its arc and node graph:



and its corresponding Prolog node form:

```
hwnode('CONSTANT'                                          /* line 1  */
       [],                                                 /* line 2  */
       [[1,[],[14]]],                                      /* line 3  */
       13).                                                /* line 4  */
hwnode('ADD',                                              /* line 5  */
       [                                                   /* line 6  */
         [_,[['CHOICE',_,'SUBRDEF']],[12]],                /* line 7  */
         ['B',[],[6,11]],                                  /* line 8  */
         [1,[],[13]]]                                      /* line 9  */
       ],                                                  /* line 10 */
       [[[],[],[15]]],                                     /* line 11 */
       14).                                                /* line 12 */
hwnode('ASSIGN',                                           /* line 13 */
       [                                                   /* line 14 */
         [_,[['CHOICE',_,'SUBRDEF']],[12]],                /* line 15 */
         [[],[['FunctionName',_,'ADD']],[14]]              /* line 16 */
       ],                                                  /* line 17 */
       [['A',[['WIDTH',_,[0,3]]],[23]]]                    /* line 18 */
       15)                                                 /* line 19 */
```

Figure 5.1: The relationship between a fragment (single statement) of ISPS source code (top), its arc and node graph (middle), and its Prolog node form (bottom).

operand name of "_", which is the anonymous logical variable of Prolog, means a controlflow arc. An operand name of "[]" represents a special type of data dependency that is an unnamed direct link to the other nodes that make up an original source-code fragment. An operand with any other name represents an arc of a source-code data dependency that involves a variable of the ISPS code with the same name. Matching these data structures are the primary activity of the inference engine.

The graph representation describes the instruction set at a very fine level of detail. A single ISPS source-code statement may consist of one or more function nodes. Sets of nodes, corresponding to a single source-code statement, are synthesized together as a group or fragment. For example, the ISPS statement "A = B + 1;" is represented as three separate nodes, one for the constant, one for addition, and another for assignment (see Figure 5.1). Arcs represent the data dependencies for A, B, 1, controlflow, and connections between the three nodes. This choice of description granularity allows breaking apart complex expressions into small units. These units can then be reasoned about individually and identically, regardless of the size of the expression they came from.

5.3.2 Determination of synthesis order

The control strategy of the inference engine (described later in this chapter) is powerful enough to alter any previously performed synthesis.

However, this power can cause a decision to thrash back and forth between two values. Carefully choosing the sequence of synthesized instruction-set fragments reduces the likelihood that such a problem will occur. This section describes the techniques and rationale involved in this synthesis-order choice.

Controlflow nodes come from begin-end blocks (which includes the set around the entire program), if-then-else statements, and loop statements of the original description program. Function nodes describe operations such as addition and assignment of the original program (see Figure 5.1). Each controlflow node (such as an if-then-else statement) points with its control-dependency arcs to node sets (such as for the if, then, and else parts of the statement). These sets consist of controlflow nodes and function nodes, or both. This hierarchy of controlflow nodes describes the potential order of execution.

The instruction-set fragment-synthesis order is based primarily on controlflow order and secondly on source-code order. The resulting order takes advantage of two factors implicit in the nature of the instruction-set description: First, the design of the instruction set is based on decoding particular bitfields, which are expressed as decoding trees in the instruction-set description. These decoding trees usually clump together similar functionalities. Secondly, a good programmer often clumps together similar ideas in source code instead of writing spaghetti code. This also localizes

similar types of functions into groups, which can have advantageous effects in synthesis.

This localization of function and the chosen synthesis order affects the inference engine in two ways. First, consecutive synthesis attempts are made on the parts of the instruction set that can use the hardware in similar, but slightly different, ways. This exhaustive testing of particular synthesis decisions quickly verifies the synthesis choice or prompts action to correct it. A weighted analysis of the number of times the choice is used may trigger a synthesis choice change (described in a later section). The second effect of synthesis order is that repeatedly using a hardware resource within a section of instruction-set description code quickly builds up resistance to change. For these reasons, a top-down, left-to-right traversal of the entire instruction-set control-dependency subgraph is a satisfactory synthesis order.

5.3.3 The dataflow-synthesis loop

The following outline describes the main actions of the synthesizer. The inference engine maintains four lists (organized as stacks); these stacks deal with the complexity of the controlflow structure just described. These four lists are: 1) A list of source-code statement nodes to synthesize (a controlflow statement may contain code of arbitrarily complexity, such as a nested "then" block), 2) A list of the source-code fragments that each statement represents (such as the contents of a "then" block), 3) A list of source-code operations to

synthesize for each fragment, and 4) A list of microoperation nodes for each operation.

The following outline describes the eight steps that control parts the inference engine. Step 1 breaks each source-code statement node (which may be as complex as a case statement) into fragments and places the fragments on the source-code fragment list. Step 2 breaks each node from the source-code fragment list into individual operation nodes and places them on the operation node list. Step 3 performs compilation based on previous synthesis results. Steps 4 to 8 perform transformation experiments to determine a way for the hardware to perform the necessary calculations.

0) Make list of instruction-set nodes to be synthesized.

1) Pick first unsynthesized source-code statement node remaining in list 1. If list is empty, then stop. Otherwise break up node into fragments and place them on the source-code fragment list.

2) Take first remaining member of source-code fragment list. If list is empty, then go to step 1. Otherwise, expand it into a list of operations and place each element of the list on the source-code operation list.

3) If all members of the source-code operation list are in catalog, then compile them in each way described in catalog. Choose the best translation for each set of compiled nodes based on context, then go to step 2.

4) If operation-node list is empty, then go to step 2. Otherwise, take innermost remaining node in operation node list and label it as a microoperation.

5) Try to create mapping between microoperation nodes and hardware nodes, based only on operation name. If successful, then go to step 8.

6) Determine which node prevents straightforward mapping; call this the problem node.

7) Undo any previous transformations to a problem node. Try to modify the problem node by using a transformation knowledge base that has not yet been tried yet on this problem, then go to step 3.

8) Try to create mapping between microoperation nodes and hardware nodes, based on both the node operation names and variables. If successful, then go to step 4. Otherwise go to step 6.

5.4 Inference Engine Synthesis Heuristics

This section begins with a brief discussion of two inference engine implementation approaches. The process of matching the instruction-set description to the hardware description occurs in three parts: matching the nodes (such as the "add" instruction being mapped for execution by the ALU), matching the arcs of a node (such as which of the two operands of an addition operation being mapped to what side of the ALU), and mapping the variables

(such as the program counter being mapped to a specific internal register). In order, node mapping, then arc mapping, and later the variable mapping are described here. When each of the instruction-set nodes is mapped completely to a hardware node, the code-generation phase of microcode synthesis is complete. Both of the hardware and instruction-set descriptions are stored within Prolog's global database (review Figure 5.1). The description of two different Prolog implementation techniques follow; each can accomplish this pattern-matching between two nodes.

One implementation technique would be an interpreter that reads two specified nodes from the database and executes a program of data-access operations. This would require explicitly checking if the two node sets correspond. In this case, the pattern-matching operations would be explicitly described by the author of the data-access program. In a second technique, a skeleton of a node (called a template and represented as a Prolog clause) could unify (the Prolog term for a combination of database search, pattern-matching, and variable assignment) with clauses from the database. If the template and database entry match closely enough, then further heuristics could process any small mismatched details. In this case, most of the pattern-matching operations are implicitly embedded within the Prolog implementation and the source code of the inference engine.

The synthesizer uses the second approach. This approach invokes a search of clauses in the database by a conventionally implemented Prolog interpreter. Heuristic processing of any mismatched detail is also specified by links to the template, which describe alternative configurations of hardware that perform the desired semantics. This design was described from the Prolog expert system implementer's point of view in [POE84]. The techniques used to guide this Prolog-based search heuristically is the topic of the rest of this section.

5.4.1 Node-structure mapping

Most of this section describes processing done to bias heuristic-based synthesis search for increased efficiency. Data structures called templates and arc lists express the bias used during search. The end of this section describes how the nodes using this bias are chosen at search time.

Three constraints affect search order when matching instruction-set description nodes to hardware nodes. First, the nodes of the instruction-set description fragment are ordered by function type. Second, the types of arcs in a node affect search order (as discussed in the section on arc mapping). Third, to base hardware-node search on past success, previously used hardware nodes are considered for reuse before considering unused nodes.

Arc types most strongly determines search order. Given equal arc types, node-function type then is considered; given equal node-function type,

previous use then is considered. After processing each arc of the instruction-set fragment (described in a later section), the techniques described in this section search for a hardware node if a corresponding hardware node is not associated with the arc.

5.4.1.1 Node search constraints. This section describes the way synthesis constraints bias the search for graph nodes. The synthesizer operates on one source-code fragment (such as shown in Figure 5.1) of the instruction-set description at a time. The Prolog clause name "instnode" represents the origin of each instruction-set-description fragment node. UMS uses this list of code-fragment nodes to create a second list, with a possibly different node order. This second list begins with the nodes most likely to differentiate the fragment from other fragments; the more common node types are placed toward the end of the list. The arithmetic or logic-function nodes (such as ADD and SUB, which represent addition and subtraction) are first on this list, data-access nodes (such as WORD and ARRAYSTORE for reading and writing array elements) are in the middle, and the most common (such as ASSIGN for assignment) types of nodes are last.

The inference engine uses the order of the second list when it searches the global database for hardware nodes to match with the instruction-set nodes. This heuristic prunes away as much of the search tree as early as possible by initially basing the search on any unusual operations within the

instruction-set-description code fragment. In practice, this eliminates searching exhaustively through assignment nodes (which are the most common) when a rare function node elsewhere in the fragment cannot match. The ordering is implicitly used, however, because the template (described below) and the arc list (described later) are based on the second list. This is another example of heuristic-based search techniques used in the inference engine.

The synthesizer follows the order specified by the second node list to create a data structure called a template, which superficially resembles a list of nodes (see Figure 5.2) from the fragment of the instruction-set description. The clauses within this structure bear a different clause name, "hwnode", which corresponds to the hardware-description clauses (records) of the global database they will be matched with. Each element of the template possesses the same major structural details as its corresponding instruction-set node. In particular, each has the same operation name, the same number of input and output sublists, and a logical variable replacement for the node number. Remaining details are left unspecified as anonymous logical variables. To constrain further search, later activity may specify some of these details. As hardware nodes are chosen, they are unified with their corresponding template element, and the variables of the template are filled with the corresponding values. Prolog's logical variables are very useful in implementing this heuristic.

Prolog Template:

```
template([
      hwnode('CONSTANT',
             [],
             [[_, _, _]],
             _),


      hwnode('ADD',
             [
              [_, _, _],
              [_, _, _],
              [_, _, _]
             ],
             [[_, _, _]],
             _),


      hwnode('ASSIGN',
             [
              [_, _, _],
              [_, _, _]
             ],
             [[_, _, _]],
             _)
]).
```

Node Structure:

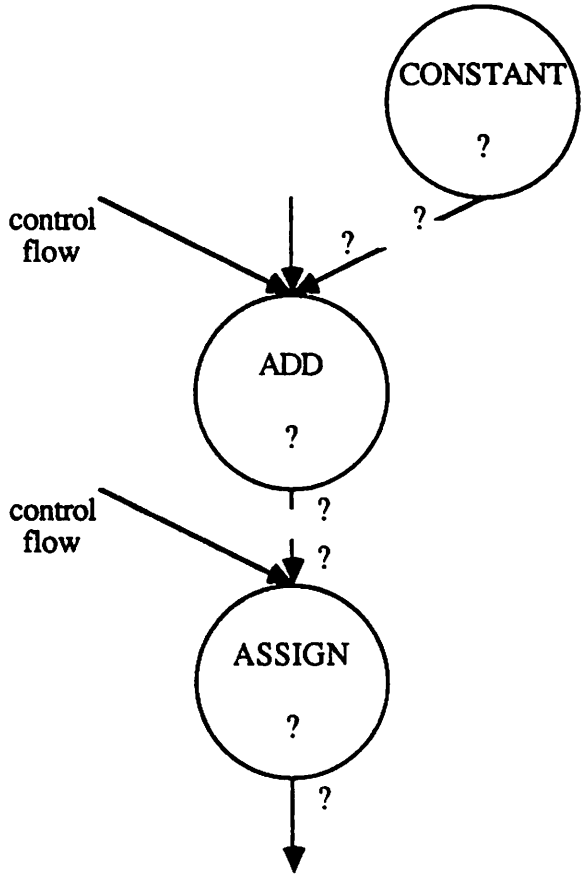


Figure 5.2: A node template for a database search. See Figure 5.1 for details of the graph structure.

5.4.1.2 Delaying hardware node choice. Each arc of an instruction-set node is processed to determine how well it corresponds to a hardware-node arc. The choice of which hardware and instruction-set nodes match together is delayed as long as possible to allow accumulating a list of constraints that must be simultaneously satisfied. This list constrains the number of candidate hardware nodes, which in turn makes the node search more selective and therefore more efficient. Most often, these constraints are

expressed as constants within the template or as shared Prolog variables unified with the template. These shared variables require that parts of the template posses exactly the same values. The node search is triggered during the matching activity of the node arcs; when processing an arc of an unmatched node a corresponding node is obtained. Thus, hardware node choice is delayed until the node's features actually need to be inspected.

5.4.1.3 Using synthesis experience to choose hardware nodes. This section describes the third and last technique to bias node choice. To maximize synthesis efficiency, it is important to apply any experience gained from previous synthesis activity. The synthesizer contains a list, called the catalog, of node numbers and node function types. This catalog describes hardware resources (such as a node from the ALU) previously used in synthesis. When a hardware node is needed to synthesize a new code fragment, the nodes of the appropriate function type from the catalog are tried first. The catalog is ordered as a stack to reflect the assumption that the hardware resources needed for the current synthesis may be most similar to the synthesis just completed. An unimplemented part of the synthesizer would increase the level of detail in the catalog by using the RUT data structure. Either method greatly improves synthesis efficiency by using the results of previous activity.

5.4.1.4 Triggering of node search. When processing begins on each arc (except controlflow arcs, whose dependencies do not require immediate processing) and a hardware node is not yet associated with that arc, one is fetched. When a hardware node is fetched, the template element of the corresponding instruction-set node is used during unification. Accumulated constraints are expressed as partially specified portions of the template. Initial unification attempts are made with hardware nodes with the correct function name from the catalog. If all of these fail, a sequential search is performed on the remaining nodes in the global database that are not in the catalog. This search succeeds only when the node matches the template's predetermined function name, and each input and output sublist contains the correct number of arguments with any accumulated constraints. The node type is the first argument of a node clause, which acts efficiently as a coarse type of search filter. The node search filter is implemented implicitly by the unification feature of Prolog's run-time system.

During the node-and-arc-matching process, any nodes and arcs that may prevent a match are noted, and a list of attempted matches is maintained. In the case of a match failure, the unsuccessful match that succeeded to the largest degree is considered further. A list of program transformations (described later) is tried on the problem node and possibly its neighbors. Because the problem node may have been transformed into a different function

type after the transformation, the node-choice process repeats on the new fragment during the remainder of the synthesis loop.

The three techniques to bias node search (ordering by node-function type, delayed node choice, and reuse of previously chosen nodes) determine in an efficient and flexible way which parts of the microengine should perform the semantics of a microprogram fragment.

5.4.2 Arc Mapping

The search for hardware nodes in the global database (discussed in the previous section) is triggered during the process of matching instruction-set arcs to hardware arcs. In other words, hardware nodes are fetched only when needed by the arc-matching portion of the inference engine. This section describes arc processing in detail.

A list of arcs is created at the same time the template is created (discussed earlier). Each member of this arc list has three components: the arc of the instruction-set node, a list of logical variables matching the corresponding "hwnode" template arc, and a list of bookkeeping information for use later in the synthesis process. When a hardware node is fetched from the global database, the logical variables of the corresponding arcs of the arc list become copies (by Prolog's unification feature) of the arcs of the hardware

node. In fact, the presence of variables instead of constants in this part of an arc-list element triggers the fetch of the hardware node before processing of the arc begins.

After introducing the general arc-mapping strategy, processing of five types of arcs will be discussed. The last part of this section describes swapping input arcs for functions with commutative semantics.

5.4.2.1 Arc-mapping strategy. The members of the arc list are processed in their list order, using recursively written Prolog code (described in detail later). When an arc is successfully compared to, or matched with, an arc of the hardware, the arc is said to be validated. If an arc cannot be successfully matched, then the arc is flagged as having temporarily failed, and processing may continue (described later) or backtracking occurs.

The inference engine's backtracking strategy is based on conventional Prolog top-down, left-to-right search. Backtracking causes zero or more of the previously successful arcs to be processed in alternative ways, using different Prolog code. The alternative code makes different decisions, which might allow validating a larger number of arcs. As a result of this search style, a previously successful arc is then automatically reconsidered from the new context after backtracking.

The arc list has a novel structure, so that Prolog's default search strategy implements the correct synthesis-search heuristics of the problem domain (also

see [POE84]). Before the final arc list is made, arcs are temporarily ordered for heuristic search by type (see Figure 5.3) by copying them into other lists called bins. The separate bins (or lists) are later combined into a final arc list that represents the search itinerary for all the arcs of the instruction-set fragment. Part of the bookkeeping information within each arc records in which bin the copy of the arc has been placed. Note that although arcs within a bin still correspond to the original node-ordering heuristic, arc type represents the strongest factor in arc-processing order.

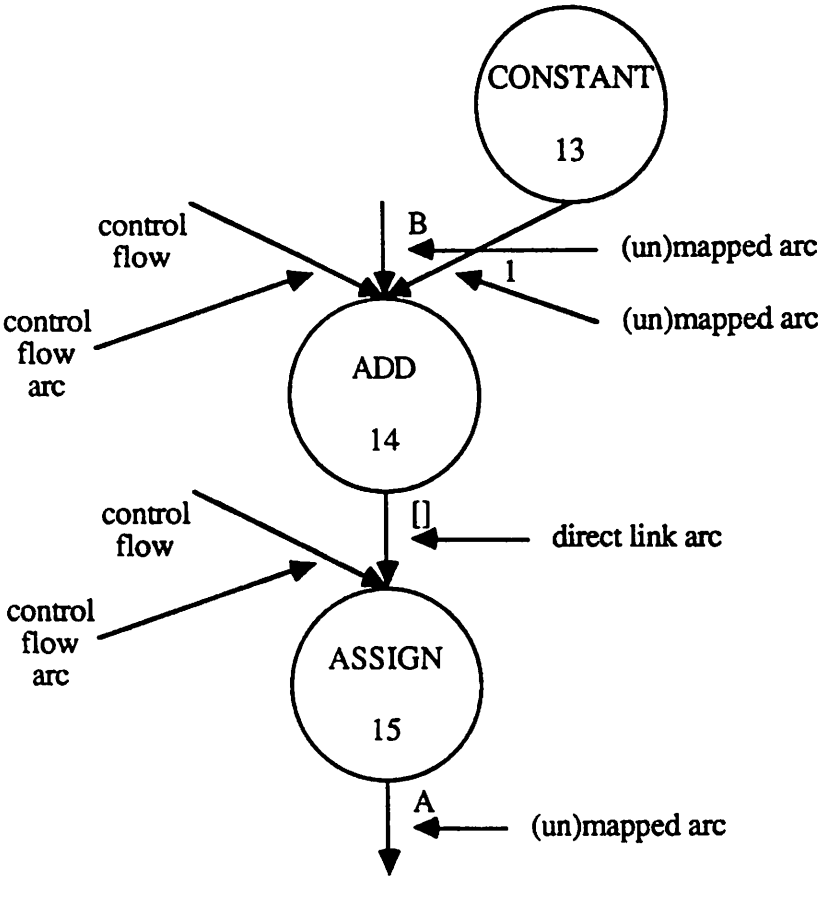


Figure 5.3: Different arc types. Compare to the template of Figure 5.1.

The arcs within each bin require different search heuristics for validation. The arc order in the arc list also reflects an estimation of increasingly expensive search-processing cost. During arc-list processing, a search-success metric is calculated before an attempt to validate each arc is made. The metric is based on the total number of arcs within each bin type, the total number of arcs, and the individual and collective success rates of these categories. If at any time the computational effort is unsuccessful enough relative to the amount of effort expended, that line of reasoning (and that part of the search space) is abandoned which triggers Prolog backtracking. An increasingly higher success threshold must be reached before processing the next arc. The success metric and the order of the arc list make it increasingly difficult to use the more expensive search strategies.

Those arcs that represent controlflow input are placed into the first bin. These arcs require a trivial amount of computational effort to validate, as it is necessary only to note the control dependency, which is often merely the value of a microword field. Direct-link arcs, which represent the connection between two nodes of the same code fragment, go into the second bin. A copy of the same direct-link arc goes into the sixth bin. Arcs from both of these bins are processed with techniques discussed in the section on topological mapping. The third bin contains arcs of variables previously mapped to hardware registers, with a copy going to the fifth bin. In the fourth bin are those arcs that

the synthesizer had not mapped to hardware, with a copy of those arcs going into the fifth bin. Note that if the synthesizer would process all the arcs in arc list, then the arcs would be analyzed in the order shown in Figure 5.4. This is one example in which Prolog's built-in search strategy does not preclude heuristic search.

Instruction-Set Nodes:	Arc-Processing Order:

`instnode('CONSTANT',`
`[],`
`[[1,[],[14]],`
	(7) (11)

`13).`
`instnode('ADD',`
`[`	

`[_,[['CHOICE',_,'SUBRDEF']],[12]],`	(1)
`['B',[],[6,11]],`	(5) (9)
`[1,[],[13]]]`	(6) (10)
`],` `[[[],[],[15]]],`	(3) (13)

`14).`
`instnode('ASSIGN',`
`[`	

`[_,[['CHOICE',_,'SUBRDEF']],[12]],`	(2)
`[[],[['FunctionName',_,'ADD']],[14]]`	(4) (14)
`],` `[['A',[['WIDTH',_,[0,3]],[23]]],`	(8) (12)
`15).`	

Figure 5.4: Potential order of arc processing. The numbers on the right indicate the order in which the arcs from the nodes on the left-hand side are processed by the inference engine.

The processing of arcs with variables is discussed in the section on variable mapping. When arcs from the third and fourth bins (previously mapped and unmapped variables, respectively) fail validation, copies of the arcs may be processed as members of the fifth bin. These expensive processing heuristics are described in the section on variable remapping. The sixth bin contains copies of the direct links. When the simpler processing of the second bin fails but the copy of the arc in the sixth bin is processed, then an expensive source-to-source transformation attempt is made to change the instruction-set description. These heuristics that process variable-use mismatches are described in the sections on the stretching a direct link within a fragment and temporary variable creation.

The bookkeeping information within each element of the arc list becomes especially important for arcs represented in more than one bin. Within each bookkeeping region of the duplicated arc pair is a common logical variable called a tag. The tag is set to a Prolog atom (or symbol) when processing of the first duplicate of the arc list is successful. If unsuccessful, then the Prolog logical variable remains unchanged. This indicates that a large amount of computational effort is needed to validate the arc. The inference engine processes an arc only when the tag is not an atom. Otherwise, the arc is ignored. Thus, the tag can act as a message that communicates between one arc and its duplicate farther down the arc list and inhibits further redundant

processing of the arc. The use of data dependencies (such as the tag) to guide multiple search strategies has an effect similar to the Prolog technique of intelligent backtracking [PER82], [PERP79], [PERP79A], [PERP79B].

5.4.2.2 Topological mapping. The direct-link arcs, which represent the connection between two nodes of the same code fragment (review Figure 5.4), are located within the second bin. Most direct-link arcs are trivial to map. To reduce the search space as much as possible, UMS tries to apply any insights derived from earlier arc-list processing. When a direct link is validated, it automatically specifies a complementary source or destination hardware node. The sequence of direct links between nodes often speeds up the search for hardware nodes that match the instruction-set fragment.

When a complex arithmetic or logical operation is to perform on a simple ALU, intermediate values of the calculation need to be stored temporarily and fetched later. The multiple operations then are performed consecutively during different ALU cycles. In such cases, a direct link with a "[]" variable name in the instruction-set arc would contain a state variable in its corresponding hardware arc. This would cause initial failure of the direct-link validation, and the arc's tag would be set to an atom. Later processing would potentially continue processing of this arc, as it is represented in the sixth bin. In the inference engine, separate Prolog clauses implement successful direct-link validation

and direct-link failure tagging for later processing and possible temporary-variable creation.

5.4.3.3 Bus mapping for variable values. Hardware allows data values to move along a set of wires from one functional unit to another. In a hardware-description language, this phenomenon is described by sequences of assignment statements, which involve many different variable names. On the other hand, instruction-set descriptions deal with variable values that may have a few or only one variable name. Bus mapping is the set of techniques that allows the inference engine to cope with these differences when it tries to associate (with variable-mapping techniques discussed later in this chapter) variables of the instruction-set description to variables of the hardware description. During the process of mapping a data-dependency arc, one or both of these processes are involved. This section discusses how bus mapping and variable mapping relate.

In a typical hardware description, data-transformation operations are relatively isolated from each other between a large amount of bus activity; the part of the code that describes addition within the ALU is an example. In contrast, operations in the instruction-set description are usually clustered together densely; for example, the hardware description fragments X:=PC; Y:=X+1; M[PC]:=Y might correspond to the instruction-set code fragment M[PC]:=PC+1. From a different point of view, the hardware description contains

a cluster of assignment statements to indicate how a value flows from one piece of the hardware through a sequence of buses and to the next hardware unit. This corresponding level of detail is absent from the instruction-set description. Note the special role that chains of assignment statements play here. Since the synthesizer depends on creating a mapping between code fragments in the hardware and instruction-set descriptions, differences in code type and density must be bridged.

The microengine datapaths are analyzed to understand the relationship between the hardware and instruction-set descriptions. The ISPS compiler provides the basis for this analysis from its control-and-data-dependency graph output. Due to the difference in code density described above, this analysis is performed only on the hardware description. A compiler postprocessor accomplishes the analysis. A data structure, called a bus, describes each step of the passage of a variable value through sequences of assignment and procedure-call statements. The bus analysis also includes the results of any variable name aliasing. Although the bus feature does not contribute any new information, it becomes quite important for efficient synthesis because the analysis is done once at compile time instead of repeatedly at synthesis time. A bus data record is a five tuple consisting of two sets of variable names and node numbers, along with a bus number. The assignment of the value of X from node 14 to Y at node 23 as part of bus 5 would be represented as

"bus(X,14,Y,23,5)." These bus data records can be searched much more efficiently than the original hardware-node data structures.

The synthesizer often applies source-to-source code-transformation techniques (see [LOV77], [STAHKN76], [STAKN76]) to modify the instruction-set description to resemble more closely the hardware description. In many cases, however, taking liberties with the node-mapping mechanism can avoid this type of expensive code modification. Modifications to the node-mapping mechanism allow "stretching" of the direct links between nodes. This stretching takes into account intermediate variables in either the instruction-set or hardware descriptions, and is based on the bus analysis described above. Both the actual mapping of variables and the corresponding changes to the node-mapping mechanism are described in a later section. The activity leading up to testing when two variables can be mapped is described next.

5.4.2.4 Confirmation of existing variable map. Bus mapping is most commonly used as a substitute for actual variable mapping, in which the implied mapping requires a value to move over a hardware bus. Consider the situation in which an instruction-set variable has been previously mapped to a hardware variable, but the arc currently under consideration suggests a different variable mapping. The synthesizer could immediately reject the validation of the arc. Instead, extra effort is made by the synthesizer to

determine if this new variable mapping can, through a chain of assignment statements, either read or write (as appropriate) the variable described in the preexisting mapping. If this is possible, then the arc is considered validated. Since this process involves an arc with a data-dependency variable, it is called stretching a variable link. This stretching of a direct link is discussed in the next section.

The bus-mapping process is actually a test of connectivity over the hardware-description dataflow graph. For example, consider the instruction-set fragment X1:=X2+1, where X1 and X2 are dedicated registers of the instruction set which are already mapped to specific hardware registers. Let X1 be previously mapped to a hardware register called Y (see Figure 5.5). Suppose the only part of the hardware description that roughly matches X1:=X2+1 is the fragment ALUOUT:=APORT+BPORT. When processing the output arc of the assignment node, an attempt is made to map the output of the hardware ALU ALUOUT to X1 of the instruction set; this contradicts the previous mapping of Y to X1. Trivial bus-mapping failures such as this one trigger additional synthesis effort.

In this example, ALUOUT is simply a temporary variable that represents one part of the output bus of the ALU. The connectivity of the hardware-description dataflow graph is tested between this write to ALUOUT and to an input of an assignment to Y. The search of the graph determines that

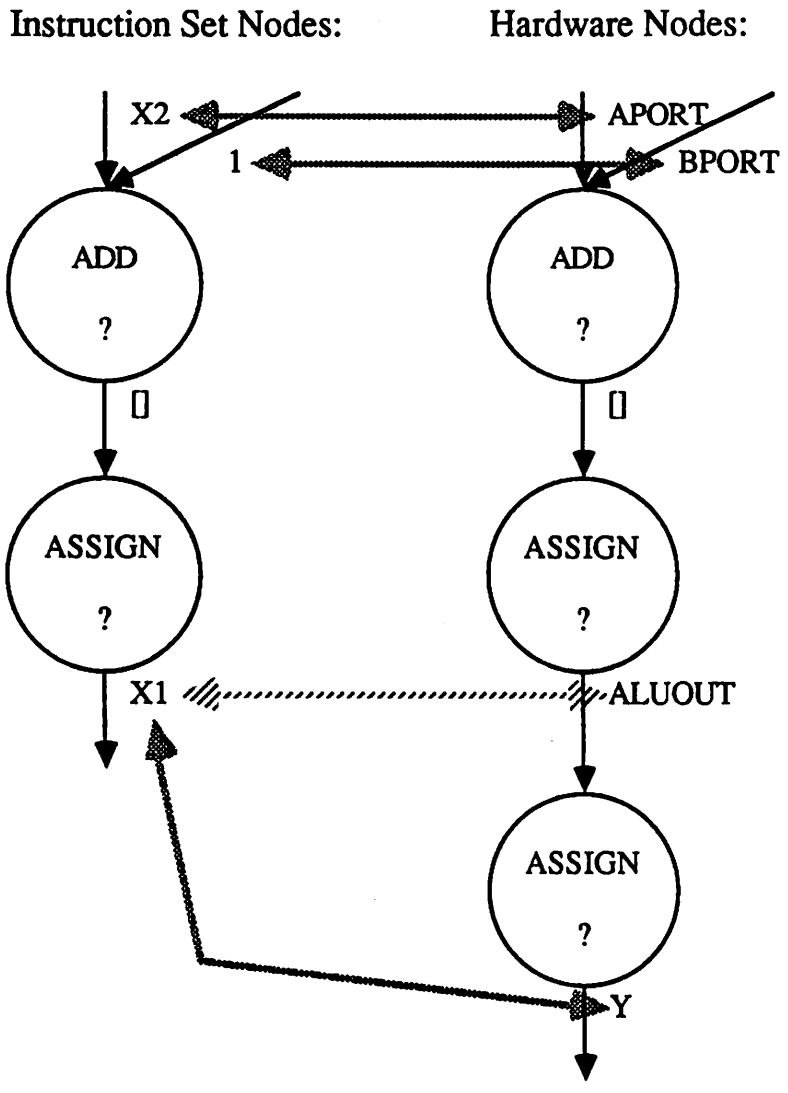


Figure 5.5: An example of mapping nodes with variables located on a bus. The instruction variable X1 is conceptually mapped to Y, although the hardware literally maps X1 to ALUOUT.

data can flow from ALUOUT to Y. This confirms the viability of the output arc of the instruction-set assignment statement, which maps ALUOUT to X1. In this case, X1 is not recorded as directly mapped to ALUOUT; instead, the synthesis record describes use of the previous mapping between X1 and Y.

When processing links such as this one, the hardware variable most distant along the bus is associated with the instruction-set variable in the resulting synthesis record. Distance can be calculated in these cases because one graph fragment is always a subset of the other; otherwise, the connectivity test would have failed. In this example, since the assignment to Y is farther from the addition node than the assignment to ALUOUT, Y is the variable recorded as mapped to X1. The inference engine handles four stretched variable-link permutations: these are the values located upstream or downstream (within the dataflow, corresponding to variables X1 or X2 of Figure 5.5) of the new or previous hardware variables farthest from the current arc (corresponding to the Y to X1 potential map or a map from ALUOUT to some unknown variable of Figure 5.5). Although a preexisting variable mapping is not maintained, the inference engine uses the same techniques to route bitfield constants (such as integers) to the ALU.

5.4.2.5 Stretching a direct link within a fragment. This section describes processing arcs from the sixth bin, which involves stretching direct links to achieve a match between node types within an instruction-set fragment. This is different than the previous section, where an arc was stretched to support an existing variable mapping. For example, in the synthesis of the PDP-8 microcode described of Chapter VII, the instruction-set fragment "AC :=

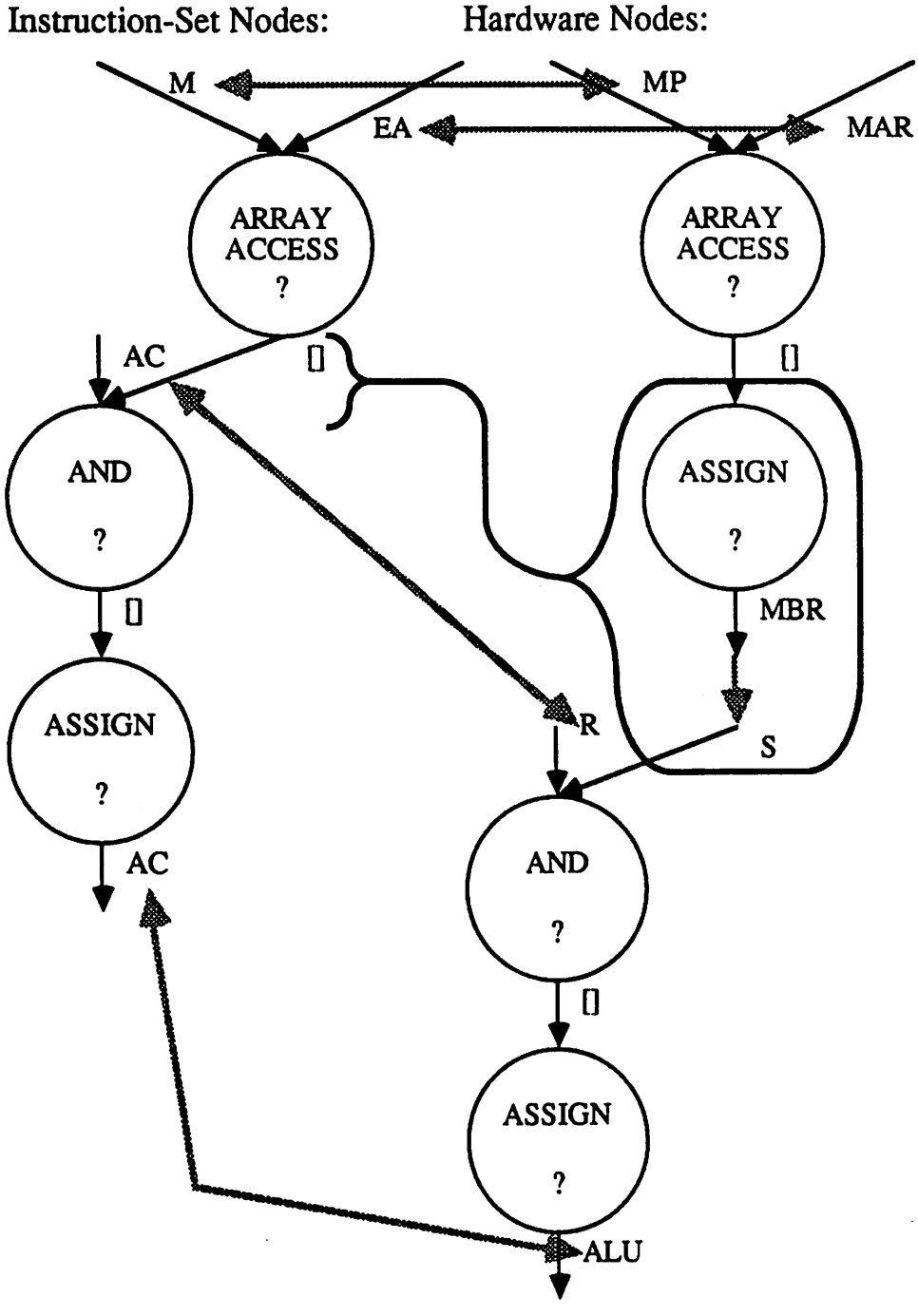


Figure 5.6: An example of stretching a direct link to map node types.
A value flows through an extra assignment node to a variable with a different name. Other heuristics process the duplicate use of the variable "AC" in the instruction-set description.

AC and M[EA]" is synthesized into microcode. This fragment has "ASSIGNMENT", "AND", and "ARRAY ACCESS" nodes (see Figure 5.6). A direct link connects the output of the "ARRAY ACCESS" node to the second input of the "AND" node. In the example of Chapter VII, only one part of the hardware can perform the "ARRAY ACCESS" and only one part of the hardware can perform the "AND" operation. Thus, only one hardware node matches each AND and ARRAY ACCESS node of the instruction-set fragment. These types of direct-link constraints strongly limit the number of code generation alternatives that require consideration and increase synthesis efficiency.

To understand stretching the direct link, consider the arc between the "AND" and "ARRAY ACCESS" nodes of the instruction-set description (see Figure 5.6). The corresponding code fragments of the hardware description are "MBR := MP[MAR]" and "alu := R and S". Ignoring the differences in the variable names (mapping instruction-set names to hardware names is described later in this chapter), these two fragments are slightly different than the instruction-set fragment we are trying to match. Two arcs specify each connection between two nodes, one in the source node, the other in the destination node. In the hardware fragment "MBR := MP[MAR]", the output arc of the "ARRAY ACCESS" node is a direct link and corresponds to the arc of the instruction-set fragment. Instead of finding a symmetrical direct-link arc within the hardware node, an arc

with a variable is found. Specifically, the arc is the second input of the "AND" node of the hardware fragment "alu := R and S". This is different than the corresponding second input arc of the AND node within the instruction-set description "AC := AC and M[EA]" (see Figure 5.6). Additional processing is required to confirm a viable correspondence between the arc descriptions of the instruction-set description and those of the hardware.

The "stretching" of the direct link is performed as follows. In this example (shown in Figure 5.6), the mismatched input-arc description of the hardware is inspected to determine the variable name associated with it. The variable name "S" is located within the expression "alu := R and S". The correctly matching output-arc description is inspected to see if it connects to an assignment statement and which variable name it is associated with. This allows extracting the variable name "MBR" from the expression "MBR := MP[MAR]". The hardware-bus descriptions are searched to determine if a bus exists between "MBR" and "S" data dependencies. Fortunately in this example, such a bus does exist, and the direct link has been functionally "stretched" to run along it. The rest of the arcs are processed conventionally in this example, and the fragment synthesis completes easily.

5.4.2.6 Creation of temporary variables. A direct connection between two operation nodes cannot always be stretched to fit within a single use of a hardware datapath, as was described in the previous section. Multiple

passes through the hardware datapath are required sometimes to perform the semantics of an instruction-set code fragment with two or more operations. This section describes another type of processing for arcs from the sixth bin. This processing changes the instruction-set description by creating temporary variables. These new variables temporarily store calculation results between multiple passes through the hardware datapath.

Consider the situation in which the instruction-set description adds three numbers, such as in the statement A:=B+(C+D), but the hardware can add together only two numbers at once. Figure 5.7 shows the only possible relationships between the instruction-set nodes and those of the hardware. Since the ALU has only one addition node, the same ALU node would match both addition nodes of the instruction-set description. Except for the output arc of the C+D addition node and the second input arc of the B+? node, the instruction-set arcs collectively match the nodes of the hardware (see Figure 5.7). During processing of the arc list, processing on all arcs except these two will succeed initially. Specifically, the processing heuristics of the second bin fail for the direct-link arcs in question. The direct-link heuristics of the sixth bin solve the mismatch with techniques similar to the stretching of a direct link described above.

Instruction-Set Nodes: Hardware Nodes:

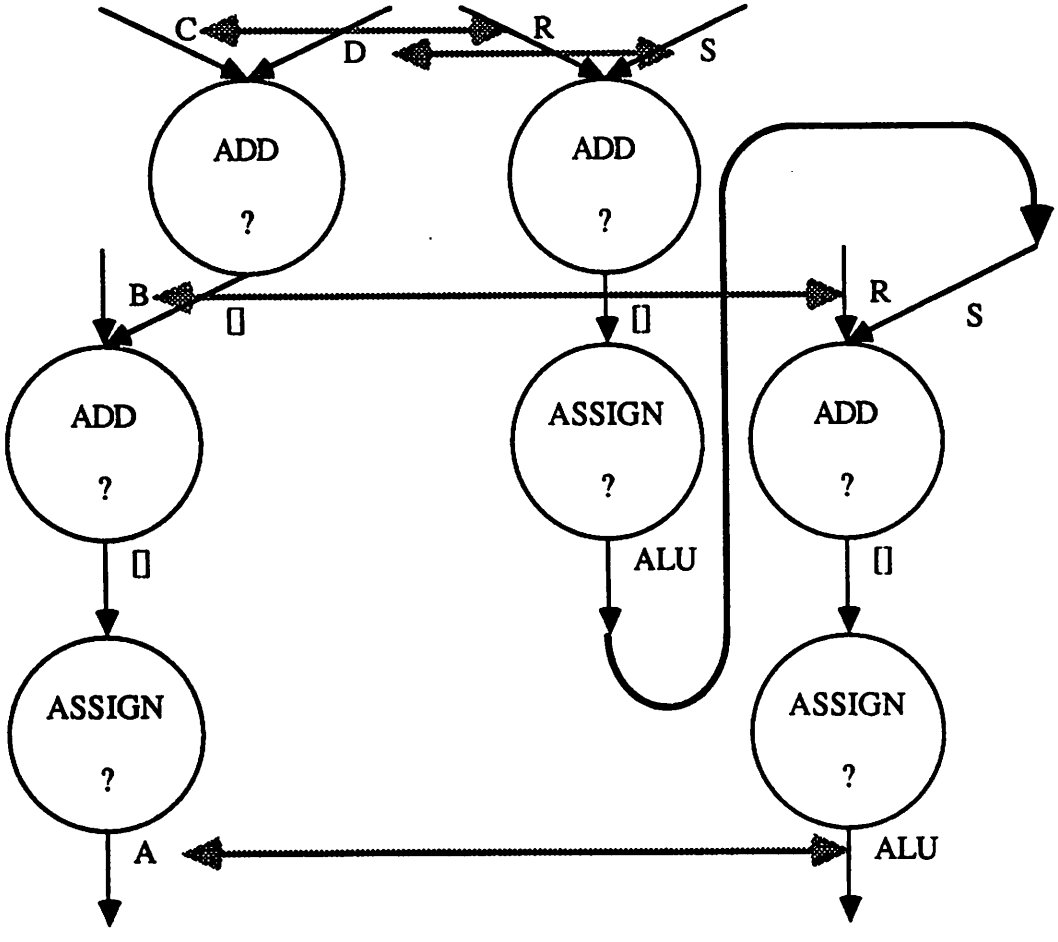


Figure 5.7: An example of an instruction-set fragment needing multiple passes through a datapath.

The arc-processing failure from the second bin (the second input arc of the addition node B+? and the output node of C+D) triggers another type of search, which is specified by a different Prolog clause of the inference engine. UMS searches the buses for a common temporary register by inspecting all bus inputs to the B+? node and all output buses of the C+D node. A variable name

contained in both datapaths is found. The variable name is a temporary register that stores the value of the calculation C+D at the end of one microinstruction and transmits the value as the input to the B+? node during a different microinstruction. If this register is part of a register bank, then one element of that array is assigned for this temporary use by register-allocation code. Once this special search is complete, the original arc is considered to have been mapped.

To avoid performing this much effort every time the opportunity arises, this expensive type of arc processing is delayed until late in the mapping process. The initial attempt at processing these direct-link arcs from bin 2 fail, sets the value of the tags to an atom, and signals more expensive processing of the same arcs as a member of bin 6.

This cost-conscious scheduling of synthesis activity is performed by: 1) Initially placing a copy of the direct-link arc into a bin that is one of the last to be processed, and 2) Requiring that the results of previous arc processing reach a threshold of success before an arc from this bin is actually processed. Factors used to compute the threshold are the total arc-success rate and the amount of computational effort needed for additional successful attempts. If most previous efforts have failed, the synthesizer will not invest the effort to map a current difficult-to-process arc. If a large number of arcs map successfully (some of them with a great deal of effort), then the synthesizer continues to work on them.

This conserves the computational effort the synthesizer has already invested. The synthesizer, however, continually reevaluates its computational investment in a search path.

The five arc-mapping techniques just discussed collectively process every type of arc used in UMS. The next section describes were two arcs may be swapped with each other.

5.4.2.7 Commutativity. The design of hardware microarchitectures is often highly irregular to allow as much flexibility as possible in hardware semantics. Given a limited number of microword control-field values, a microword field value is likely to be used for a unique operation type instead of a variation on an existing operand type. For example, it may be possible to route two operands (such as A+B) to the ALU for addition but a microword field value may not exist for an equivalent function with the operands reversed (such as B+A). Hardware designs often make this tradeoff of functionality for irregularity. From a microprogrammer's point of view, however, the irregular applicability of commutativly rearranging microoperation operands increases the challenge of microprogramming.

For those operations that allow commutativity, the microprogram must take into account the idiosyncrasies of the hardware to perform the desired semantics. Specifically, the inference engine must be able to reverse

dynamically the operand orientation for commutative operations. This section describes techniques to deal with these irregularities.

The initial design of the UMS inference engine assumed that commutativity of operands was a relatively rare occurrence. This design decision seemed justified, particularly because the initial synthesis was to be performed on the relatively regular AM2901 bitslice ALU architecture. Human microprogrammers learn to perform commutativity so readily that it requires little intellectual effort. As a result, changing the order of operands for selected functions (such as addition, OR, AND, etc., but not subtraction) initially was designed to be an expensive transformation of the instruction-set description controlled by a program-equivalence rule (described later in this chapter). However, it soon became clear how often this type of manipulation was needed to produce results.

To implement automatic commutativity of operands, both operand orientations are put in the arc list. When a template for these types of nodes is made, special bookkeeping information is constructed within the arc list. This is based on an operation-type table that describes which are commutative (addition, and, inclusive or, exclusive or, etc.). When the first orientation is chosen during processing of the arc list, logical variables in common with the arcs of the two operand orientations are set to values that trigger passing over the alternate orientation. If search using the first orientation does not succeed,

then logical variables are instantiated to prompt processing of the other orientation and resumption of the search. If the second orientation also does not succeed, then the search up to that point has failed. If orientation choices are possible for multiple operand pairs of an instruction-set fragment, then the inference engine cycles through all the permutations. Commutativity heuristics enlarges the coverage of the synthesis search space (see Figure 5.8) at the cost of inference engine complexity.

The five arc-mapping techniques and the swapping of commutative arcs discussed in this chapter are all integrated parts of the UMS inference engine.

Instruction-Set Nodes:	Arc-Processing Order:

`instnode('CONSTANT',`
`[],`
`[[1,[],[14]],`
	(9) (15)

`instnode('ADD',`
`[`
`[_,[['CHOICE',_,'SUBRDEF']],[12]],`
`['B',[],[6,11]],`
`[1,[],[13]]]`
`],`
	(1)
	(5) (8) (11) (14)
	(6) (7) (12) (13)

`[[[],[],[15]]],` `14).`	(3) (13)

`instnode('ASSIGN',`
`[`
`[_,[['CHOICE',_,'SUBRDEF']],[12]],`
`[[],[['FunctionName',_,'ADD']],[14]]`
`],`
	(2)
	(4) (17)

`[['A',[['WIDTH',_,[0,3]],[23]]],` `15).`	(10) (16)

Figure 5.8: Potential order of arc processing with commutativity of addition. The numbers on the right indicate the order the arcs from the nodes on the left-hand side are processed by the inference engine. Compare to Figure 5.4.

5.4.3 Variable Mapping

So far, the discussion of the inference engine has focused on two issues: matching instruction-set nodes to hardware nodes and matching instruction-set arcs to hardware arcs. There is a third and final issue that must be addressed: matching instruction-set variables to hardware variables. Characteristics such as the width of the bitfield (number of bits in the variable value) as well as whether the value is both read and written or only read must also match. Those temporary variables of the hardware that represent buses between hardware components are unable to store values (that is, be written into). Temporary hardware variables are not allowed to bind to instruction-set state variables. This section discusses these variable-mapping issues.

5.4.3.1 Initial variable mapping. In the arc list described earlier in this chapter, arcs with previously mapped variables were contained in the third bin, arcs with unmapped variables in the fourth bin, and difficult cases of both types in the fifth bin. This section discusses only those cases in which the mapping is easily accomplished from the third or fourth bins. The next section, on variable remapping, discusses processing challenging variable arcs from the fifth bin.

It is usually easy to match a previously mapped hardware variable for arcs from the third bin with one from the instruction set. A small database of previous matches is maintained by the synthesizer. The current map candidates are simply compared to data. If this comparison is unsuccessful, then additional processing is performed on this pair as an element of the fifth bin.

Much more effort is needed to map arcs of the fourth bin (i.e., those arcs with previously unmapped variables). Both the instruction-set and hardware description are written in the ISPS language. Many of the characteristics of each ISPS variable are specified in variable declarations within the ISPS source code. These declarations are translated into graph form along with the rest of the code. However, because different variables within multiple source-code modules may have the same name, the code (now in graph form) is searched along the controlflow paths to find the appropriate variable definition.

Once the variable definitions are found, the instruction-set and hardware variables are compared for compatibility. For the hardware description, controlflow and dataflow analysis determine whether a variable is a memory array (indices are used in references), an input port (only reads are found), or an output port (only writes are found). The analysis also determines whether a

variable should or should not be part of the instruction-set machine state (the value is written in just one place, and read directly afterward).

Compatibility checking begins by inspecting the bitfield width (number of bits in the variable) for equivalence (however, the bits may be numbered differently). If both variables are arrays, then the instruction-set array is checked to ensure there are at least as many cells in it as in the hardware (the cells may be numbered differently). When the graph forms of the ISPS descriptions were created each of the variable definitions was annotated (after dataflow analysis) to specify whether the variable was used in the source code only for reads or for both reads and writes. This read or read/write attribute is checked for compatibility between the instruction-set and hardware descriptions. Failure of any of these checks has the potential to trigger further and much more costly processing in the fifth bin. Finally, the hardware variable is checked to ensure that it is not part of a hardware bus. Later discussion describes how hardware variables are determined to belong to a hardware bus during synthesis. Thus, simple variable mapping is primarily a variable characteristic-lookup process. Subsequent sections describe variable mapping that is not this simple.

5.4.3.2 Variable remapping. Multiple uses of an instruction-set variable typically reaffirm a mapping to a hardware variable. However, an attempt is made sometimes to map an instruction-set variable to a hardware

variable other than the one it was mapped to previously. This is more likely to occur during the first few times the inference engine encounters a variable. One simple example of this situation is shown in Figure 5.5, where a previous map to a hardware variable does not work. Whereas this previous section described confirmation of such a mapping, this section describes the conditions and techniques that cause altering previous variable-mapping synthesis choices.

A variable map may need to change when a new instruction-set fragment uses a variable in a manner inconsistent with previous uses. There are two possible motivations for remapping a hardware variable to an instruction-set variable: 1) to converge the current and all previous uses of the instruction-set variable on a single hardware variable and 2) to diverge the mapping of hardware variables when the new mapping of a hardware variable interferes with previous uses. These cases are discussed here in turn.

On the rare occasions when a variable is remapped, some of the new variable uses do not cause problems while other uses cannot be successfully changed. In the former cases, one application of a transformation rule (this process is described in a later section) per variable-use problem is performed. The problem part of the instruction-set description is resynthesized, and another remapping attempt is made. This potentially recursive modification is triggered only when relatively little previous synthesis activity relies on the

remapped variable. Otherwise, it is assumed that the previous synthesis was correct and the current variable-mapping attempt should fail.

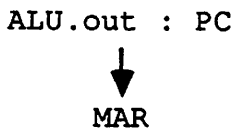
5.4.3.3 Converging multiple instruction-set variable uses. The convergence of multiple uses of an instruction-set variable is described first, using an example from the synthesis described in Chapter VII.

The first time the variable "PC" appears in the instruction-set description is in the expression "i=M[PC]". Here, the value of "PC" (Program Counter) is read. Initially, the variable "PC" of the instruction-set is mapped (see the first map of Figure 5.9) to the hardware variable "MAR" (the input of the memory-address register). Simple reasoning motivates this first mapping: When the instruction-set node is mapped to the hardware node, the arc of the instruction-set fragment containing the PC data dependency corresponds to the "MAR" dependency hardware arc. Specifically, the instruction-set description fragment "i=M[PC]" is mapped to the hardware-description fragment "MBR=MP[MAR]", where "MP" is the primary memory, "MAR" is the memory-address register, and "MBR" is the memory-buffer register. When no other constraints exist, the value to be read is assumed to be exactly where it is needed. When a later instruction-set fragment also must read the value of "PC" as the input to an "AND" operation, a conflict occurs because there is no way to read "MAR" in the hardware. This conflict triggers variable remapping to change the mapping of "PC" to a more appropriate part of the hardware.

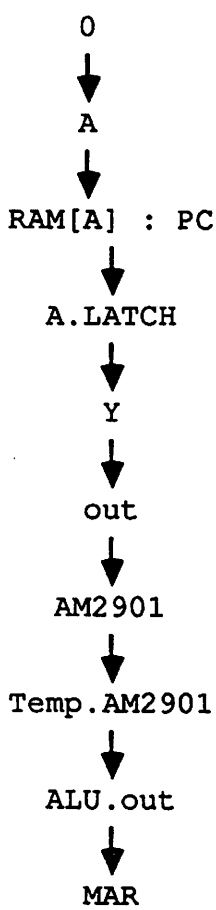
The mapping

```
MAR : PC
```

is changed and tested as:



and is changed again and retested to become:



to allow this data flow:

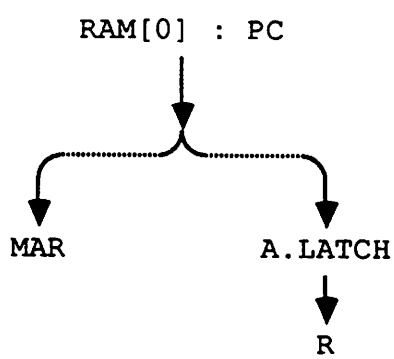


Figure 5.9: An example of variable remapping. Initially, the instruction-set variable "PC" is mapped to the hardware variable "MAR". Later synthesis constraints require modifying this previous result. The mapping of "PC" is moved "upstream" from "MAR" to "ALU.out". This mapping does not satisfy all constraints, so the mapping of "PC" is moved and tested again and again, until the successful binding of "PC" to "RAM[0]" is found (from Chapter VII).

Search attempts are made to find candidate hardware variables that can move the needed value along hardware buses. In this remapping example, the hardware data-dependency graph is searched for variables that write to "MAR" (see Figure 5.9); this occurs because the previously mapped instruction-set fragment "i=M[PC]" performed a read from "MAR". Such hardware variables can satisfy the "PC read" of the original hardware fragment "MBR=MP[MAR]", which was previously mapped to an instruction-set node. The search begins with hardware variables that are closest to the "read PC" hardware arc (i.e., those arcs that directly move a value into the previously mapped node). The search progresses to candidate variables connected by an increasingly larger number of arcs through "ASSIGN" nodes. These later candidate variables can perform the required movement of variable values along a hardware bus. Such search at this point checks only the read and write data-dependency constraints defined by the original node and arc mapping.

Once a candidate hardware variable that satisfies all previous data-dependency constraints has been found, it is tested for satisfaction of the new constraint (of the current instruction-set fragment). In this example, the first candidate variable is "ALU.out", which is read by the hardware variable "MAR" (see Figure 5.9). The additional constraint is if "PC", when mapped to "ALU.out", can be read by the new arc. In this particular case, assume it cannot be, and the search of the hardware graph continues. Eventually, the variable

"RAM", which satisfies both the previous dataflow constraints (the hardware mapped to "PC" can be read by "MAR" of the hardware) and the new requirements of the current instruction-set fragment (the hardware mapped to "PC" can be read by "R" of the hardware) is found.

Once the data-dependency constraints are satisfied, the hardware variable characteristics are checked to match those of the instruction-set variable. This process was described in the previous section on initial variable mapping. The data-dependency based search resumes if the hardware variables do not possess the proper characteristics, such as incorrect bit width or number of array elements.

This example of variable remapping has shown how to make possible two different reads of the hardware mapped to the instruction-set value PC. The inference engine has changed the part of the hardware that is considered to contain the value of the instruction-set variable PC. The move was "upstream" along a hardware bus to a point where the value could travel "downstream" to be read by the two different parts of the hardware. The inference engine performs very similar reasoning to satisfy the need of multiple writes to the same instruction-set variable.

Time is considered implicitly when the inference engine remaps a variable that is both read and written. For both reads and writes, two such nodes must be chosen: one "upstream" to represent the source of the variable

value read, one "downstream" to receive the writing of the variable value. Implicit in the hardware description is the assumption that during one microword-execution time cycle, the value written during the previous microword execution is the same as the one read during the subsequent cycle. See [POE81A] and Chapter IV for techniques that describe more complicated timing constraints in hardware state variables.

5.4.3.4 Diverging multiple variable use. While the previous section described changing the instruction-set to hardware variable mapping to allow consistent use of a variable, this section discusses diverging a variable mapping into two separate ones. This occurs when one part of the instruction-set description needs a hardware resource that has been mapped to another part of the instruction-set description for a specific, but different, use.

Figure 5.10 shows an example of how variable remapping diverges hardware resource. This example discusses only moving bindings of written variables down a bus. The inference engine applies similar logic to move read-variable bindings up a bus or to move variable bindings in both directions. Figure 5.10 shows that a state variable "A" of the instruction-set description has been mapped previously to the "F" output port of the ALU. The current instruction-set fragment also tries to use the hardware variable "F". This conflict prompts an attempt, through techniques similar to those discussed in the

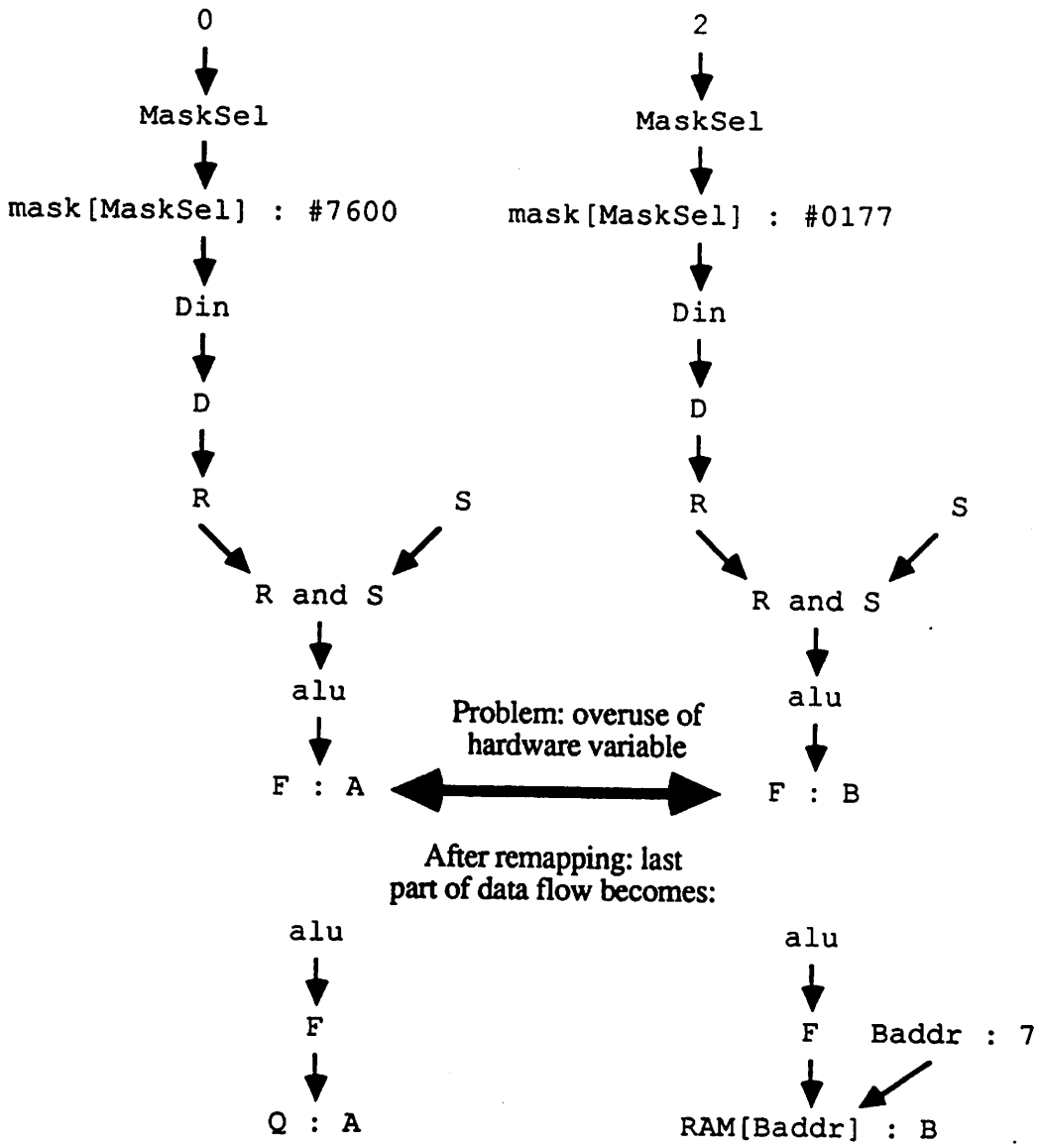


Figure 5.10: An example of variable remapping. Each variable of a fragment of the hardware dataflow path is shown and, if appropriate, a colon follows with any mapping of that variable to one of the instruction set. Temporary instruction variable "A" is mapped initially to hardware variable "F". Variable remapping is triggered when a different instruction-code variable "B" also tries to use hardware variable "F". The solution (used in the example of Chapter VII) is to remap the original use of "F" to "Q" and to map the new use of "F" to "RAM[7]".

previous section, to move the mapping of the instruction-set variable "A" farther down the datapath.

In this example, "A" is found to be a suitable binding for the hardware register "Q", shown in the bottom half of Figure 5.10. When a candidate location is found for the previously mapped instruction-set variable, it is tested for compatibility with all other uses of the variable. If these are successful, then new mapping is recorded. Now that all previous uses of the conflicting instruction-set variable have been adjusted, the inference engine resumes its attempt to find a proper binding for the current instruction-set variable. In this example "RAM[7]" of the hardware is a good mapping for "B" of the instruction-set description, as shown in the bottom right corner of Figure 5.10.

The synthesizer's strategy is to delay decision making as long as possible in order to discover as many constraints as possible. However, arbitrary decisions are sometimes necessary before any real progress can be made. Because of this philosophy, previous decisions sometimes must be reconsidered as new constraints are discovered. This section on variable mapping and remapping concludes the discussion of synthesis revision in the inference engine.

5.4.3.5 Hardware-bus analysis. Once the synthesizer completes a variable-remapping process, it retraces the datapath of the hardware. All intermediate variables (such as A.LATCH through ALU.out in Figure 5.9, or F in

Figure 5.10) along the hardware datapath are tagged as part of the hardware bus. Later synthesis activity does not allow the mapping of an instruction-set variable to a hardware-bus variable. This concludes the discussion of variable mapping techniques.

5.5 Semantics Transformations

Previous sections have discussed the inference engine processing when a match is to be found. However, a fragment of the instruction-set description cannot always match the hardware. The validation of the arc-list elements fails, even after the most exhaustive search allowed by the heuristics that control computation cost. This section describes how the synthesizer deals with synthesis failure by modifying the instruction-set description while preserving its meaning.

Arc-validation successes and failures are recorded during search. The record of the most successful match to the hardware (the set of nodes with the largest number of arc validations) is inspected. To determine which arc failed just after the maximum number of arc validations, the corresponding arc-list validation record is inspected. Due to the progression of arcs in the arc list, this is the easiest-to-match node that does not match. This arc is labeled the problem arc, and the node it originates from is labeled the problem node. The

subsequent section describes the techniques that use this information to perform the needed transformation.

After isolating the synthesis difficulties, the inference engine applies semantics-preserving transformations to that instruction-set description fragment. An example of a semantics-preserving transformation, or microprogramming trick, is the equivalence of adding a number to itself and shifting left by one bit.

A set of transformations rules such as these is kept within the synthesizer and sorted by level of generality. The trick just described can be specified as a pattern of only one addition node, so it is classified as very simple. The bitfield-extraction transformation described in Chapter I is much more complex due to the number of nodes required to specify it. Generality is defined by the number of nodes needed to specify the transformed pattern.

Transformation of the instruction-set description graph is considered computationally expensive and is used as little as possible. The UMS ISPS compiler of the synthesizer makes explicit many of the implicitly described details of the source code's intended behavior. When a transformation is made, much computational effort is used to maintain the consistency of these details. The cost of pattern-matching database entries (using the Prolog unification feature) with a node template is much less than the total transformation cost. Therefore, UMS considers first the most-specific

transformation rules rather than the more-general rules. Although this transformation-database index strategy requires more pattern-matching effort, it also prevents frequent and erroneous use the more general rules and is therefore less expensive overall.

The transformation application's pattern-matching process succeeds only when the previously identified problem arc or problem node is part of the match. The inference engine attempts to apply the transformation rules consecutively in the order just described until pattern-matching is successful. When a match is made, the inference engine again tries to synthesize the fragment. If synthesis is successful, the synthesizer proceeds to the next instruction-set description fragment. If unsuccessful, the synthesizer resets the instruction-set description to its state just prior to the transformation and proceeds by attempting to pattern-match the next transformation rule. When the synthesizer runs out of rules, it first begins to apply double, then triple, and then higher multiple applications of transformation rules, until synthesis succeeds. After more than a few unsuccessful rule applications, however, the synthesizer user is warned that the knowledge base does not allow successful synthesis.

The transformations just described are implemented as fundamental part of the UMS inference engine. The next section describes how the inference engine performs these transformations.

5.6 Design of a Production System

A production system is a programming technique commonly used in the artificial intelligence field (see [NIL80] or [HAY84] for an introduction, or [DAVK77], [HAYWL78], [MCDF78], [MCDNM78], [WATH78], [GEO79], [JEFF80], [BARR81], [COHF82], or [ROS83] for more discussion). A production system consists of sets of rules that collectively perform a task. Each rule in UMS describes a substitution of a small set of nodes and their arcs from the instruction-set description for another set of nodes and arcs. In effect, these nodes and arcs are transformed into different ones. This section describes the design of the production system that performs node transformation for UMS.

The design of this production system was greatly simplified by using Prolog as the implementation language. Prolog is based on unification, a type of pattern-matching. Unification determines which rule to apply. Unification is also used to construct replacement nodes and pass parameters from the original node set.

A production-system rule contains two main parts: the LHS (left-hand side) and the RHS (right-hand side). These correspond to the "before" and "after" patterns, respectively, of the transformation. In UMS, the LHS is a list that describes to which instruction-set node(s) the LHS should match. Figure 5.1 illustrates a set of nodes and Figure 5.11 shows the LHS and RHS of a rule.

Simplified Prolog Form of a Node-Transformation Rule:

```
/*                                                    */
/* Equivalence of Adding One and Incrementing         */
/*                                                    */
/* A + 1 ==> INCR(A)                                  */

rule('Equivalence of Adding One and Incrementing',
      [
       instnode('CONSTANT',
                [],
                [[1,[],[R2]]],
                R1),
       instnode('ADD',
                [
                 [A,[],R3],
                 [1,[],[R1]]
                ],
                [[[],[],[R4]]],
                R2)
      ],
      [
       instnode('INCR',
                [[A,[],R3]],
                [[[],[],[R4]]],
                R2)
      ],
      29
     ).
```

rule comments

symbolic rule name

rule LHS

rule RHS

rule number

Figure 5.11: Prolog form of a semantic node-transformation rule. See Figure 5.1 for the relationship between an ISPS source code, Prolog node notation, and an arc and node graph.

Each rule partially specifies a Prolog data structure. Partially specified means that a portion of the structure is specified by Prolog literals (for example "ADD" in the LHS of Figure 5.11); other parts are specified by Prolog variables (R1 in Figure 5.11). Note the use of the same variable name in more than one location, such as R1 in the both LHS nodes in Figure 5.11. This practice allows describing complex constraints, such as an arc, that must be satisfied across

more than one node. It also allows passing parameters between the LHS and RHS.

To understand a simple change of the instruction-set description, consider the rule in Figure 5.11. Here, the LHS represents the arc and node representation of the ISPS code that adds the number one to a variable. This code is to be replaced with code that increments the variable, as shown in the RHS. To transform a set of nodes, the rule (for example, see Figure 5.11) is decomposed by the production system into the rule name, the LHS list, the RHS list, and rule number. To help the UMS user understand the behavior of UMS, the symbolic rule name and rule number are printed at the computer console after rule use. Some sophisticated rules allow specifying members of the LHS or RHS at rule-application time.

As described earlier in this chapter, UMS invokes the program-transformation production system when a specific node of the instruction-set description, called the problem node, does not match a node of the hardware description closely enough. To determine if the problem node matches part of the LHS, the problem node number initially is unified with (or "assigned" to) the number of the first LHS node member. Then an attempt is made to unify that node with a potentially matching instruction-set node from the global database. If that unification fails, the problem node number is moved to the next node member of the LHS, and unification is attempted again. If the unification

attempt is unsuccessful, then subsequent nodes of the LHS are tried. When all LHS nodes have been tried unsuccessfully, the next rule is attempted. This LHS-matching process is designed to determine very efficiently whether a part of the LHS matches the problem node.

When one of the LHS nodes matches the problem node, each remaining node of the LHS is unified a with node from the global database. When nodes from the LHS are unified with instruction-set description nodes, variables within the LHS are given values. In the example of Figure 5.11, when the second LHS node is unified with a node in the database, the variable R1 is set to a value. When it is time to search for the first node, the fact that R1 has a specific value highly constrains the search space, thus making the search much more efficient. As each rule-LHS node is selected, its common LHS variables increasingly constrain search.

The variables in common between the LHS and RHS help define the nodes that will replace the LHS. In Figure 5.11, for example, consider what occurs when the second node of the LHS is unified with the database. The name of the input to the INCR node is determined by the variable "A" (Figure 5.11), which is shared between the LHS and RHS. The node number for this input is similarly specified. Usually this use of common variables requires few calculations to be performed for the transformation, and most of the work is done efficiently within the Prolog run-time system.

Rule parameters, which are not shown in Figure 5.11, may be filled with a fragment of Prolog code to provide a type procedural attachment. If present, the code is evaluated when the rule is applied. This optional code may include algorithmic descriptions of arbitrarily complex constraints or provide for construction of LHS fragments. Examples of transformations requiring special procedural attachment are calculations for variable bit-width constants or the successful construction of integer constants.

Once every node member of the LHS has matched a node from the global database, these nodes are replaced with nodes of the RHS. Both the LHS and RHS are listed in a record of the transformation. This record is used in the event that the global database must be restored to its previous state. Such restoration is an important part of the inference engine's search strategies, (discussed in an earlier section of this chapter) when the transformation does not promote successful synthesis.

As this section has shown, use of the Prolog language strongly aids the implementation of a graph-based production system.

CHAPTER VI

CONTROLFLOW SYNTHESIS AND MICROCODE COMPACTION

6.1 Introduction

The unimplemented controlflow-synthesis process described in this chapter determines a microprogram's microoperation order for the available hardware as constrained by control and data dependencies. (In contrast, the implemented dataflow synthesis described in Chapter V determines which individual microoperations process the microprogram's data.)

This chapter begins by showing why dataflow synthesis alone is not sufficient to create usable microprograms and the description code-context factors used to create appropriate controlflow. A simple step-by-step example of controlflow-synthesis processing follows. A second controlflow-synthesis example shows how more advanced techniques are used when semantics need to be moved between the controlflow and dataflow representations.

This chapter discusses execution-time and code-space optimization and their interaction, which are also controlflow-synthesis goals. UMS explicitly distinguishes between controlflow and dataflow within its analysis of instruction-set and hardware descriptions. The synthesis decisions in controlflow and dataflow are largely independent. The controlflow- and

dataflow-synthesis spaces can be searched consecutively (so that their variations are additive) instead of simultaneously (which results in a multiplicative number of variations) when the RUT is used to process possible interactions. This allows more efficient search since, in general, search effort is proportional to the number of variations.

An important factor in controlflow synthesis is microengine parallelism (the hardware's ability to perform multiple operations at once). The unimplemented RUT in UMS is intended to be the basic tool for exploiting potential parallelism and guiding controlflow-synthesis search. This chapter also compares the RUT of UMS to conventional microcode-compiler local and global microcode-compaction technology (alternatively called microoperation scheduling or simply "packing"). Global packing within UMS is based in part on two previously published papers ([POE80], [POE81A]) that describe the V-Compiler's global microcode packer implementation written in 35,000 lines of Bliss language code. Microprogram compaction is covered next in the chapter, first local then global. The last topic in the chapter is resource allocation, such as register allocation.

6.2 Controlflow Synthesis Requirements

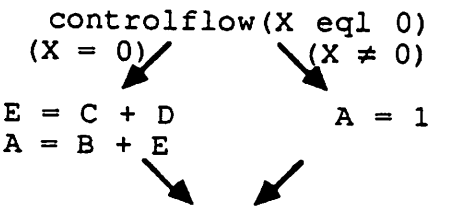
This section describes the additional synthesis effort required after completion of dataflow synthesis, the types of code-variation choices that must be made, and the information used in this processing. Controlflow synthesis would begin with two steps. First, controlflow compatible with the previous mapping of the instruction set to hardware dataflow is searched for in the hardware. Second, every detail and additional constraint of this hardware controlflow is processed.

Part 1 of Figure 6.1 shows the instruction-set description input source-code that UMS receives. The inference engine of UMS creates during dataflow synthesis a mapping between each part of the instruction-set description's data-dependency graph and the hardware-description graph. The mapping is many to one; many arithmetic operations in the instruction set can be mapped to the same ALU in the hardware description. Dataflow synthesis does not consider controlflow constraints of the microengine. Controlflow arcs, shown as arrows in parts 2 and 3 of Figure 6.1, remain to be created after dataflow synthesis.

Preventing irrelevant parts of the hardware from affecting the desired calculation is just as important as choosing the most appropriate hardware for calculations in the instruction-set description. Fortunately, the UMS design

```
DECODE X =>
      begin
0  :=    begin
         A = B + C + D
         end,
1  :=    begin
         A = 1
         end
      end
```

Part 1: Source Code



Part 2: Controlflow Graph

Address:	Branch Choice:	Branch Address:	Microoperation:	Controlflow:
PC + 0		goto PC + 1/PC + 2	controlflow(X)	
PC + 1	(X ≠ 0)	goto PC + 4	A = 1	
PC + 2	(X = 0)	goto PC + 3	E = C + D	
PC + 3		goto PC + 4	A = B + E	

Part 3: Branch If Zero Controlflow

PC + 0		goto PC + 1/PC + 3	controlflow(X)	
PC + 1	(X = 0)	goto PC + 2	E = C + D	
PC + 2		goto PC + 4	A = B + E	
PC + 3	(X ≠ 0)	goto PC + 4	A = 1	

Part 4: Branch if Not Zero Controflow

PC + 0		goto PC + 1/PC + 2	controlflow(X)	
PC + 1	(X = 0)	goto PC + N	E = C + D	
PC + 2	(X ≠ 0)	goto PC + 3	A = 1	
PC + 3				
PC + N		goto PC + 3	A = B + E	

Part 5: Remote Code Block

Figure 6.1: Alternative controlflow bindings. For clarification, some controlflow is shown with arrows. The graph shown in part 2 describes the controlflow of the instruction-set description-code fragment in part 1. In part 2, dataflow synthesis has broken up the source code "A = B + C + D" into two statements: "E = C + D" and "A = B + E". Controlflow synthesis changes the value-testing-based DECODE statement into a test for equality to zero or nonequalty to zero (see text). This graph can be expressed as microcode in alternative ways (as shown in parts 3, 4, and 5) by using different controlflow bindings to the hardware description.

makes this easy. The hardware description may contain items unnecessary that for a specific microprogram. For example, a microprogram composed only of arithmetic operations may not use a hardware shifter. UMS manipulates an unambiguous graph code representation instead of a textual one that may obscure some side effects. There is therefore no need for UMS to be concerned about the potential side effects of these unused components interacting with the microprogram. The hardware description explicitly describes any undesirable interactions that might prevent inappropriate controlflow and dataflow synthesis.

Often in conventional compilers, different techniques and representations are used to process controlflow and dataflow. In contrast, the UMS controlflow-synthesis process uses many of the same techniques and representation to map microoperations to microengine-specific controlflow. At best, this is a simple process of mapping the corresponding controlflow between the hardware and instruction-set descriptions. Because the dataflow alone is not a completely synthesized microprogram, it may naively require actions that the datapath in isolation could perform but cannot be triggered due to control hardware-design constraints. In this case, the techniques described subsequently require more synthesis effort to adapt a difficult controlflow fragment into a microprogram than the microengine can perform.

The additional microprogram constraints further limit the number of alternative ways to express the microprogram. As these controlflow synthesis constraints are satisfied, they sometimes require more microwords in the microprogram than dataflow synthesis alone requires. Any necessary modification to the microoperations of the original dataflow synthesis may require additional processing to integrate into the microprogram. Where modifications are made, the RUT can be used to choose between alternatives. UMS would invest extra effort more freely to solve any controlflow-synthesis difficulty than it does for dataflow synthesis. This controlflow-synthesis strategy would protect the investment in previous dataflow synthesis. The remainder of this chapter describes how these additional constraints are detected and resolved.

6.2.1 Controlflow expression variations

This section illustrates how controlflow semantics often can be expressed in many alternative ways. For example, the source code in part 1 of Figure 6.1 is essentially an IF-THEN-ELSE statement that allows the value of the A variable to receive one of two values. The corresponding controlflow graph of microoperations produced by dataflow synthesis is shown in part 2 of Figure 6.1 (the figure does not show dataflow arcs). This controlflow graph does not constrain enough (to be satisfied by only one alternative) where each part of the microcode resides in a microword array.

Parts 3, 4, and 5 of Figure 6.1 show alternative expressions of controlflow for the same microoperations and dataflow. In part 3, the conditional branch is located within the microword symbolically labeled PC+0. If X equals zero, then controlflow progresses to microword PC+2, PC+3, and PC+4. Otherwise, the next microprogram steps following PC+0 are PC+1 and PC+4. Part 4 shows a similar pattern except for the reversed conditional branch. Part 5 shows a portion of the A = B + C + D calculation occurring directly after the conditional branch, with the remainder occurring in a remote block of microcode. The techniques designed to choose between alternatives are based on hardware availability and RUT size, and are described later in this chapter.

6.2.2 Controlflow in hardware descriptions

This section discusses the different sources and types of controlflow-node inputs and how they relate to conventionally written hardware descriptions. The ISPS source code for a simplified microengine is shown in Figure 6.2 (without the variable definitions described in Chapter IV). This microengine used for most of the discussion in this chapter is sophisticated enough to support execution of a small instruction set. The hardware described in Figure 6.2 is different than the one described in the instruction set discussed in Chapter VII; this simplified microengine directly accesses memory instead of using an instruction buffer when it fetches an instruction. The microengine of Figure 6.2 is also unusual because it is without support for bitfield-based

```
Microengine :=                                          ! line 1
begin                                                   ! line 2
uPC = 0 next                                            ! line 3
REPEAT                                                  ! line 4
   MainLoop :=                                          ! line 5
   begin                                                ! line 6
   DECODE Reset =>                                      ! line 7
      begin                                             ! line 8
      0 := begin                                        ! line 9
           DECODE uwrd[uPC]<0>                          ! line 10
              begin                                     ! line 11
              0 := Abus = Memory[MAR],                  ! line 12
              1 := Abus = Reg[uwrd[uPC]<1:3>]           ! line 13
              end next                                  ! line 14
           DECODE uwrd[uPC]<4>                          ! line 15
              begin                                     ! line 16
              0 := Bbus = Memory[MAR],                  ! line 17
              1 := Bbus = Reg[uwrd[uPC]<5:7>]           ! line 18
              end next                                  ! line 19
           DECODE uwrd[uPC]<8>                          ! line 20
              begin                                     ! line 21
              0 := ALUbus = Abus + Bbus,                ! line 22
              1 := ALUbus = Abus - Bbus                 ! line 23
              end next                                  ! line 24
           DECODE uwrd[uPC]<9>                          ! line 25
              begin                                     ! line 26
              0 := Memory[MAR] = ALUbus,                ! line 27
              1 := Reg[uwrd[uPC]<10:11>] = ALUbus       ! line 28
              end next                                  ! line 29
           DECODE uwrd[uPC]<12>                         ! line 30
              begin                                     ! line 31
              0 := begin                                ! line 32
                   DECODE uwrd[uPC]<13>                 ! line 33
                      begin                             ! line 34
                      0 := uPC = uPC + 1,               ! line 35
                      1 := uPC = uwrd[uPC]<14:20>       ! line 36
                   end end                              ! line 37
              1 := begin                                ! line 38
                   Now = uPC next uPC = uPC + 1 next    ! line 39
                   DECODE uwrd[uPC]<13>                 ! line 40
                      begin                             ! line 41
                      0 := IF ALUbus eql 0 =>           ! line 42
                             uPC = uwrd[Now]<14:20>,    ! line 43
                      1 := IF ALUbus neq 0 =>           ! line 44
                             uPC = uwrd[Now]<14:20>     ! line 45
           end end end end,                             ! line 46
      1 := begin                                        ! line 47
           uPC = 0                                      ! line 48
end end end end                                         ! line 49
```

Figure 6.2: ISPS source code of the simple microengine hardware description example used in this chapter. See Figure 4.9 for a diagram of its dataflow.

control-tree decoding of the bits in the instruction stream. In a full instruction-set description, control-tree decoding is usually described by a large DECODE statement (such as shown in Figure 1.2) that branches to an address listed in a table indexed by the currently executed instruction's operation-code value.

A transformation rule could simulate the branch table as a sequence of decrements and conditional branches on zero for this pedagogical microengine's instruction-set emulation. If the function input is zero, then the first address from a table would be branched to during application of the hardware conditional branch of line 43 in Figure 6.2. If the function is not zero, then the input would be decremented, and a similar conditional branch (if zero) would indicate the value was equal to one. This algorithm could continue for all possible values of the input variable. Whether or not the microengine possesses facilities for control-tree decoding, the controlflow issues implicit in this microengine design are subsumed in the techniques described in the rest of the chapter.

Information from two data sources determines controlflow within a computer hardware description. The control most often comes from the fields of the microstore (the microprogram). For example, the "1 :=" part of line 35 in Figure 6.2 shows a dependency of a controlflow node to the microstore. Data from the current state of the emulator (such as data fetched from memory or input/output devices) may control conditional branches. "ALUbus eql 0" of line

42 in Figure 6.2 refer to parts of the emulator state. Both of these types of microcode control sources need to be processed by UMS.

Most controlflow nodes contain two input arcs. The first input is a control-dependency-forming part of a controlflow hierarchy (except for the outermost controlflow level, which describes the simulation's beginning). This allows controlflow constraints to be chained together. All controlflow nodes except for REPEAT contain a second input. This second input is the controlflow nodes' data dependency, discussed previously, which typically acts as the "index" to a DECODE statement. This index value may come from a word of the microprogram or the state of the microengine. The controlflow synthesis described later in this chapter processes all of these different controlflow-node arc types.

6.2.3 Controlflow analysis in UMS and microcode compilers. This section describes the code analysis of conventional microcode compilers and compares it to UMS. Description source code contains information (such as whether a conditional branch microoperation is part of an IF-THEN-ELSE or WHILE-DO construct) about the intended semantics of the microprogram. The controlflow, and therefore the potential for microoperation movement, associated with these alternatives is very different. If both the THEN and ELSE regions contain the fragment "X := X + 1", then the fragment could conceivably be moved before or after the IF (or conditional branch). On the other hand, if

the fragment "X := X + 1" is located within a DO region, then it should not be moved outside the loop. UMS performs a global control-and-data-dependency analysis on both the hardware and instruction-set descriptions to understand these distinctions in detail.

The control and data dependencies in the instruction-set description are to be duplicated exactly in the microengine by UMS. After controlflow and dataflow synthesis, each arc, node, and variable of the instruction-set description graph would map to a corresponding part of the hardware. The hardware is capable of performing many more alternative actions than specified within a microprogram. For this reason, the hardware contains many more control and data dependencies than required by the implicit constraints on the hardware described within the microprogram. Therefore, only those arcs, nodes, and variables actually used by the microprogram would be considered active in the hardware, and they unambiguously specify what is required of the hardware to run the microprogram.

In contrast, most microcode compilers (such as described in [SINT80]), simply translate high-level language constructs into microcode without global control-and-data-dependency analysis. These microcode compilers manipulate controlflow while taking a conservative approach (discussed in [FIS79]). Code analysis is based typically on the object code's microoperations instead of using the higher-level semantics of the source code. More extensive

analysis of object code is possible, but without preservation of the original source code's meaning, it is expensive and incomplete (see [HEC77] or [AHOU77]). As a result, the operation of most microcode compilers is based on using less information than is available about the intended behavior of the desired microcode, thereby limiting the types of code movement.

The conservative analysis (see [FIS79]) performed by many microcode compilers partitions the code into regions called basic blocks (or BB). A target microoperation is defined as the next microoperation to be executed after completing the current microoperation. The first microoperation of a BB is the first microoperation of the entire microprogram, the target of a conditional jump, or the target of two or more microoperations. The last microoperation in a BB is a conditional jump, the target of another microoperation jump, or the last microoperation of the microprogram. In all intermediate microoperations of a BB, controlflow proceeds unconditionally from the predecessor to the successor microoperation.

The basic block regions contain only sequential (or straight-line) controlflow. The microoperations within a BB can be freely reordered based on the principles of local microcode packing. In practice, a BB may contain a controlflow statement, typically a conditional jump or multiway branch, appended to its end. This optional controlflow statement always receives special handling by a packer, and is always put into the last microword of the

packed basic block. Since little is understood about the meaning of the controlflow at these the basic block boundaries, microoperations are seldom moved across them (described later). When they are moved, the move is based on similarly conservative rules in traditional microcode compilers.

6.2.4 Compaction

Both local and global microcode compaction are required to complete synthesis. Local microcode packing occurs within blocks of code with only sequential controlflow. Global microcode packing can process microcode-containing conditional jumps and other control constructs. Typically, global packing first packs microoperations locally between controlflow constructs. In UMS, local packing is intended to be based on the resource allocation techniques of the RUT. A later part of this chapter will discuss compaction in detail.

6.2.5 Resource constraint satisfaction

Often there are miscellaneous tasks that need to be performed at the end of controlflow synthesis. Allocation of specific registers and microword addresses are representative of these tasks, which are described at the end of this chapter.

6.3 A Simple Controlflow Mapping

This section will show controlflow analysis and synthesis principles in action using a simple example. The next section describes a situation where this simple controlflow-dependency analysis is not sufficient.

Controlflow dependencies are explicit in the internal UMS node representation. Controlflow dependencies are usually easy to extract from the structure of the hardware description, where they are controlled only by the microstore. First, all controlflow nodes in the hierarchy are found by tracing their first inputs (the second inputs are the optional data dependencies). Then UMS would find all data dependencies (to the microprogram) for each controlflow node. These usually are read dependencies to fields in the current microword. Internally, UMS addresses sections of code based on their node numbers (shown in Figure 4.5).

Consider determining a hardware match for an instruction-set description fragment that increments the microprogram counter. To simplify discussion in this chapter, source-code line numbers of Figure 6.2 (which describes a simple but viable microengine) label all hardware- description code fragments. The only corresponding microengine fragment matching the dataflow in Figure 6.2 is "uPC = uPC + 1", shown in line 35. Controlflow comes to this expression from line 33 and requires bit 13 of the microword to be zero. Line 33 in turn

receives controlflow from line 30 and requires bit 12 of the microword to be zero. Similarly, line 30 receives controlflow from line 7, where the value of Reset (the state of the computer-console system-reset button) must be zero. The REPEAT statement of line 4 does not require a data dependency for controlflow passes to line 7, only a control dependency. Line 4 receives controlflow from line 1, which is the outermost controlflow level and describes the beginning of the hardware simulation. With the exception of Reset, all data portions of the control dependencies for line 35 come from the array called "uwrd" (which contains the microprogram).

This example shows how the first inputs of the controlflow node determine the hierarchy of participating controlflow nodes. The example also shows how the second input determines selection of the required microword data dependencies (composed of bitfield values). The current implementation of UMS does not perform the controlflow synthesis just described, but it would be easy to add this capability.

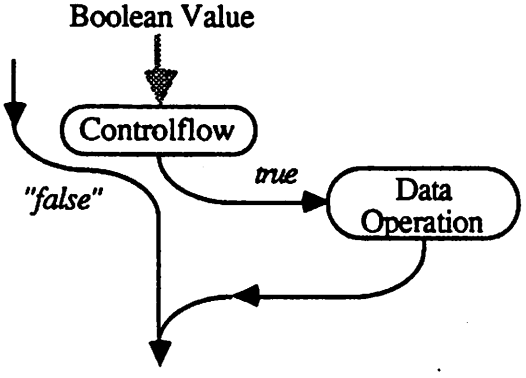
6.4 A Difficult Controlflow Mapping

The controlflow synthesis principles shown in the previous simple example are not sufficient when more than microword field values are used to determine controlflow (such as in conditional branches) or when semantics

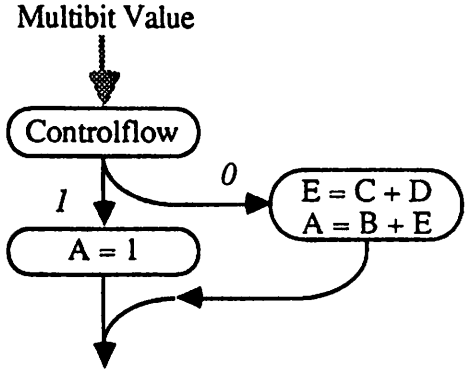
must be modified to migrate between dataflow and controlflow representations. Although controlflow synthesis is not yet implemented in UMS, its implementation would be based on the graph-manipulation process already used for the dataflow synthesis.

Synthesis is based on creating a one-to-one mapping between nodes of the instruction-set and hardware descriptions, where only the instruction-set description may be modified. The hardware description is at a much lower level of detail; one line of the instruction set may require one hundred lines of microcode to implement. The amount of detail in the instruction-set description thus must be increased to achieve the mapping. The hardware description contains a detailed specification of a microengine's conditional branch. The instruction-set description must be modified to describe explicitly a conditional branch so it matches the nodes describing the hardware conditional branch. A more sophisticated form of these graph-manipulation processes are described in an example that matches the instruction-set description shown in part 2 of Figure 6.3 with the hardware of part 1.

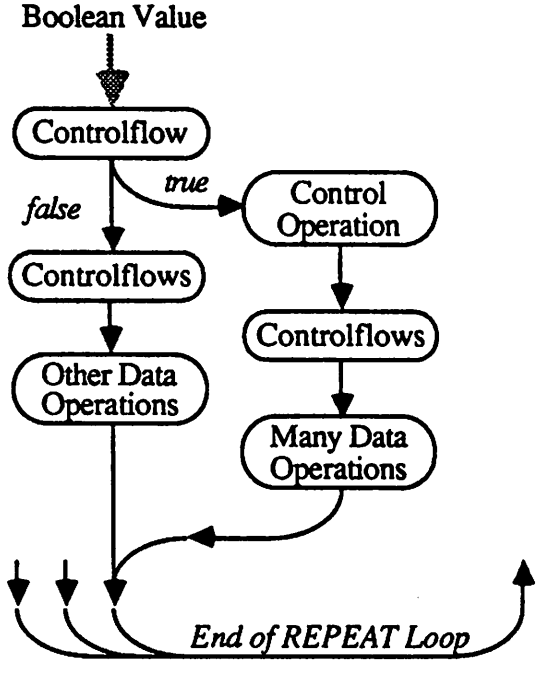
This section will begin by synthesizing the controlflow-node arc inputs to the data-processing microoperations of the example. The remaining controflow-node output arcs are processed next. Last the interaction of the microengine's program counter with the example's controlflow is discussed.



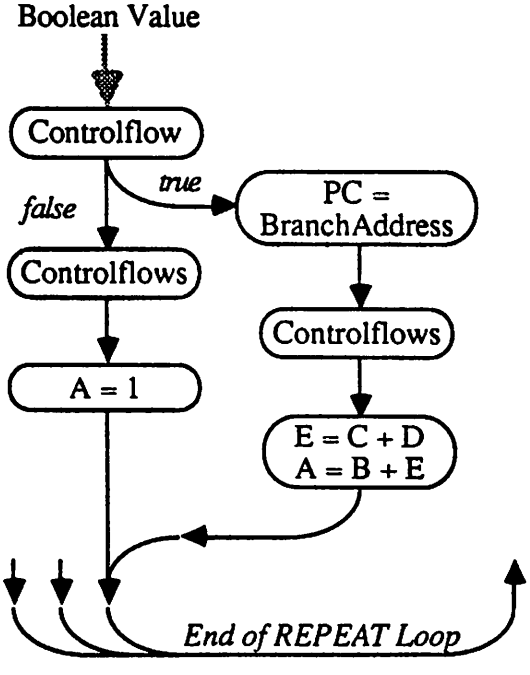
Part 1: Initial Interpretation of Hardware Controlflow for Line 42 in Figure 6.2.



Part 2: Controlflow of Instruction Set IF Statement of Figure 6.1 Part 2.



Part 3: Reinterpretation of Hardware Controflow for Line 42 in Figure 6.2.



Part 4: Transformation of Instruction Set Controlflow for IF Statement.

Figure 6.3: Reinterpretation of controlflow and dataflow in respect to the fetch-execute loop. Initially, the code in line 42 of Figure 6.2 (IF ALUbus eql 0 => uPC = uwrd[Now]<14:20>) is interpreted (see text) as an IF statement that may or may not trigger the execution of a single data operation, as shown in part 1. UMS initially fails to match the instruction-set description graph of part 2 to the hardware-description graph of part 1. This prompts closer inspection of the hardware, which reinterprets the operation in part 1 as a conditional microcode jump because it sets the microprogram counter at the end of a REPEAT loop. The conditional jump may cause the execution of many data operations if the jump is taken or it may allow the execution of other data operations if not taken, as shown in part 3. Part 4 shows how the initial instruction-set fragment can be transformed to match the new hardware interpretation.

6.4.1 Controlflow-node input-arc synthesis

This section considers mapping to hardware the conditional jump from the instruction-set controlflow of Figure 6.1, part 1. X is defined as a multibit value previously determined by a different calculation (not shown in Figure 6.1), not as a single bit value. Earlier dataflow synthesis would have mapped the value X to a Reg register within Figure 6.2. UMS translates both the DECODE and IF source-code statements into the same node-form primitive called "controlflow." Controlflow processing using emulator-state information requires much more effort to map because it combines hierarchical controlflow with specific microengine state dependencies, and either factor may need additional synthesis. The initial controlflow mapping attempt fails because no hardware-controlflow nodes (implicit in Figure 6.2) that contain an input arc from a multibit value and output arcs with both the "0 :=" and "1 :=" values are found. Additional search effort could similarly determine that there is no way to move the value of X (from a Reg register) to the data-input arc of a controlflow statement containing both "0 :=" and "1 :=" outputs. These types of synthesis problems require more powerful techniques than have been discussed so far.

When controlflow synthesis fails, the inference engine would try to isolate the single arc and node mapping failure. Because multiple arcs or nodes might cause failure, UMS would try to identify the first, simplest, or most-basic reason for failure. Once identified, UMS tries to transform only the arc or node into a

different form for successful synthesis. However, the transformation rule may specify changing more than one arc or node. In this example, the data-input arc for the X value is chosen as the initial problem. No attempt is made at this time to deliberately transform either of the "0 :=" or "1 :=" output arcs (although this could happen as part of a complicated transformation); only the troublesome X input arc is targeted for change.

A transformation of the X input arc can be justified because the primary role of controlflow is causality. Specifically, the control or data dependencies leading to a controlflow node may be modified as long as the same stimulus prompts the same behavior. This is less constrained than dataflow transformations, which must always produce the same data values. Therefore, some transformations triggered by controlflow arcs are appropriate only for modification of controlflow because they would garble the meaning within dataflow arcs. Because the production system of UMS is triggered by the need to modify a specific arc, inspection of the arc type as a precondition of the transformation can constrain use to only controlflow arcs.

6.4.1.1 Transformation of input arcs. Transformation rules can describe translations between numeric values, such as zero, and truth values, such as false. In attempting to solve this input-arc synthesis-mapping problem, UMS would test a transformation-rule precondition. This would successfully verify that there are only two (X = 0, and X = 1) controlflow-node output types,

and that one of them is zero. The successful precondition test would allow the transformation rule to transform the input arc of the instruction-set controlflow node from a simple multibit variable value (X as shown in part 1 of Figures 6.1 and part 2 of Figure 6.3) to a truth value (the calculated result of X eql 0, shown in part 2 of Figure 6.1 and part 4 of Figure 6.3). The transformation would also create in the instruction-set description an equality-test (X eql 0) operation node, which has a direct connection to the modified controlflow node. At the same time, the transformation would change the output arcs from referencing values (1 and 0) to truth values (true and false, in this context meaning "equal to zero" and "not equal to zero") to correspond to the input test. The transformation result allows the instruction-set nodes to look more like part 3 of Figure 6.3.

6.4.1.2 Completion of input-arc mapping. The mapping attempt would resume with the modified controlflow node and its directly connected equality-test node of the instruction-set description. The inference engine would quickly match the controlflow and equality-test nodes in part 2 of Figure 6.1 (originally a DECODE statement in the source code) to the nodes implicit in line 42 of Figure 6.2 (originally an IF statement in source code, now in controlflow-node form internally). The only other hardware nodes that would match as well in the microengine of Figure 6.2 are those of line 44, which performs a similar IF function. After controlflow-node matching, arc matching

would begin. The inference engine would quickly match up the arcs for the zero constant in the condition of Figure 6.1 part 2 to Figure 6.2 line 42, while the X input arc and the truth value output arcs require extra effort.

The node match would suggest mapping the X variable of Figure 6.1 part 2 to the ALUbus variable of Figure 6.2 line 42. X resides in a hardware Reg register, and a search by UMS would fail to find a way to route the value of X along a bus from the register to the ALUbus variable. The only way to move the value of X to the variable ALUbus is through the ALU. UMS would determine that, after using a simple transformation, it can route a value to the output of the ALU by adding (or subtracting) zero to (from) it. UMS would also need to determine a source for this zero value. The value could come from memory or a register. Note that this zero value is different than the zero in the previously described equality-test node because the previous zero was built directly into the hardware of Figure 6.2 (line 42) and cannot be used for any other purpose. One source of this constant zero, which could be stored permanently in a register, is to subtract any value from itself at processor-initialization time. Some microengines, such as the one discussed in Chapter VII, contain an array of constants in a ROM. For this example, the zero is simply assumed to be available as an element of the register array. This completes the example's synthesis of the controlflow-node input arc.

6.4.2 Controlflow-node output-arc synthesis

As shown in this section, it is even more difficult to map the controlflow-node outputs then the inputs in this example. The candidate hardware controlflow node contains only a "true" output arc (shown in part 1 of Figure 6.3) because it is semantically an IF-THEN statement. The operations that receive an arc from an ISPS controlflow branch (such as shown in the right side of part 1, Figure 6.3) are called a target. The statement following the IF statement (pointed to by the bottom arrow of part 1) explicitly receives a controlflow arc (labeled "false") from the controlflow parent of the IF statement. This statement, however, can be interpreted as an implicit "false" arc from the IF statement controlflow node (meaning the IF-THEN-ELSE has a nil ELSE). The current state of the instruction-set description contains the pattern (a modified version of part 2 of Figure 6.3) of the "E = C + D" and "A = B + E" statements receiving controlflow arcs from the "true" side of "X eql 0", while the "A = 1" statement receives a controlflow arc from the "false" side of "X eql 0." Since the hardware and instruction nodes are similar, but not enough to be mapped, UMS would perform one more transformation in attempt to bridge the difference and allow a successful mapping.

6.4.2.1 Transformation of output arcs. The next transformation-rule application requires detecting the microcode conditional-jump mechanism within the microengine description. The transformation allows executing the

instruction-set-description code blocks for both the "true" and "false" alternatives only when one hardware controlflow alternative (the "true" arc) exists. The one arc must allow execution of only a single statement when no other controlflow alternatives (such as a "false" arc) exist. Implicit in this transformation is knowledge about the nature of hardware conditional jumps and the relationships between one- and two-branch controlflow nodes. Originally, the ISPS language motivated this transformation rule, which allows explicit IF-THEN but not IF-THEN-ELSE statements.

6.4.2.2 Detection of common microengine components. Many hardware descriptions stereotypically express the widely used microcode conditional branch, which UMS could detect. Program transformation of instruction-set descriptions can use this knowledge about the detected branch. The primary use of a conditional branch is to affect the behavior of the microengine fetch-execute loop hardware. In ISPS source code, the fetch-execute loop in a microengine description can be formed explicitly with a REPEAT loop (used in Figure 6.2) or implicitly with a RESTART statement (used in the example of Chapter VII). At the beginning of this fetch-execute cycle, a specific word in the microcode array is read to control the microengine in that cycle. Either of two actions can occur: the microword is read explicitly into a buffer word (as used in the example of Chapter VII) or its bitfields are read repeatedly during the cycle (as shown in Figure 6.2, lines 10, 15, 20, 25, etc.).

Either of these alternatives is easy to detect due to the complete control-and-data-dependency analysis performed by UMS. The microword is the most common controlflow-node input data dependency. Similarly, the microprogram counter is the most common indexed microword-array data dependency within the hardware description (again see Figure 6.2, where the microcode array is named uwrd and the microprogram counter is named uPC).

The analysis just described allows detecting the microword array and the microprogram counter in the hardware description. These variables fulfill a special role because they primarily determine controlflow in microprogrammed hardware. To improve the possibility of a match with the hardware, temporary variables may be created freely and added to the instruction-set description, however, only a few specific permanent variables may be added. These permanent variables are the correlates of the microword array, the microprogram counter, and variables representing any hardware input or output ports. This example will show how the detected microword and microprogram counter can play important roles in specialized transformations.

6.4.2.3 Hardware conditional branch analysis. A conditional branch is defined as the last time a value is assigned to the microprogram counter within any controlflow path through a microengine fetch-execute cycle that also immediately precedes an IF statement. The IF statement must also contain an input data dependency that is not based on the value of a microword

so that the microengine's input data calculates the condition. Using this definition, it could be determined that lines 43 and 45 of Figure 6.2 are the only conditional branches for the microengine. Further, these lines of code could now be interpreted correctly as part of microengine controlflow processing. As discussed below, reinterpreting these nodes as being part of controlflow would strongly affect controlflow synthesis. Since detection is based on general microengine-design principles and not on features of a specific hardware description, this branch analysis is valid for most microengine designs.

The hardware-controlflow node-map candidate (line 42 of Figure 6.2) contains just one target operation (uPC = uwrd[Now]<14:20>). Inspecting the target of the hardware's conditional-branch map candidate (line 42 of Figure 6.2 and part 1 of Figure 6.3) shows that the candidate includes both the microword array and the microprogram counter, and one of them is written. This suggests that the conditional branch target is not an ordinary data-manipulation operation (as first suggested in part 1 of Figure 6.3) but that it further describes controlflow in the microengine. On closer inspection, it could be determined that the target is an unconditional jump, since the microprogram counter is written. The combination of the target unconditional jump and its parent controlflow operation can be viewed during synthesis as a conditional jump.

Synthesis of a single controlflow arc of the instruction-set description might require the participation of multiple controlflow arcs of the hardware description. Of immediate interest would be the controlflow arc labeled 0 in part 2 of Figure 6.3. In this case, search would be performed to find a sequence of hardware-controlflow arcs. These would range from the controlflow-arc source (the hardware controlflow-node map candidate corresponding to the instruction-set controlflow node in part 2 of Figure 6.3) to the controlflow-arc destination (the hardware nodes mapped during dataflow synthesis to the first of the instruction-set description operations "E = C + D" and "A = B + E"). The target of the true hardware-controlflow arc, which was previously interpreted to be the "Data Operation" hardware description in part 1 of Figure 6.3, would be considered a "Control Operation" (shown in part 3) and therefore classified as just another controlflow arc. This part of the arc-mapping attempt would therefore not be immediately successful.

Since much of the synthesis effort with these hardware candidate nodes has been successful, extra effort would be made to continue with them. Further search finds controlflow arcs in the hardware description (lines 43, 46, 49, and 4 of Figure 6.2) subsequent (or downstream) to the "Control Operation," which assigns a new value to the microprogram counter and now is considered a hardware branch. These arcs lead to the beginning of the fetch-execute loop (illustrated as the controlflow operation following the "Control Operation" of

Figure 6.3 part 3 and line 4 of Figure 6.2). The fetch-execute loop contains controlflow arcs from hardware nodes; both these were mapped previously to the instruction-set description operations "E = C + D" and "A = B + E" (shown as the "Many Data Operations" of part 3). This set of consecutive hardware-controlflow arcs satisfies the original synthesis requirement: a path from the original controlflow node to the two addition statements. After the "Many Data Operations," a final pass through the end of the fetch-execute loop would prepare the hardware to process the next instruction, as shown in Figure 6.3 part 3. This analysis shows that it may be possible to complete the mapping. The synthesis effort described below would create this viable mapping.

A similar process would tentatively map the controlflow associated with the "A = 1" operation in part 2 of Figure 6.3 to the hardware interpretation shown in part 3. The implicit "false" controlflow arc in part 1 of Figure 6.3 goes to the end (and therefore wraps around to the beginning) of the hardware fetch-execute loop (the REPEAT statement in Figure 6.2). At the start of this hardware fetch-execute loop, a controlflow statement invokes data operations (shown as "Other Data Operations" in part 3) before the controlflow paths finally merge with the end of the true controlflow path. Note that the reinterpretation and synthesis of hardware controlflow has left unchanged the previously

performed dataflow synthesis, and the extra controlflow-synthesis effort just described would be justified by the previous dataflow-synthesis success.

Now, with the enhanced interpretation of the hardware description (Figure 6.3 part 3), the topology of the description more closely resembles the instruction-set description in part 2 than the original hardware interpretation of part 1. However, the additional controlflow nodes in part 3 each must be mapped to part of the instruction-set description represented in part 2. This would require adding new instruction-set nodes to part 2 until it looks like part 4. Some of the extra instruction-set-description detail, which should not be matched with the data-manipulation semantics of one of the instruction sets, would match the hardware conditional-branch target now identified as a control operation. The next section describes the process that adds these nodes to the instruction-set description.

6.4.3 Completion of controlflow mapping

At this point in microprogram synthesis (in preparation for microcode packing), all of the instruction-set-description semantics would have been matched to parts of the hardware description. However, all of the hardware components, which must be used during specific fetch-execute cycles of the microengine, would not necessarily have been matched in this way. This occurs because the initial mapping of the instruction-set to the hardware-description does not consider controlflow or dataflow context

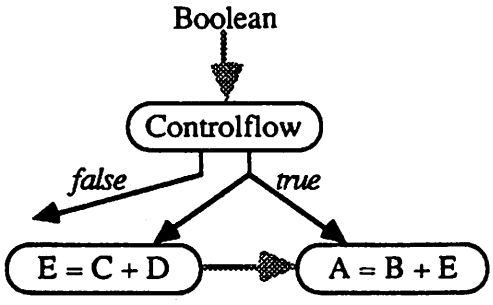
completely. Additional context details would be required to make the hardware perform the desired behavior. Extra effort would be required to transform the instruction-set description until it can be mapped completely to the hardware description, which is at a much lower level of detail and is therefore more complex. Although arc-mapping mechanism extensions (such as not recording the movement of values along a hardware bus) were used during dataflow synthesis to avoid creating temporary variables in the instruction-set description, these extensions are both much simpler than those required for controlflow and take place at an earlier phase of synthesis. Any extensions used earlier in dataflow synthesis are now fixed and would be made explicit in controlflow synthesis.

One of the hardware's most complex control-and-data dependencies is based on the microprogram counter (such as the variable uPC in Figure 6.2) and its affect on the hardware fetch-execute loop. This section continues the example from the previous section, where the mapping of an instruction-set controlflow branch was proposed, based in part on hardware that changes the value of the microprogram counter (line 43 of Figure 6.2 shown as "Control Operation" in part 3 of Figure 6.3). Although manipulating the microprogram counter is relevant to this controlflow-synthesis example, it has not yet been shown how the branch maps to the instruction-set description. This section

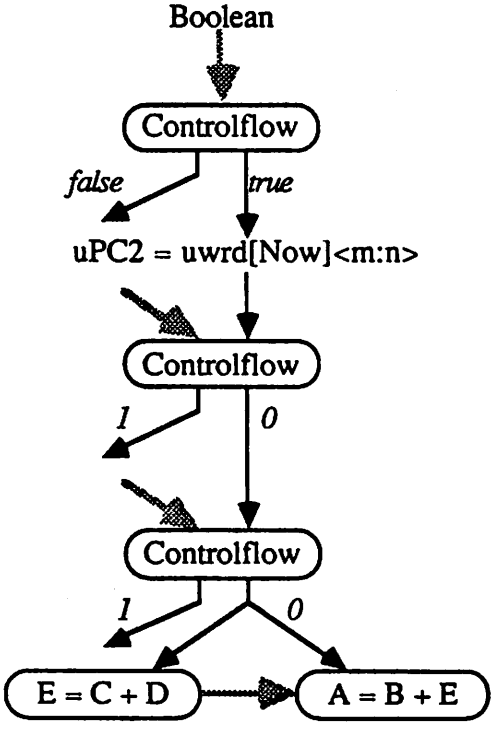
describes the techniques used in that mapping using a portion of the previous example in the presentation.

Following a top-down synthesis order, after synthesis of a controlflow node, synthesis would begin for its sibling nodes. The node number of any sibling nodes would be placed on a stack for their turn in synthesis. In this example, since "A = B + C + D" (the "true" side) originally came before "A = 1" (the "false" side), controlflow for the former would be synthesized first. For brevity, only the "true" side of the example is discussed further, although completion of controlflow synthesis for the "false" side of part 4 in Figure 6.3 would be similar. The ISPS source code for this simplified example is "IF Boolean => begin E = C + D next A = B + E end", as shown in Figure 6.4. The output of the control node is to two microoperations (like the "X = 0" side of the conditional branch in Figure 6.1) with a data dependency between them.

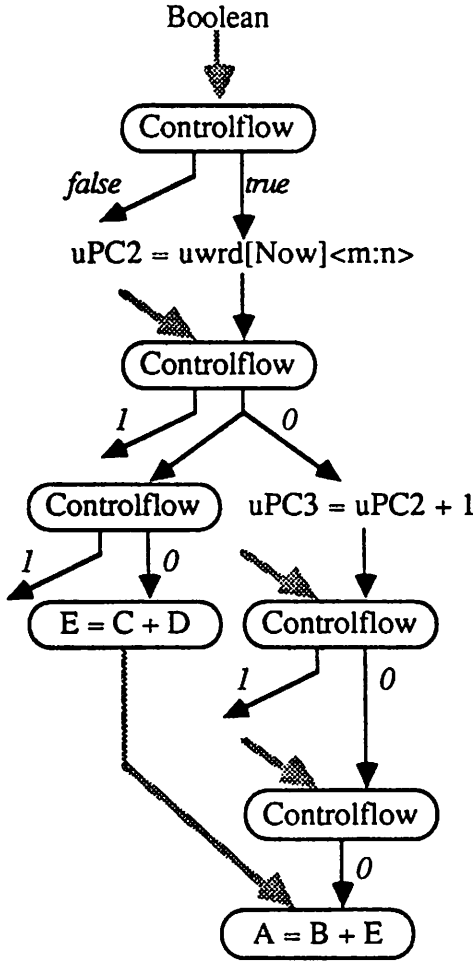
6.4.3.1 Mapping to hardware controlflow-node chains. The data dependencies and arcs leading to a controlflow node may be modified as long as the same stimulus prompts the same behavior; this occurs because the primary role of controlflow is causality. This idea was actually used earlier in this chapter, where the multibit value of Figure 6.3 part 2 was changed to a Boolean value shown in part 4 while the behavior triggered by any data-dependency value was unchanged.



Part 1: Original Controlflow Dependency by Single Variable in Instruction Set Description



Part 2: Modification of Controlflow Dependency to Reset Hardware and Arithmetic Operation Decode



Part 3: Modification of Controlflow Dependencies to Allow Multiple Microword Execution

Figure 6.4: Modification of the data dependencies of controlflow in the instruction-set specification. Part 1 shows the control (small nonshaded arcs) and data dependencies (large shaded arcs) for "IF Boolean => begin E = C + D next A = B + E end" in ISPS; note controlflow originates from the value of a single variable. Since E = C + D writes a value that A = B + E reads, their order cannot be reversed. Part 2 shows a controlflow transformation to allow a variable to indirectly determine the controlflow, through a controlflow jump and two more chained controlflow nodes. Part 3 shows further RUT-triggered transformation, for placement of the two arithmetic operations into different microwords (see text).

Modifications are required before the instruction-set description such as shown in part 1 of Figure 6.4 can map completely to the hardware of Figure 6.2 and 6.3 part 3. At this point in the example, the first controlflow node and the Many Data Operations of Figure 6.3 part 3 have been mapped between the hardware and instruction-set descriptions. The modifications must account for the hardware Control Operation and Controlflows of the right-hand path shown in Figure 6.3 part 3. Since the left path of Figure 6.3 part 3 requires similar processing to the right, it is not considered in the remainder of the discussion.

In the hardware and instruction-set descriptions of this example, earlier controlflow synthesis had found tentatively a controlflow path from the controlflow node to the first of the two addition statements. The inference engine would have overlooked deeper problems with this synthesis until it had solved more fundamental problems (just described). Now, the inference engine may notice that no assignment statement in the instruction-set-description pathway corresponds to the hardware statement "uPC = uwrd[Now]<14:20>" (shown in line 43 in Figure 6.2 and as the Control Operation in part 3 Figure 6.3). Mapping the instruction-set-description arc originating from the first controlflow node to the hardware fails because it connects to this assignment statement (compare parts 2 and 3 of Figure 6.3) instead of the expected addition node. UMS determines which arc-mapping attempt fails, then attempt to modify it. Variable remapping of the statement

operands is not attempted because these variables do not contribute directly to the mapping failure. A transformation of the instruction-set description could match it to the hardware description. Part of the transformation could create the proper synthesis-table entries to make it appear as if the new nodes had been part of the original dataflow synthesis.

A UMS transformation rule adapted for this situation creates new variables and transforms the instruction-set description. Most program transformation rules of UMS completely specify sets of nodes in their "before" and "after" configurations. However, node prespecification is not required by UMS, which is flexible enough to allow specifying nodes at rule-application time.

Previous analysis determined that the "uPC = uwrd[Now]<14:20>" statement is a controlflow branch and not a normal data value assignment. UMS checks at rule-application time to see if the hardware variables uPC and uwrd of Figure 6.2 have been mapped to any instruction-set variables. This variable mapping has not yet occurred in this example. Because these hardware variables have been previously analyzed to be the microprogram counter and microword array, special permanent variables to map them to would be created in the instruction-set description. The variable Now of Figure 6.2 would be mapped to a unique instruction-set-description temporary variable. Additional nodes using these new variables would then be added to

the instruction-set description to match the hardware description nodes to the instruction-set description nodes. UMS creates new instruction-set variables and nodes to match the hardware during rule-application time. UMS would also specifically assign the microprogram counter to the value of an as-yet unspecified hardware-microstore field: "uwrd[Now]<14:20>." Address binding described later gives this field a specific value.

A process similar to that just described for the hardware-assignment statement (line 43 of Figure 6.2) would add two additional controlflow nodes (shown in Figure 6.4 part 2) to the instruction-set description. This would closely match the hardware-controlflow path shown in lines 7 and 20 of Figure 6.2. The data-dependency inputs to the controlflow nodes are from fields of the microword array. These bitfield values specify implicitly the microcode. The unused control-dependency output arcs (in this case, both with the value 1) of these newly added instruction-set-description controlflow nodes are present only to assist mapping to the hardware, not to point to any dataflow nodes.

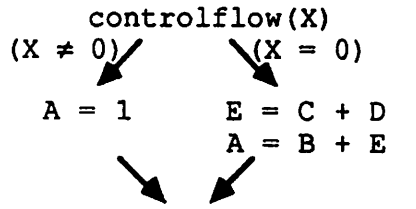
Because a complete controlflow path from the controlflow operation to the first of the addition operations (in Figure 6.4 part 2) now exists in the example, the RUT would be used to determine resource constraints, allocate resources, and perform a type of microcode packing. In this example, resource conflicts would exist in the example (both operations need to use the single ALU implicit in Figure 6.2) and the RUT would require locating the two addition

microoperations in different microwords. Tracing the actual data dependencies checks the decision to move the microoperations. The RUT would be changed correspondingly to extend from two to three the number of time steps allowed for this path through the microprogram. If the synthesizer's inference engine had been designed to overlook control constraints such as the one just mentioned, then the final resource-use measure would be inaccurate. A similar process would trigger creating extra controlflow nodes for the instruction-set description's second addition statement, so that the graph finally would look like Figure 6.4 part 3.

Figure 6.5 shows the dataflow (**bold font**), controlflow, and microword bitfield requirements (*italic font*) for the complete original example of Figure 6.1 and the hardware of Figure 6.2. This simple code fragment hints at the numbing amount of detail common in real microcode. Each label type (single or double letters, lowercase or capital letters) in this figure represents a different pass through the fetch-execute loop of the microengine hardware. Much of the complexity, compared to the original instruction-set description, comes from the details of running the microprogram on the microengine, in particular manipulating the microprogram counter. The control words are the primary factor that add complexity to the modified instruction-set description controlflow. This difficult controlflow mapping has shown how fundamental microengine

a **uPC0 = 0 : Line 3**
b Reset = 0 : Line 9
c *uwrd[uPC0]<0> = 1 : Line 10*
d *uwrd[uPC0]<1:3> = 0 : Line 13*
e **Abus=Reg[uwrd[uPC0]<1:3>] : Line 13**
f *uwrd[uPC0]<4> = 1 : Line 15*
g *uwrd[uPC0]<5:7> = 1 : Line 18*
h **Bbus=Reg[uwrd[uPC0]<5:7>] : Line 18**
i *uwrd[uPC0]<8> = 0 : Line 20*
j **ALUbus = Abus + Bbus : Line 22**
k *uwrd[uPC0]<9> = 1 : Line 25*
l *uwrd[uPC0]<10:11> = 6 : Line 28*
m **Reg[uwrd[uPC0]<10:11>]=ALUbus:Line 28**
n *uwrd[uPC0]<12> = 1 : Line 30*
o **Now = uPC0 : Line 39**
p **uPC1 = uPC0 + 1 : Line 39**
q *uwrd[uPC0]<13> = 1 : Line 40*
r *uwrd[Now]<14:20> = ? : Line 43*
s **controlflow(ALUbus) => uPC2 = uwrd[Now]<14:20> : Line 43**

General Microprogram Controlflow:



aa Reset = 0 : Line 9
bb *uwrd[uPC1]<0> = 1 : Line 10*
cc *uwrd[uPC1]<1:3> = 2 : Line 13*
dd **Abus=Reg[uwrd[uPC1]<1:3>] : Line 13**
ee *uwrd[uPC1]<4> = 1 : Line 15*
ff *uwrd[uPC1]<5:7> = 1 : Line 18*
gg **Bbus=Reg[uwrd[uPC1]<5:7>] : Line 18**
hh *uwrd[uPC1]<8> = 0 : Line 20*
ii **ALUbus = Abus + Bbus : Line 22**
jj *uwrd[uPC1]<9> = 1 : Line 25*
kk *uwrd[uPC1]<10:11> = 3 : Line 28*
ll **Reg[uwrd[uPC1]<10:11>]=ALUbus:Line 28**
mm *uwrd[uPC1]<12> = 0 : Line 30*
nn *uwrd[uPC1]<13> = 1 : Line 33*
oo **uPC4 = ? : Line 36**

A Reset = 0 : Line 9
B *uwrd[uPC2]<0> = 1 : Line 10*
C *uwrd[uPC2]<1:3> = 4 : Line 13*
D **Abus=Reg[uwrd[uPC2]<1:3>] : Line 13**
E *uwrd[uPC2]<4> = 1 : Line 15*
F *uwrd[uPC2]<5:7> = 5: Line 18*
G **Bbus=Reg[uwrd[uPC2]<5:7>] : Line 18**
H *uwrd[uPC2]<8> = 0 : Line 20*
I **ALUbus = Abus + Bbus : Line 22**
J *uwrd[uPC2]<9> = 1 : Line 25*
K *uwrd[uPC2]<10:11> = 6 : Line 28*
L **Reg[uwrd[uPC2]<10:11>]=ALUbus:Line 28**
M *uwrd[uPC2]<12> = 0 : Line 30*
N *uwrd[uPC2]<13> = 0 : Line 33*
O **uPC3 = uPC2 + 1 : Line 35**
AA Reset = 0 : Line 9
BB *uwrd[uPC3]<0> = 1 : Line 10*
CC *uwrd[uPC3]<1:3> = 7 : Line 13*
DD **Abus=Reg[uwrd[uPC3]<1:3>] : Line 13**
EE *uwrd[uPC3]<4> = 1 : Line 15*
FF *uwrd[uPC3]<5:7> = 6: Line 18*
GG **Bbus=Reg[uwrd[uPC3]<5:7>] : Line 18**
HH *uwrd[uPC3]<8> = 0 : Line 20*
II **ALUbus = Abus + Bbus : Line 22**
JJ *uwrd[uPC3]<9> = 1 : Line 25*
KK *uwrd[uPC3]<10:11> = 3 : Line 28*
LL **Reg[uwrd[uPC3]<10:11>]=ALUbus:Line 28**
MM *uwrd[uPC3]<12> = 0 : Line 30*
NN *uwrd[uPC3]<13> = 0 : Line 33*
OO **uPC4 = uPC3 + 1 : Line 35**

Code Text Styles:

Microengine Data Flow
Dynamic Microengine Constraint
Microprogram Constraint

Variable Bindings:

Reg[0] = X
Reg[1] = 0
Reg[2] = 1
Reg[3] = A
Reg[4] = C
Reg[5] = D
Reg[6] = E
Reg[7] = B

Figure 6.5: Control- and dataflow of the microprogram from Figure 6.1 for the microengine of Figure 6.2. Each line of code has a single- or double-letter label.

features are deleted and how this knowledge is used to reinterpret parts of the hardware description graph for successful synthesis.

6.5 Local Compaction

Local microcode compaction (alternatively called microoperation scheduling, or simply packing) is the process of placing individual microoperations without controlflow forks or joins into microwords (elements of the microprogram memory array). All the typical compiler optimizations (such as folding, eliminating redundant operations, moving invariant code out of loops, etc. [AHOU77], [WULFWGH75], [GRI71]) are assumed to have been performed on the input code by a microcode-packing system (within a microcode compiler). The existing literature (see [FIS79], [MAL78], [MA78], [DEW76], [AGE76] and especially [LANDSM80]) adequately covers construction of early microword-packing algorithms. In UMS, another goal of packing is to fix each microoperation within the group to a specific microcode address (see [TAKTBSK82], [TANKE78]).

This section begins describing local compaction techniques with a discussion of how microoperations are represented in UMS and in traditional microcode compilers. The prioritization of microoperations and the microoperation placement procedures are presented next. Computational

costs, microarchitecture independence are then discussed. Last the work of job-shop scheduling is compared to microcode compaction.

6.5.1 Representation of microoperations

Microoperation definition and classification is crucial to microcode compaction because microoperations define the units of semantics that may be rearranged. In conventional microcode compilers (reviewed in [SIN80], [DAS80]), microoperations are fixed atomic units of computation (such as addition or assignment) created by the code generator. Typically, each individual microword bitfield defines a microoperation. However, decomposition of microengine behavior into atomic units is often arbitrary. In some cases, two closely related microoperations (with neighboring bitfields) may be classified as one microoperation (with a larger bitfield). Efforts to define microengine-description languages suitable for describing a wide range of hardware is a very similar classification problem.

In UMS, microoperations correspond to fragments of the instruction-set description's control-and-data-dependency graph that have been mapped to a specific part of the hardware-description graph. Typically, a small number of UMS graph nodes describe a conventional microoperation. Given many description-modification techniques, the instruction-set fragments mapped to the hardware may have originated in many forms. Therefore microoperations in UMS are not as distinct as in conventional compilers. UMS partitions the

microprogram and hardware into a finer level of detail than the microoperations of conventional microcode compilers. This graph-based microoperation representation potentially allows a wider range of efficiency-improving transformations than traditional code-detail levels.

6.5.2 Microoperation prioritization

To represent data dependencies, typical microcode packers create a DAG of the BB (basic block of code containing sequential controlflow) before the block is packed. From the DAG, each microoperation is assigned a level number; this number describes the longest data-dependency pathway to the end of the BB, or equivalently the height of the parse tree. Also recorded are the number of parse-tree nodes that receive data directly from the microoperation and the total number of nodes dependent on the microoperation. A priority ordering is created determined primarily by microoperation level number. Priority-order ties are resolved in favor of the microoperation with the most direct descendants, and then by total number of descendants. In contrast to the UMS RUT, this priority ordering has a major affect in compaction.

6.5.3 Compaction strategies

Conventional microcode compilers process sequentially microwords for local packing. The packer takes a list of microoperations, creates the priority list, and tests microoperations in priority list order to see if they may be placed

into the current microword. The testing considers both control and data dependencies and resource availability. When the list is exhausted or the microword is full, the next microword is used in conventional microcode packing.

Various packing heuristics differ primarily in the order in which they test members of the unplaced microoperations (determined by their placement in the priority list, for example, see [LANDSM80]). When two or more microoperations appear equally qualified for the same position in the test order, one microoperation is chosen for that position. Later decisions are based on this order, so the decision effect is propagated. This limited insight often prevents optimally packing microprograms. Potentially, better packing would occur if such premature decisions between equally qualified microoperations were postponed until more information was available about their relative merit. An important strategy in test-order choice is to make decisions about those microoperations with the strongest constraints first. This strategy is relatively successful and efficient when based only on the length of the chain of data-dependency arcs from the end of the data-dependency graph (see [FIS79], [FISLS81]).

In contrast to conventional microcode packing techniques, the RUT is intended to process microwords for local packing in parallel (even though parts of the RUT algorithm contain serial operations). The number of levels in a RUT

represents the number of microwords required to contain the microprogram; this is implicitly a representation of the length of the data-dependency arc chains within the microprogram. The RUT would use information about both the data-dependency arc-chain length and hardware-resource bottlenecks (first described in [POE81B]) to determine packing order, and the RUT would be used freely across controlflow forks and joins.

This RUT-based local packing is intrinsically likely to provide better results than conventional packing. Most conventional packing algorithms require making arbitrary decisions that impair packing quality (see [FIS79], [MAL78], [MA78], [DEW76], [AGE76] and especially [LANDSM80]). This is done mainly to determine which microoperation should be added next to the partially packed microprogram. In contrast, RUT-based packing completely accumulates control and data constraints for all microoperations. Arbitrary choices are made only when it has been determined within the RUT that a choice will not make a difference in microcode size. Since additional constraints may accumulate, microoperations are bound to specific microwords only after controlflow synthesis is complete.

After using the unimplemented bottleneck-detection algorithm, hardware-resource availability or data-dependency constraints may force lengthening of a RUT. When a RUT is lengthened, a microoperation typically would be allocated to one of a set of microwords or levels, not to a specific

microword. This would allow making weaker decisions than in conventional packing algorithms, which allocate a microoperation only to specific microwords. As more microoperations are placed into a set of RUT microwords, each microoperation-placement set may decrease in size. Only when no more unplaced microoperations remain would the RUT make placement decisions between seemingly equally qualified microoperations. In contrast, conventional microcode packing typically makes these decisions earlier when the candidate sets are the largest, using less information. This reliance on premature decisions by conventional packing algorithms is likely to create suboptimally packed microprograms compared to the RUT.

Before packing, microwords or levels within the RUTs are not associated with specific code (microstore) addresses (such as is required by line r of Figure 6.5). Instead, symbolic addresses (described earlier in this chapter) would be used. When multiple microwords are to be associated with the same address, the RUT interleaving process would then attempt to verify the soundness of the association. After completing both controlflow and dataflow synthesis, addresses would be assigned to microwords.

6.5.4 Cost and computational efficiency of compaction

Testing unplaced microoperations repeatedly for dependency constraints and resource availability is the main factor in the high computational cost of microword packing. Each of these tests is typically a form of list scanning.

Thus, the cost of conventional microword packing increases proportionally much faster than the number of candidate microoperations (see [ULL73]).

Packing algorithms differ in how they chose the next candidate microoperation from a list of unpacked microoperations for placement into the current microword. Fisher ([FIS79]) used the Fernandez and Bussell lower bound ([FERB73]) to determine the quality of different microword-packing algorithms. This produces a lower bound on the size of the packed microprogram when data precedence, but not resource constraints, is considered (see measurements in [HENMA83]). After many experimental trials, Fisher determined statistically that two packing algorithms produce results of nearly the same quality, and these algorithms are also very close in quality to the Fernandez and Bussell lower bound.

The best-performing packing algorithm is rather complex, taking 26 times more computer time to calculate than the second-best algorithm. The second-best algorithm, which was implemented in the global packer (discussed below) of the V-Compiler, performs within .8% of the first in quality. This algorithm is very simple: order the microoperations according to their height in the data-dependency graph. The next microoperation candidate for placement is the highest remaining one in the graph. The RUT is in part based on this simple heuristic that determines the number of potential microoperation-placement sites.

Fisher ([FIS79]) argues that local packing is relatively inexpensive. It takes 142 milliseconds to pack 40 microoperations on a CDC 6600 supercomputer ([FIS79]), 2 seconds to pack 24 microoperations on a VAX ([ATK84]). Fisher argues that this even makes it feasible to recompact selectively some sections of code, particularly for the final version of a microprogram. Although microoperations are inexpensive to compact once, microprograms for complex instruction-set machines such as the Digital Equipment VAX contain many thousands of microoperations and typically are recompiled thousands of times during their development. This author argues that microcode packing is nevertheless expensive when considered from a CPU-development project point of view, particularly when microcode is repacked while microoperations are moved between basic blocks ([POE80]). Further, when packing becomes part of a search-based synthesis process (such as described elsewhere in this thesis), it then becomes critical to decrease the total computational cost. In UMS, the potential interaction of synthesis search and RUT-based microcode packing may achieve significant savings compared to the sum of these individual computational costs.

6.5.5 Compaction and microarchitecture independence

In microcode generation systems for multiple microengines, the ease of retargeting the compaction subsystem is a major concern. The bulk of the V-Compiler microcode-packer code is machine independent, so that a user

may easily adapt it to many different microarchitectures. Most of the packer code processes only the control and data dependencies of a microprogram. The packer software is partitioned so that the user need write only two small subroutines, called FIT and COMBINE (first described in [PATGPS81]), to apply the code applicable to a new microarchitecture (see Figure 6.2, [SIEBN82], [KLAD81], [SINT80], or [BELN71] for what such a design may look like). Otherwise, the operation of the packer is "blind" to the target microarchitecture.

FIT takes a description of the candidate microoperation (chosen because on its control and data dependencies) and a description of any microoperations previously placed into the candidate microword. These descriptions are implemented as lists, and their structure is known only to FIT and COMBINE. The output of FIT is a truth value that determines if resource constraints prevent the microoperation from "FITting" into the microword. If the packer is to place the microoperation into the microword instead of just testing if it is possible, then COMBINE is called. COMBINE receives the microoperation and microword descriptions and modifies the microword description to record the presence of the microoperation.

In contrast to the V-Compiler packer, the UMS system does not require the user to write specialized subroutines such as FIT and COMBINE. Instead, UMS uses techniques to automatically extract information from the ISPS hardware description. The RUT would use this information to perform a type of

microcode packing. Instead of being virtually ignored by the V-Compiler packer until the last stage of the algorithm, information about the hardware would become one of the primary decision-making criteria in UMS. Further, microarchitectural information implicit in the RUT may often affect synthesis search, and allow closer search-tree pruning. These two techniques, the V-Compiler's specialized hand-written machine-dependent subroutines and UMS's microarchitecture description analysis represent two fundamentally different approaches to solving the compaction retargeting problem.

6.5.6 Microcode compaction as job-shop scheduling

One of the major strengths of Fisher's thesis is identifying the similarity between packing microoperations in microwords and the scheduling of processor resources (such as described in [COFF76]). This association made possible the estimate of a lower bound on the number of microwords needed for a packing described earlier. From this point of view, it has been shown that the packing problem is very difficult to solve optimally (NP hard, see [ULL73]). However, one class of algorithms has been empirically shown to determine a nearly optimal solution with a moderate amount of computer resources (see [MAL78], and [FIS79] for the experimental results). Nevertheless, this association does not suggest the possible synergy between synthesis and packing described earlier by using the RUT to prune the search space. This concludes the discussion of local compaction techniques.

6.6 Global Compaction

This section discusses global packing algorithms from two main points of view: one for packing within a machine-independent microcode compiler and a second for microcode synthesis. This section will define global compaction, describe some of its corresponding microoperation ordering techniques, and compare traditional, V-Compiler, and UMS global compaction techniques. A discussion of code generation techniques to support global compaction concludes the section.

Basic blocks (or BBs) are microprogram fragments without internal conditional branches. Global compaction occurs when microoperations are moved across BB boundaries and packed. The author implemented a retargetable global microcode packer ([POE80], [POE81A]) for this thesis as part of the V-Compiler [PATGPS81] project. The unimplemented RUT, an alternative packing technique intended for inclusion in a future version of the UMS system, is also discussed in this section. As discussed earlier, the most important difference between the approaches of V-Compiler and RUT is how microarchitectural details are used. While the V-Compiler packer hides these details from the decision-making process, the RUT would base much of its decision making on them.

Global microcode packing is motivated by observing that some sets of microprogram instructions are executed much more often than others; this is most apparent in instructions that compose the internal portions of a loop. In a microprogrammed central processing unit application, this difference in execution frequency occurs because some types of user instructions are performed more often than others.

In one of the two extremes for microword packing, a set of microoperations may be tightly packed together in a small set of microwords, to the exclusion of microoperations of any other sets. This would allow faster execution of a controlflow path using the set. Alternatively, a set of microoperations may be placed sparsely in a set of microwords that also contains microoperations that fulfill another function. If this other function is of lesser importance (i.e., the average path through the controlflow uses its result less often), then the average execution time is slower than in the previous case. However, a net decrease in the total number of microwords may be observed in the second alternative. Because typical microcode contains many conditional branches, these types of packing heuristics require moving microoperations across them. It is important to use this type of speed/space tradeoff information in the packing of microprograms. To date, global packing implementations have been biased toward improving speed without these secondary space

considerations, probably with the expectation that the primary packing strategy provides the proper balance.

6.6.1 Basic block compaction order

Global microcode packing requires detailed understanding of controlflow constructs, such as conditional branches. The forking controlflow paths of an IF-THEN statement can be optionally labeled (see [POE80], [POE81A]) with the probability for taking each path. For a DO-WHILE statement, the label describes the number of times the loop is typically executed. These parameters imply the relative amount of use for each basic block, and help set up priorities within the global packing process. Although the programmer may initially make guesses for the values of these parameters, more realistic data may be gathered by running simulations of the microcode. UMS currently does not provide for probabilities in the input syntax; nor does it yet perform the type of global packing described here.

Labeling conditional branches in terms of branch probabilities or loop counts describes microoperation use only in a relative, local context. A global measure of block use is needed to prioritize correctly the packing of basic blocks. The priority of the outer-most set of BBs is taken to be 1, or 100%. Where the creation of basic blocks is determined by conditional branches, the absolute priority of BB use is determined hierarchically from the outermost set of BBs to those that are interior to it. In this case, the priority of execution of

each target BB path is calculated by multiplying the priority that the previous BB is to be executed by the branch probability for that path. For loops, the BB's priority is the smaller of 100% and the calculation of the previous BB use multiplied by the expected number of times the loop will be executed. For controlflow joins, the priority of the target BB is the sum of the priority of those BBs that point to it. For subroutines code bodies, the priority is set to the maximum priority of those BBs containing a call statement to the subroutine. Once each basic block has a global priority measure, then it is used to order the packing of basic blocks. Note that instead of a linear decomposition of the microprogram controlflow based on conditional branch source and destinations, this strategy describes a hierarchical decomposition.

6.6.2 Trace scheduling

Fisher ([FIS79], [FISLS81]) suggests a number of methods for choosing BB packing order. His emphasis is to pack completely a controlflow "path" of BBs before considering any other BBs. This information provides the potential for more intelligent global packing than undirected movement of microoperations across conditional branches ([TOKTTY78]) or movement based only on structured controlflow analysis ([WOO79]). Path choice starts at the beginning of the microprogram and moves progressively to the end of the microprogram, choosing as it goes the BBs most likely to be executed until a chain of BBs is found. Alternatively, the choice could begin at the end of the

microprogram and work backward. Another alternative is to choose the most likely used BB and then work both forward and backward from it until determines a path from the start to the end of the microprogram. Fisher's thesis does not explore in depth any of these alternatives, although he suggests that they all would provide sufficiently good results.

The problem with Fisher's approach is that trace scheduling does not necessarily reflect the way a microprogram is used. Consider an instruction-set emulator (such as the one described in Chapter I). Separate multiway controlflow forks and joins provide for fetching each input operand and performing the semantics of the instruction. Fisher proposes optimizing the most common type of operand fetch for the most commonly executed instruction type. This author argues that because the most common instruction type does not necessarily use the most common types of operand fetches, this strategy is likely to provide a suboptimum solution.

6.6.3 V-Compiler and UMS global compaction

This section describes how global compaction was implemented in the V-Compiler ([POE80], [PATGPS81]). This section also describes the intended *UMS* global compaction design.

The BBs scheduled for packing were ordered (or in the case of UMS, final placement of RUT microoperations) by decreasing priority of use, regardless of path. The BBs are packed in the sequence determined by the priority ordering.

Where any BB contains interior BBs with the same priority measure, interior BBs were packed first.

This ordering of the BBs is necessary because of the type of microoperation movement described below. In general, the movement transfers any relatively unimportant microoperations from a higher priority BB to a BB less likely to be executed. This approach avoids the problem in Fisher's thesis of packing a BB less likely to be used on a main path at the expense of moving microoperations into a BB that is globally more likely to be executed but happens to be off the particular chosen path. Implicit in this ordering is that the pathway packing through the microcode completes in decreasing priority order.

The use of the RUT and the RUT's interleaving technique precludes the movement of microoperations described next. However, many of these movements are implicit in the interleaving process. Before the microoperations of the BB are packed into microwords, each microoperation would be tested to determine whether it can be moved out of the BB. The first test would determine whether a microoperation can be moved into a previously packed microword (either prior or subsequent), when that microword already contains microoperations. The microoperation to be moved is the still-unpacked one with the highest priority. This would not preempt microword fields from using previously placed microoperations because the microoperations are already fixed in a packed BB. These movements would not add extra microwords to the

microprogram. In the V-Compiler, conditional branches are not currently moved past one another individually, but they could be if a basic block is moved (described next).

Next, an attempt would be made to move each microoperation from the BB to be packed to unpacked BBs. This would be done without duplicating the microoperations concerned. Fisher states that one should never try to place a microoperation in an unpacked block. His approach is to create a new basic block between the block to be packed and the unpacked block. He then places the microoperation into that new block. This type of movement is considered equivalent to moving microoperations into unpacked blocks.

Note that BBs that receive microoperations are less likely to be executed than the one currently under consideration. At this point, this strategy is risky because there is a chance that the movement will cause a net increase in the number of microwords. This is not a risk in UMS because interleaving and counting the RUT levels would determine if extra microwords are required. If these microoperations remain in the current BB, they might coinhabit microwords that perform other operations instead of being placed in microwords of their own in the candidate BB. A later step would compensate partially for some of this risk. When the recipient BBs are to be packed, an attempt is made to move these same microoperations back into holes in the

original BB. A complete interior basic block may be similarly moved; it would be considered a microoperation with a large set of data dependencies.

Finally, given a favorable cost-benefit analysis, microoperations would be duplicated and moved past joins or forks in the controlflow. The cost-benefit analysis would consider the increase in the total number of microoperations, the likelihood that the current BB will be executed, and the likelihood that the recipient BBs will be executed. If the movement would cause a net increase in the speed of the microprogram (measured by the expected number of microwords executed), then the movement would be made. After these three movements are attempted on each basic block, the block would be locally packed.

The interior of a loop would be packed first in the same way as any other basic block. After meeting two qualifications beyond control and data dependencies, the microoperations would then be moved into the loop's unused microword fields from above or below. The inputs and outputs of these microoperations first must be data independent of those variables modified in the loop. Secondly, if the microoperations modify a variable value instead of only reading or writing it, they must also be executable a variable number of times. Specifically, an increment cannot be performed a variable number of times, but a "store" can be. Instead of moving discrete microoperations, UMS would interleave the sections of code. Just as the RUT simultaneously

considers multiple microoperations in local packing, this global packing strategy simultaneously considers priorities between multiple basic blocks.

6.6.4 V-Compiler code generation for global microcode compaction

This section describes the V-Compiler's intermediate-level language, which facilitates machine-independent global packing and a well-structured interface between many parts of the compiler. When appropriate, the design is compared to UMS, which is intrinsically machine independent. The original goal of the V-Compiler was to produce very high-quality microcode while performing packer-motivated code manipulations in an orderly manner. This approach sacrificed ease of implementation for higher code quality (compare to [VEG82A], [VEG82B]) because of the insight and effort required to tune the code generator. The approach is compared to the intended use of the RUT for global packing (discussed earlier in this chapter). Related but different issues (such as code generation in particular contexts) register allocation, and design details of the microword, are kept from the packer.

To make code generation easier, microprogram in the V-Compiler is described (using productions [CATT78]) as a set of nested functions. Functions may contain a variable number of arguments. As in UMS, each parameter of a function may be an expression of arbitrary complexity and call other functions. The V-Compiler code-generation output (and input to the packer) is a list of microoperations hierarchically structured by controlflow constructs. Four types

of information are embedded within this packer input: 1) the data dependencies (specified by variable definitions, function invocations, and assignment expressions), 2) the controlflow that affects data dependencies across basic blocks, 3) concepts that manually override the action of the packer within a microword, and 4) concepts that manually override the action of the packer between microwords.

Structured controlflow functions are not usually found directly in hardware. Since the object code is a nonhierarchical sequence of microwords, the V-Compiler packer must serialize the structured controlflow found in the code-generator output. In contrast, UMS works with control-and-data-dependency graphs, and such serialization is not required.

To override some of the actions of the V-Compiler packer, UMS uses a pseudomicrooperation syntactically similar to the other packer-input microoperations. A pseudomicrooperation can then be thought of as a machine-independent envelope that labels the semantics of the enclosed set of microoperations. A pseudomicrooperation is used only by the packer and never passed through as output.

The assignment operation is specified by the special function ASSIGN, which contains two parameters: destination and source. As is similarly done in UMS, a set of destinations (e.g., to indicate that both the accumulator and the condition codes are set) is indicated by a list as the first parameter. The second

parameter or source may be an individual variable name or an arbitrarily complex expression. Hardware functions are described through the use of an expression, such as ASSIGN((ACC,CC), SUM(R1,R2)), as an assignment source. Nested assignment statements beyond the type of nesting allowed in the control statements are legal in the language.

Unlike UMS, the packer propagates data dependencies only within nested assignments to the outermost level of the microoperation. The data dependencies then determine the partial ordering between microoperations. No attempt is made to disambiguate the data dependences within a nested assignment statement because it is considered as one microoperation: an indivisible unit of computation. However, this type of nesting may have special significance for the hardware. Currently, microarchitectures with span-dependent jump instructions are not considered, although this concern could be processed with existing techniques ([LEV80], [ROBE79A], [WILL79], [SZY78], [FRIS76]).

6.6.4.1 Controlflow description in the V-Compiler. The controlflow semantics of the V-Compiler's intermediate language is designed to be machine independent and to assist in the serialization process. These pseudomicrooperations and special functions specify the way data dependencies interact with controlflow forks and joins. When controlflow forks and joins are present, data-dependency analysis becomes much more

complicated. However, the semantics describe exactly how to interpret the controlflow, and the computationally expensive and only partially accurate process of deducing the meaning of the controlflow from lower-level primitives is avoided (see [HEC77] for this type of deduction). UMS detects any special hardware constraints in the hardware description. Pseudomicrooperations and other special functions are therefore unnecessary.

CALLs and RETURNs for subroutines have conventional semantics. The subroutine definition contains a list of variables and registers of two categories: input and output. This information is provided in the packer input, whether from the source code or code-generation process. UMS automatically detects this information. The EXIT statement describes a type of controlflow join and contains a single argument that is a label of a block of code that appears later in the source-code stream. It is similar to the EXIT statement of Bliss but allows execution of a final routine and supports common routine tails without the overhead of an extra CALL and RETURN or violating the rules of structured programming. A "snapshot" of the current data dependencies is made when this statement is encountered. When the block is defined, a combination of all such snapshots determines the data dependencies across the EXIT. Instead of using an EXIT concept, the RUT interleaving technique allows common subroutine tails.

RBLOCK is used mainly to describe branch tables. In some microarchitectures, a high-level language's case construct is implemented as a multiway-branch instruction and placed before a table of branch addresses. Each of these addresses takes up a single microword and is actually an unconditional jump, although the microword may also contain other microoperations. Here hardware constraints, such as the requirement for microwords in branch tables to be physically adjacent to each other, are communicated to the packer by its input language. UMS is designed to automatically detect these constraints from the hardware description and use them during address allocation.

The semantics of the RBLOCK statement dictate that the microoperation immediately before the RBLOCK causes a jump to the beginning of the RBLOCK and the last microoperation in the RBLOCK be an EXIT, RETURN, or a jump to the microoperation following the RBLOCK. Only linear controlflow, not control forks or joins, is present. Thus, the packer associates special semantics with the microoperation immediately before an RBLOCK and the last microoperation in a RBLOCK. Position in the packer's input code determines this special meaning, and the packer does not understand anything about the particular microoperations themselves. In contrast, UMS understands the semantics of the microoperations exactly and uses this knowledge to improve microcode quality.

The FLOW pseudomicrooperation describes controlflow forks and joins in a general way and is controlled by its parameters. The first parameter is a set of microoperations that calculate the condition or index that determines which controlflow path is to be taken. By convention, the last microoperation within this set performs the actual controlflow fork (a conditional jump). Each branch of the fork then has an implicit data dependency with this microoperation. All the parameters of the FLOW have an implicit TIE (described later) around them.

The second and subsequent parameters are the blocks of code for each branch of the controlflow fork. It is assumed that only one branch executes each time the FLOW statement is performed. If the FLOW statement contains only two parameters, then an IF-THEN statement is described. Similarly, a three-parameter FLOW describes an IF-THEN-ELSE or two-alternative case statement.

The data dependencies become complex with the FLOW statement. If each alternative path in a FLOW statement redefines (assigns a value to) a particular variable, then that variable is collectively redefined by the FLOW. Each variable may then be potentially defined as a set of microoperations. If only some of the FLOW paths redefine a variable, then the variable is defined either by the FLOW or the microoperations before it.

If two FLOW statements do not have any variables in common, then the packer may change the order in which they occur if they are in the same

hierarchy level; alternately, the packer may pull one out of the other if they are nested. In the latter case, the semantics of the program could be changed. To prevent this, the code received by the packer must be written in a particular way. When the branching of the controlflow is nested, some variable (for example, the microprogram counter) is read and written within the first parameter of the FLOW statement. Using this variable maintains the logical ordering of the controlflow branching. Since controlflow and dataflow are both described in exact detail in UMS, this type of artificial code augmentation is unnecessary to generate code correctly and efficiently.

The LOOP statement contains three parameters: the first one is a test, the second the loop body, and the third another test. The first or third parameter may be left blank to indicate a posttest (DO WHILE or DO UNTIL) or a pretest (WHILE DO or UNTIL DO). The semantics is defined such that the body of the loop may be performed zero or more times.

For timing reasons in the V-Compiler, some microarchitectures require placing a set, or tuple, of microoperations within the same microword. If these sets are enclosed in a set of parenthesis prefixed with a unique function name (such as TUPLE), they are considered an individual but complex microoperation. The packer will not have any special understanding of this function name and treat it as if it was a typical microengine-dependent function. The TIE pseudomicrooperation places the microoperations found in its

(variable number of) parameters in adjacent microwords. Some microarchitectures are given latitude in their timing constraints. For example, after a main memory read has been requested, the data is not valid until after execution of two microword cycles, and the data is valid only during the third, fourth, and fifth cycles. Complex intermediate-language structures are necessary to describe this type of timing constraint in the V-Compiler. In UMS, timing constraints are described explicitly in the hardware description and are satisfied implicitly during synthesis without user intervention.

6.6.5 Equal-edge data dependencies

In conventional microcode compilers, there is a DAG distinction between the normal partial ordering of the microoperations and "equal-edges" (especially see [FIS79]). The "equal edge" type of partial-ordering relation allows placing two microoperations in the same microword (hence their dependencies can be considered to be "equal" in respect to placement) instead of different microwords even though one microoperation writes a value that the other reads. Examples of an equal edge between two microoperations are the setting of an ALU operand size bit and a subsequent add operation. Another example can be found in a register constructed of master-slave flip-flops. The previous value of the register may be read in the same microword as the write because the newly written value is transferred to the output of the register at the end of the microcycle. The view taken by the V-Compiler is that equal edges

are a machine-dependent consideration and as such should not be part of the main packing algorithm. Since UMS works with the unambiguous controlflow and dataflow graph of the hardware, such distinctions are unnecessary.

6.7 Resource Constraint Satisfaction

After dataflow synthesis, controlflow synthesis, and compaction, two tasks remain before UMS synthesis is complete: register allocation and microword address allocation. Part of the decision process has already performed: register values have been bound to specific register arrays, and information about conditional jump destinations has been accumulated. This section describes how these two tasks are performed to bind registers and addresses to specific values.

6.7.1 Register allocation

Registers for temporary variables are intended to be allocated with the RUT just like any other resource with multiple available units. At controlflow- and dataflow- synthesis time, these temporary variables are not necessarily bound to a specific index in the register array. Just before microword addresses are bound, conventional register-allocation techniques ([AHOU77], [WULFWGH75], [GRI71]) would determine register addresses based on the previous constraint accumulation and satisfaction.

6.7.2 Microword address allocation

The synthesizer would not assign addresses to microwords at controlflow- or dataflow-synthesis time. Instead, it would symbolically record any constraints involving each microword address in a microword-constraint cell. A set of pointers would connect the microword-constraint cell to the RUT entries for that microword. A microword-constraint cell may be associated with multiple RUTs. Some of the constraints are for microword field values, such as for line r of Figure 6.5. Microword-constraint cells also contain any other constraints that require satisfaction. One of these microword-address constraints demands that uPC2 (ine L of Figure 6.5) and uPC3 (lines BB onward) must satisfy the constraint described in line O in Figure 6.5 as uPC3 = uPC2 + 1 (also shown in line 35 in Figure 6.2).

After completing controlflow and dataflow synthesis, a part of UMS would try to assign microword address-allocation constraints with the span-dependent jump techniques found in conventional microcode assemblers ([TAKTBSK82], [LEV80], [ROBE79A], [SZY78], [FRIS76], [WILL79], [TANKE78]). If the constraints could not be satisfied, then UMS would attempt other synthesis alternatives or use the RUT to guide further transformations. In the sample microengine of Figure 6.2, any conflicts in microword-address allocation are likely to be solved with the information in line 36 (a controlflow jump) when the information in line 35 (sequential microword allocation) is unsatisfactory.

However, these microword address allocation techniques usually trigger additional synthesis effort.

The microword address allocation just described is the last element in the sequence of additional synthesis constraints described in this chapter. The earlier elements consist of controlflow synthesis, local compaction, global compaction, and register allocation. The RUT data structure and its associated algorithm can potentially aid simultaneous satisfaction of these constraints.

CHAPTER VII

AN EXAMPLE OF SYNTHESIZING AND COMPILING MICROCODE

7.1 The UMS Implementation

This chapter describes the state of the current UMS implementation and gives examples of its microcode synthesis behavior. The UMS implementation consists of over 300 pages of Prolog source code. In almost equal proportions the code consists of an ISPS compiler, a global controlflow-and-dataflow postprocessor, and a dataflow-synthesis inference engine.

If controlflow synthesis was implemented, it would share much of the dataflow synthesis code. The amount of code required by a RUT implementation is estimated to be minor compared to the other UMS components. The current knowledge base of program transformation rules is less than one dozen rules; these were added to UMS as they were needed for the example described in this chapter.

After describing the PDP-8 architecture and a sample microengine, this chapter gives examples of the UMS implementation synthesizing dataflow for PDP-8 emulation microcode. Each microword synthesized by UMS is compared to the hand-written microcode found in [SIEBN82]. The differences found in the microcode are primarily due to different register allocations and

missing controlflow fields. The UMS-generated microcode fields are found to be of quality equal to the hand-written microcode.

7.2 The Digital Equipment Corporation PDP-8

The PDP-8, one of the most widely used minicomputers manufactured by Digital Equipment Corporation, is the architecture used in this thesis to demonstrate microcode synthesis. There are three reasons for choosing this computer architecture: a "real" architecture was desired; its instruction set is of medium complexity (comparable to many current 8-bit microprocessors); and ample documentation of a microcoded implementation and its associated hardware (particularly in the machine-readable form of the ISPS language) was available. Although this implementation was not actually built, it was tested extensively by simulation.

7.2.1 PDP-8 architecture

The PDP-8 operates on 12-bit words within a 4096-word array. Its instructions have limited memory-addressing capability, so that memory is segmented into pages of 128 words each (page offsets of up to 7 bits in size are described in the instruction word, see Figure 7.1). Most instructions may address either page zero (depending on bit 4 of the instruction word shown in Figure 7.1) or the current page as determined by the program counter. The first

eight memory locations in page zero may also be used as autoincrement registers during indirect addressing. The processor's simple logical and integer arithmetic operations are based on the use of an accumulator. A detailed description of this computer, along with the ISPS instruction-set (page 125) and hardware (page 224) descriptions used in the thesis, appears in the book *Computer Structures Principles and Examples*, by Siewiorek, Bell and Newell [SIEBN82].

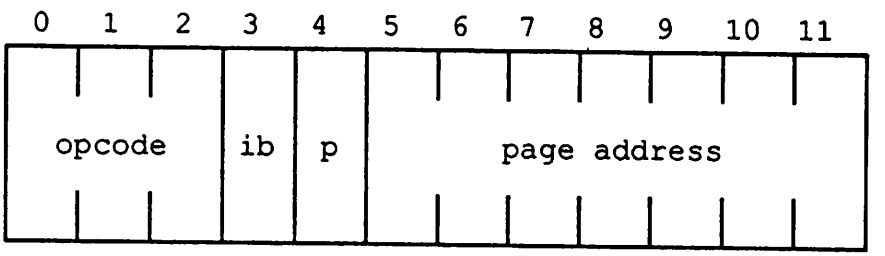


Figure 7.1: PDP-8 instruction format.

7.2.2 PDP-8 hardware

The hardware used in this microcoded PDP-8 implementation is based on the AM2901 bitslice ALU and AM2910 microsequencer (refer to Figure 7.2, see [AMD81] for chip descriptions and Chapters 13 and 14 of [SIEBN82] for discussion of their use). These integrated circuit chips are used widely in industry to implement central processing units. Each ALU integrated circuit performs operations on 4-bit wide data called a slice or nibble. Multiple AM2901s (the PDP-8 implementation uses three) may be connected together to provide arbitrarily wide datapaths in 4-bit width increments. The AM2910

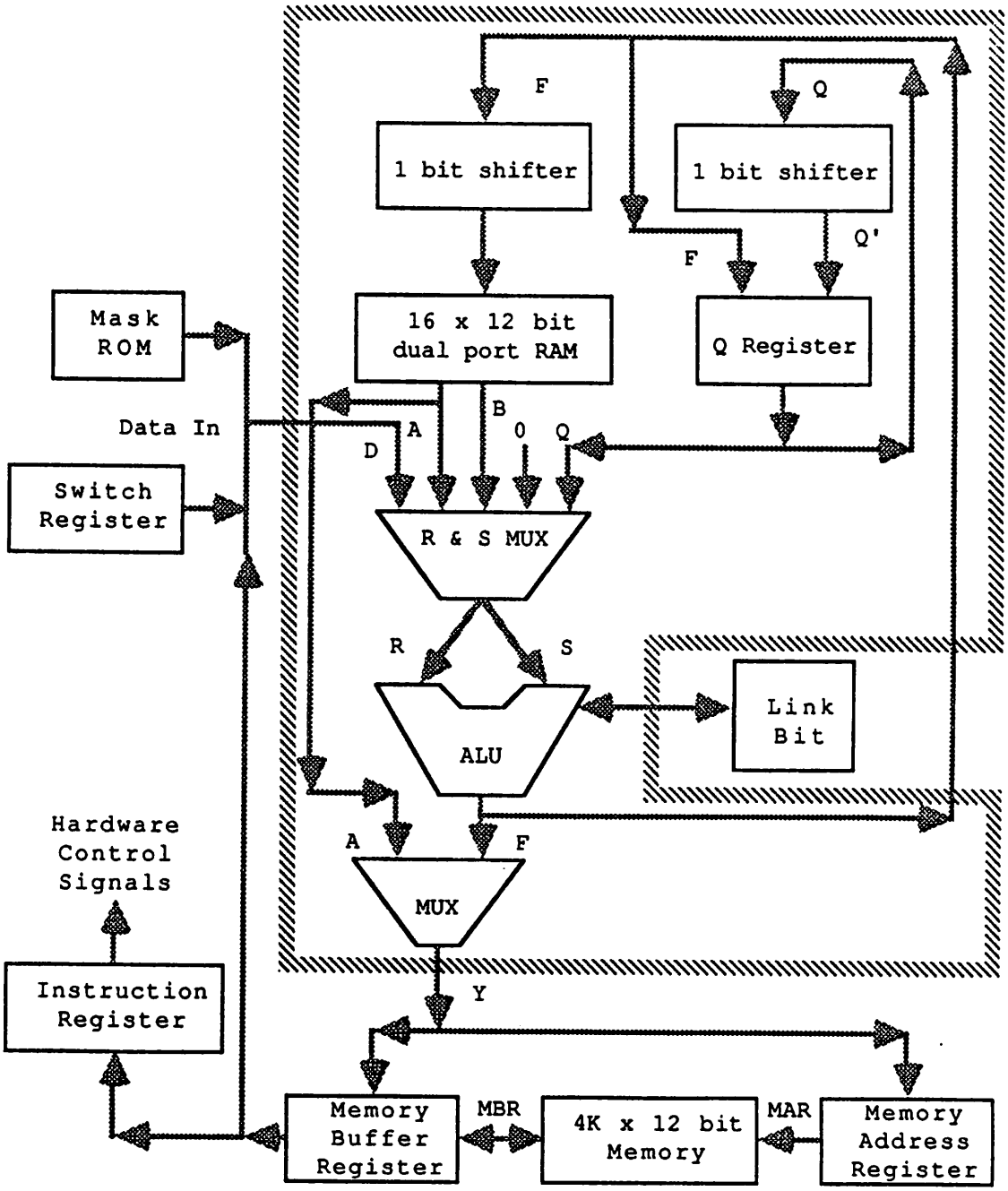


Figure 7.2: Partial block diagram of an AM2901- and AM2910-based PDP-8. The thick outline is the AM2901 hardware boundary (derived from [SIEBN82] and [AMD81]).

microsequencer is a microengine control unit on a chip. It can allow sequential execution of microinstructions, alter the execution sequence based on external conditions, and provide microinstruction subroutine linkage and return. It is designed specifically for use with the AM2901.

The contents of the currently executed microword controls the AM2901 ALU and the AM2910 microsequencer. The AM2910 in turn determines the address of the next microword to be executed. A conventional one-stage microword pipeline is used in the hardware implementation of the PDP-8. Without the pipeline, a race condition would exist when the microword controls the microsequencer. This is because the microsequencer simultaneously controls which microword is executed. The pipeline acts as a register to hold the fetched microword constant while it is executed. The rest of the microengine state is found in the eight registers in the AM2901 ALU (only eight of the sixteen available are used in this microengine), the Q register of the AM2901 (used for shifting), the main memory array, a link bit (architecturally similar to the carry bit), the MAR (memory-address register), and the interrupt-enable bit. Under control of a field in the microword, input data may be received from a switch register (representing the switches on the computer's front panel), a special ROM of constants, or the MBR (memory-buffer register from the main memory array).

7.2.3 PDP-8 microword

The PDP-8 emulation's 70 microwords are 48 bits wide. Bits 0 to 5 control the special-purpose logic that accesses the memory array and controls manipulation of the one-bit wide link and interrupt-enable registers. Bits 6 to 8 control which constant ROM location may be used. Bits 9 to 11 determine whether the data input to the AM2901 is from the MBR, the switches, or the constant ROM. Bits 12 to 14 control which register from the B side of the dual-port RAM is read, and bits 15 to 17 perform the same function for the A side. Bits 18 to 20 determine which registers of the AM2901, if any, are written. Bits 21 to 23 control which arithmetic or logic function the AM2901 ALU performs. Bits 24 to 26 determine which operands the ALU uses. Bits 27 to 32 control condition-code generation. The AM2910 microsequencer allows only one bit of condition code, so this field controls which of many conditions may be tested. Bits 33 to 41 are the microword addresses for conditional jumps and subroutine calls. Bits 42 to 47 determine the sequencing operation of the microsequencer.

7.3 The PDP-8 Synthesis Experiment

The UMS microcode-synthesis experiment described in this chapter involves dataflow synthesis (controlflow is currently unimplemented) of some of

the basic PDP-8 instructions. An annotated trace of this synthesis session is located in the Appendix. Due to the amount of detail involved in microcode and microcode synthesis, only a few fragments are described in this chapter. These fragments show arc and node matching between the instruction-set and hardware descriptions as well as transformations of the instruction-set description and revision of previous synthesis. Because only an interpreter (the C-Prolog interpreter [PER84]) was available for use with UMS, the rate of synthesis is relatively slow. The example shown in this chapter requires about one hour of VAX 780 execution time. Nevertheless, this is estimated to be faster than conventional microcode methodology. UMS does not use any prior knowledge about the hardware or instruction set in this example.

The microengine and instruction-set descriptions were written independently of this thesis as simulations by different authors. The two descriptions used are the exact ones received from Carnegie-Mellon University. They contain small differences from the descriptions found in Chapters 13 through 15 of [SIEBN82]. To show the power of UMS to process descriptions originally created for other purposes, it was decided to use the slightly improved versions obtained from CMU instead of modifying them to match exactly what was published in [SIEBN82]. The microengine description is about 600 lines of source code, which UMS translates into 804 nodes. The

PDP-8 instruction-set description is about 230 lines of source code, which become 269 nodes.

7.3.1 First fragment

The first fragment, "i=M[PC]", specifies how the instruction-set description reads from main memory "M", indexed by the value of the program counter "PC", and stores the value into instruction register "i". After considering three other parts of the hardware, UMS structurally matches "i=M[PC]" with "MBR=MP[MAR]". UMS performs a type of register allocation based upon dataflow constraints, variable bitwidth and, in the case of "M", how many elements are in the array. First it first tries to map the variable "PC" to the hardware variable "MAR". When this succeeds, it maps "M" to "MP" of the hardware, and "i" to "MBR". This synthesis is shown in Figure 7.3, where a "x" microword field value means the field controls irrelevant dataflow to this example or that it specifies controlflow to the next microword. If the microengine is to perform the operation "MBR=MP[MAR]" then the controlflow constraints associated with the microword require the value #10 (8 in decimal) in a specific field (shown in Figure 7.3).

In the long run, however, the hardware memory-address register cannot be allowed to act only as a program counter. As will be shown in this example, when UMS detects additional use constraints about either of these variables, this mapping is changed (this is called variable remapping), and any previous synthesis activity that relied on this inference is updated.

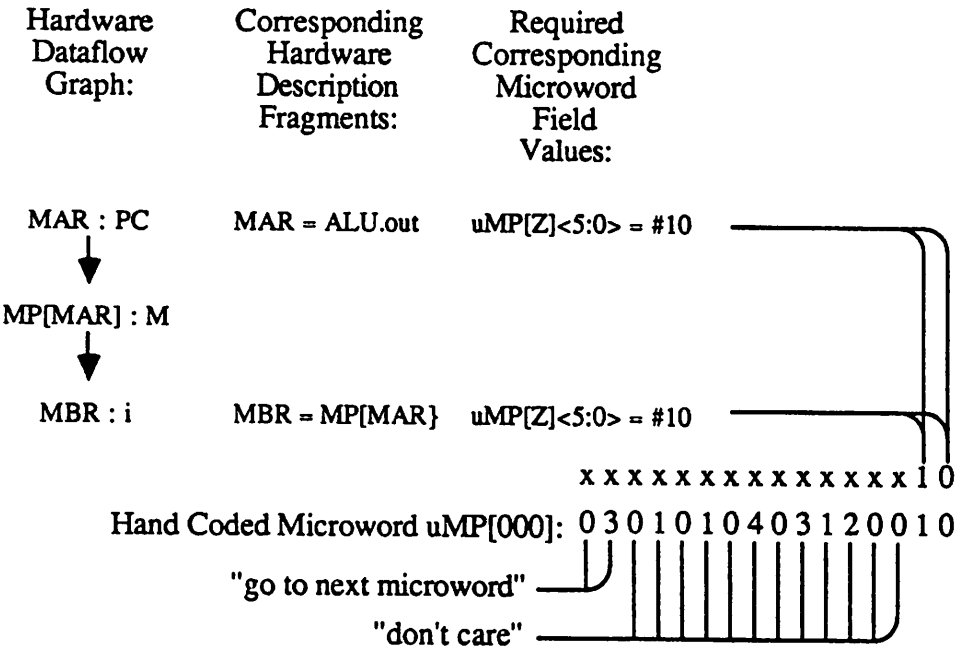


Figure 7.3: Synthesis of the phrase "i = M[PC]" and the initial mapping of "i", "M", and "PC". Hardware data-dependencies and control constraints are also shown. "x" means don't care in the synthesized microword (see text).

7.3.2 Second fragment

The next instruction-set fragment for synthesis is "cpage=PC<0:4>", which extracts the five high-order bits of the address in the program counter. The fragment was first shown in Figure 1.3 in node and arc form. This fragment is slightly different than what was published in [SIEBN82], as noted in Figure A.3. All bitfield extraction operations in the microengine fail to match because they do not provide the correct width bitfields.

UMS keeps a record of the closest match and tries to analyze why search failure occurred. After inspecting this record, UMS focuses its attention on why it can't move the variable "PC" to an appropriate bitfield extraction

operation. In this case, UMS attempts to transform the problem-causing instruction-set node into something else with a semantics-preserving transformation. A programming trick is chosen which transforms the bitfield extraction into performing an AND operation on the value with the appropriate constant (this is shown in Figure 1.4). This transforms "`cpage=PC<0:4> next eadd=cpage@pa`" into "`$Newvar1 = (PC and '111110000000) next eadd = ($Newvar1 or (i and '000001111111))`", where "`pa`" had been specified in the variable definitions to be equivalent to "`i<5:11>`".

Synthesis resumes, and the third hardware candidate can provide a successful match; "`$Newvar1 = PC and '111110000000`" can be mapped to "`alu = R and S`" (part of the microengine ALU). (Actually, because the AND operation is commutative and it is easier to move in the hardware constants to the first ALU input, "`$Newvar1 = '111110000000 and PC`" is attempted to be mapped to "`alu = R and S`"). However, this new use of "`PC`" is not compatible with its previous "`MAR`" mapping, because "`MAR`" cannot be read in the hardware. UMS then tries to find a part of the hardware which can furnish a value to both "`MAR`" and the newly used "`S`". The hardware variable "`A.LATCH`" satisfies these requirements, and both uses of "`PC`" are now mapped to "`A.LATCH`" (shown in Figures 7.4 and 7.5 with the corresponding microword-field controlflow dependencies).

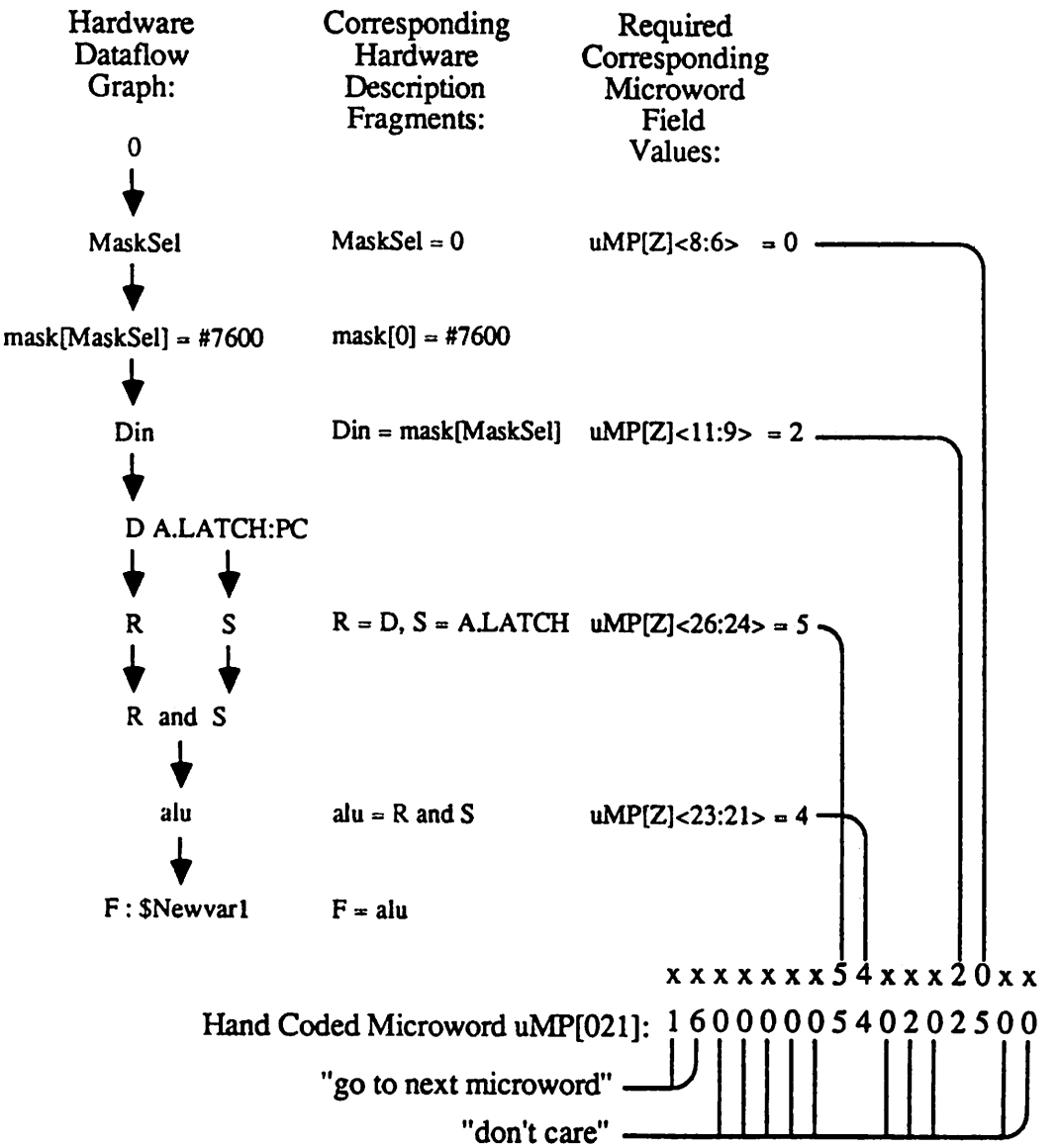


Figure 7.4: Synthesis of the phrase "$Newvar1 = PC and #7600". Commutativity was used for this synthesis. This phrase was derived from the phrase "cpage = PC<0:4>". Hardware data-dependencies and control constraints are also shown. The last synthesized microword field is 0 instead of 5 due to an unimportant difference in register allocation within the constant array. "x" means don't care in the synthesized microword (see text).

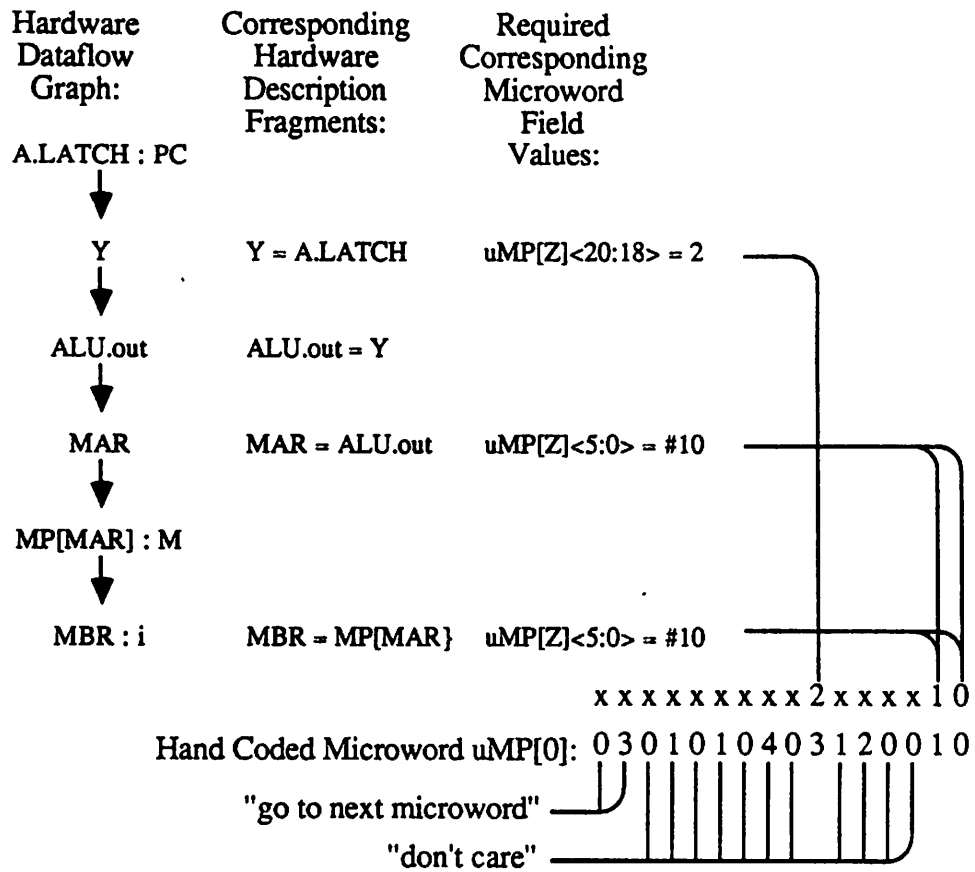


Figure 7.5: Synthesis triggered remapping for "PC" in the phrase "i = M[PC]". Hardware data-dependencies and control constraints are also shown. The microword field difference is due to incomplete synthesis. "x" means don't care in the synthesized microword (see text).

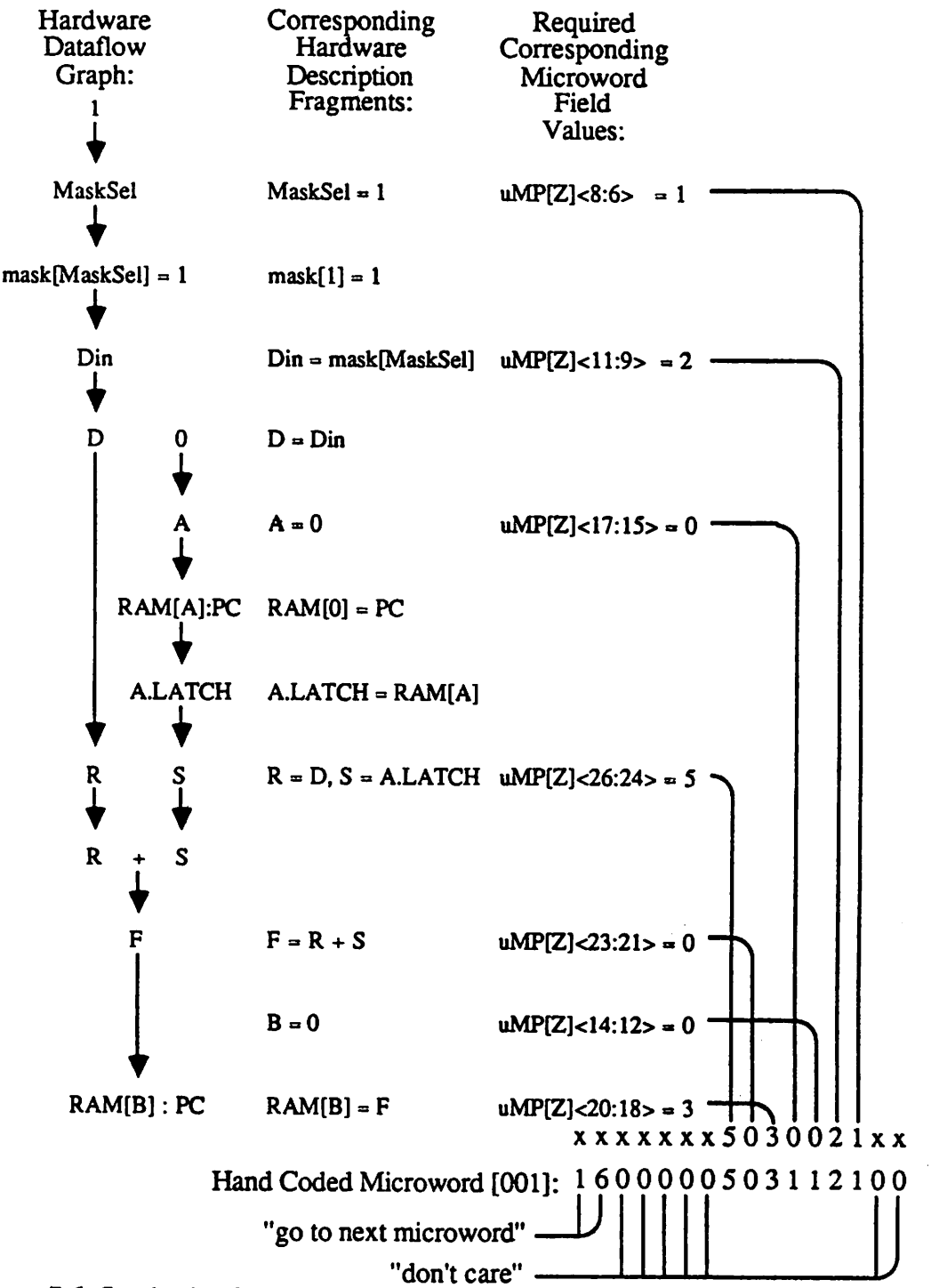


Figure 7.6: Synthesis of the phrase "PC = PC + 1". Commutativity was used for this synthesis. Hardware data-dependencies and control constraints are also shown. Two microword fields are 0 0 instead of 1 1 due to an unimportant difference in register allocation within the register array. "x" means don't care in the synthesized microword (see text).

7.3.3 Third fragment

Initial synthesis search also fails for the instruction-set description fragment "PC=PC+1". The microengine does perform addition, but it takes three arguments: the traditional two and a carry input. After isolating the problem arc and node, a transformation rule changes the fragment into "PC=PC+1+0", which can then match with "alu=R+S+Cn" in the microengine. Figure 7.6 shows this dataflow synthesis and the controlflow synthesis it implies. Now that the value of "PC" must be written (it was only read in previous synthesis), the previous synthesis is changed to reflect the new constraints for "PC". Figure 7.7 shows how these new constraints for "PC" affect the synthesis for the first fragment.

7.3.4 Fourth fragment

The next instruction-set fragment to be synthesized is "eadd='00000@pa", which calculates for the instruction the effective address for the zero-page data-access mode (this occurs when bit 4 of the instruction is zero, as represented in Figure 7.1). Initial synthesis search also fails for this fragment, because the microengine datapath does not directly support this type of bit manipulation. UMS notes that the instruction-set variable "pa" is actually an alternative name for the seven low-order bits of the variable "i". A transformation about bit masking changes the fragment into "eadd='000001111111 and i", providing the microcode shown in Figure 7.8.

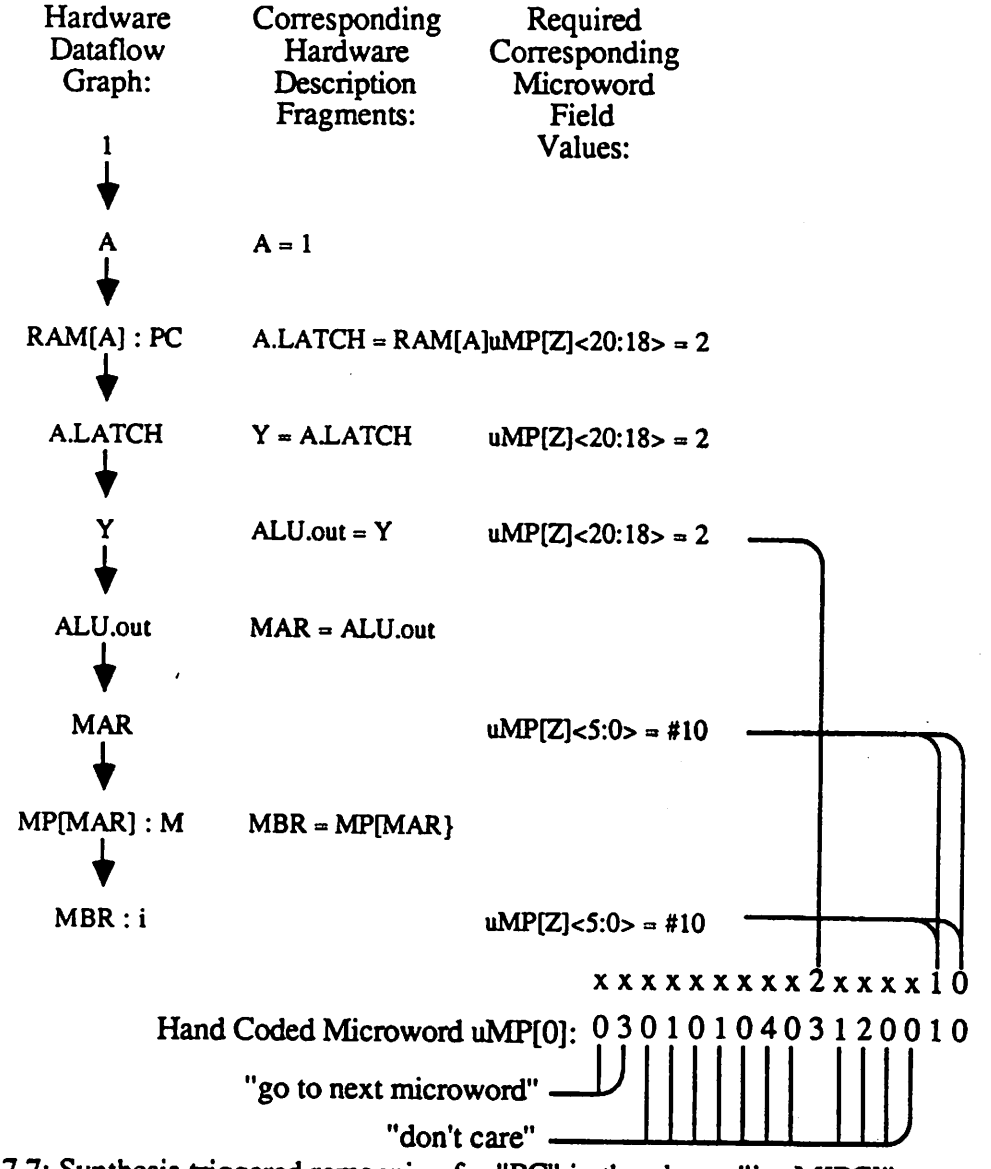


Figure 7.7: Synthesis triggered remapping for "PC" in the phrase "i = M[PC]". Hardware data-dependencies and control constraints are also shown. The microword field difference is due to incomplete synthesis. "x" means don't care in the synthesized microword (see text).

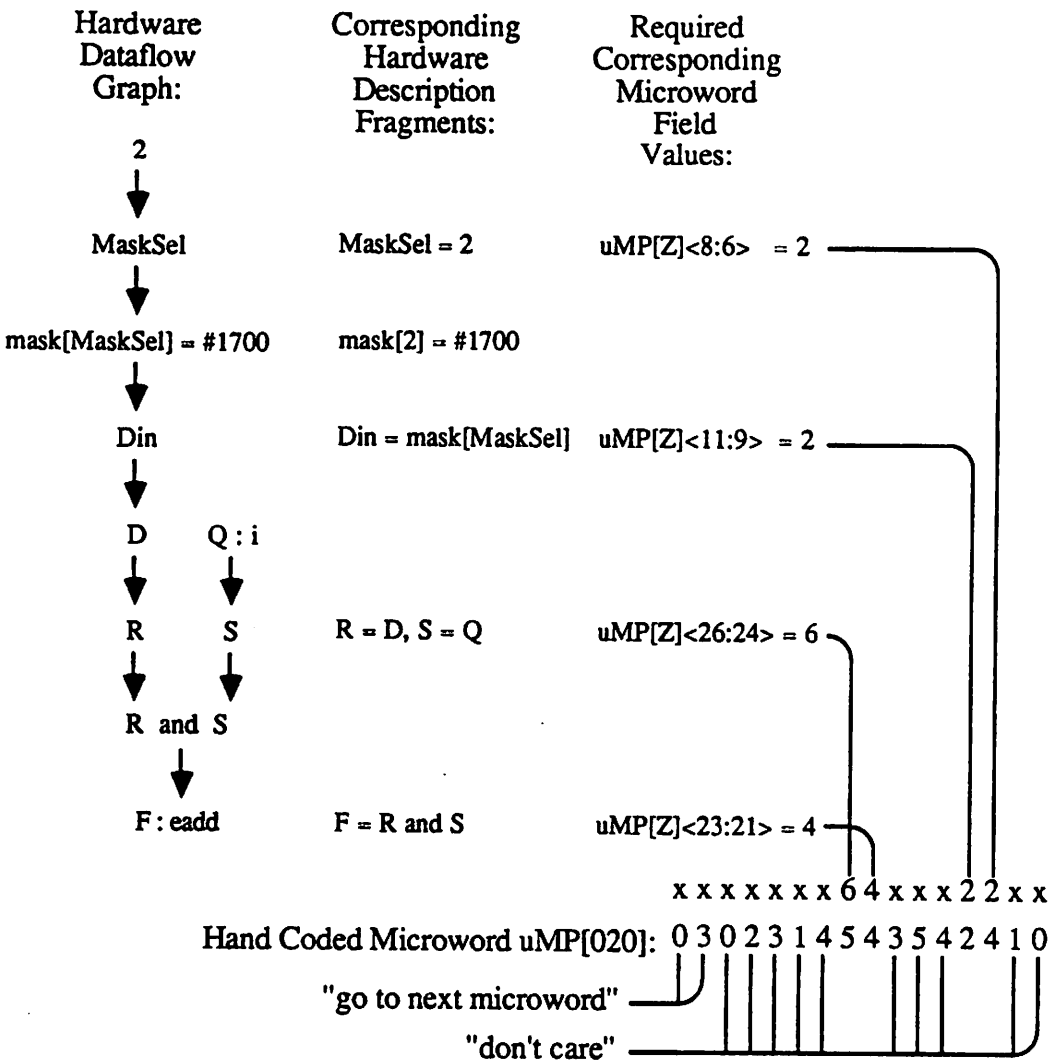


Figure 7.8: Synthesis of the phrase "eadd = #0177 and i". This phrase was derived from the phrase "cpage = PC<0:4>". Hardware data-dependencies and control constraints are also shown. The two sets of microword fields 5 4 and 2 4 are respectively synthesized as 6 4 and 2 2 due to mapping "i" to the Q register instead of RAM and an unimportant difference in register allocation within the constant array. "x" means don't care in the synthesized microword (see text).

The variable "`i`" now has two constraints: it must be used as an output for the previous expression "`i=M[PC];`" and it must be accessible as an input for the expression "`eadd='000001111111 and i`". The initial attempt at remapping "`i`" fails. In this case, a general rule is used which can add a value zero and a

zero carry input to a value to an expression. This rule is used to tentatively change the expression "`i=M[PC]`" into "`i=M[PC]+0+0`".

Although the new fragment "`i=M[PC]+0+0`" also cannot be mapped to the hardware, since UMS has already invested a great deal of synthesis time in the work done so far, it keeps trying by applying still another transformation rule called Rule number zero. Rule number zero spreads complex calculations across multiple microwords by substituting a write and read to a unique newly created temporary variable for a single data-dependency arc. This rule changes the calculation "`i=M[PC]+0+0;`" into "`$Newvar3=M[PC] next i=$Newvar3+0+0;`". These fragments are successfully synthesized.

7.3.5 Summary

In this chapter, four instruction-set fragments were synthesized. All of the code described so far is comparable in execution speed and space to the hand-coded example in [SIEBN82]. This work has shown how UMS schedules in a timely manner its effort between different tasks and subtasks and uses heuristic filters to detect the needed hardware features to run microprograms.

CHAPTER VIII

CONCLUSIONS AND SUGGESTED FUTURE WORK

8.1 Accomplishments of the UMS Project

This thesis demonstrates that expert system technology can be used to synthesize microcode from a hardware description and a description of the desired instruction set. The first naive implementation of the UMS inference engine design required unmanageably large amounts of unguided search. Later implementations used specialized domain-specific search-management techniques. Bus analysis and specialized bus data structures improved search speed. Specialized node-and-arc matching search techniques minimized useless search. The inference engine evolution was based on deeper understanding of the problem domain and the creation of new implementation techniques to exploit this understanding. Similar evolution has taken place from the first search-based recursive decent parsers of the automatic programming field to the LR parser technology of today.

One consequence of the UMS design evolution is that some researchers believe it does not appear to be an artificial intelligence program using classic AI ideas. Instead, it appears to be a more procedural solution, which is a natural consequence of its level of refinement.

The UMS hardware description and code-analysis techniques are critical to the quality of synthesized microcode. As discussed in Chapters I, VII, and the Appendix, this analysis allows optimizations that are not usually found in conventional compilers. However, the useful limits of these hardware description techniques have not yet been critically tested, as the example microengine had only a single-stage microword pipeline and did not use tristate buses. It is expected that no major extensions of these techniques would be needed for further examples.

The examples in Chapter VII show the success of dataflow synthesis techniques based upon graph data structures. However, the unimplemented controlflow synthesis techniques, which would require a significant amount of additional implementation effort, have yet to be evaluated.

Multiple domain-specific search optimizations have been developed as UMS has evolved. These include node-importance sorting, delaying node choice, use of previously successful nodes, sorting arcs in bins, node-and-arc and matching strategies, direct-link stretching, variable remapping, and commutative operator processing which are described in Chapter V. The RUT discussed in Chapters IV and VI and the global microcode compaction techniques of Chapter VI show potential as well. The bus analysis described in Chapter IV also provides an important search speed-up. Only experience with the UMS problem domain has motivated the development of these techniques.

8.2 Implementation Considerations

Given the ease of programming in Prolog, the decision to use this language is not regretted. However, when the choice was made, the intended host was a Digital PDP-10, which is a large mainframe often used for artificial intelligence software development. The Prolog implementation by David Warren [WAR77] provided an excellent interactive interpreter and a high speed compiler. Unfortunately, the large amount of space required by the hardware and instruction-set databases (49K and 144K bytes, respectively, of pretty printed source code) did not fit in the PDP-10, which has only an 18-bit standard address space. This memory limitation was not discovered until the implementation was well under way. As a result, a Prolog implementation that would support a larger address space was sought.

C-Prolog [PER84] running on a VAX 780 was chosen as the UMS host. Unfortunately, C-Prolog is only available as an interpreter. Database search in C-Prolog is serial. Therefore, any time that a node is searched for, even if the node number is known, potentially each record in the database is sequentially accessed. UMS would run much faster if clauses had multiple indexes; hash codes were used internally by the Prolog implementation to classify clauses by operator type and node number. The inference engine code and the two

databases require 50K bytes of atom space and 659K bytes of heap in C-Prolog running on a VAX architecture. When the next generation of Prolog compilers, which will have a higher operating speed and full database indexing, are used with UMS, UMS should run between 10 and 100 times faster.

8.3 Evaluation of Experiments

The experiments with UMS shown in Chapter VII represent about the limit that can be easily performed, since the Digital VAX 11/780 computer run requires more than one hour. For the example of Chapter VII, about 15 nodes, or about 5 lines of high-level source code, are synthesized per hour when beginning with no previous synthesis experience on a new microarchitecture. UMS performs faster as it gains more experience with a microengine. This author suggests that the current performance of UMS is very competitive with conventional microprogramming performed by hand, especially representing the first hour on the job. The actual computational cost of the UMS approach should be competitive, especially because the next generation of engineering work stations will provide VAX 11/780 Prolog performance at a relatively small purchase cost. At this point, microcode synthesis may become very tempting for use in academics and industry.

Future work would involve extending the implementation of controlflow synthesis, especially regarding the RUT. Expansion of the program-transformation knowledge base, particularly with regard to speed and space tradeoffs, and use of UMS with many microarchitectures, including those with interrupts, would be equally important. The user interface to UMS should be enhanced, perhaps along the lines of work by Vegdahl [VEG85]. However, the type of microcode synthesis shown by UMS is indeed practical.

APPENDIX

THE PDP-8 SYNTHESIS EXPERIMENT TRACE

A.1 UMS Startup

The phrase "`prolog -h 2000 -g 2000 -l 1000`" shown in the computer-execution trace of Figure A.1 invokes the C-Prolog interpreter with enough virtual memory to run UMS. The phrase "`['aphalf.nod','pdp8.nod', 'syn.'].`" loads into C-Prolog the Prolog syntax-based hardware and instruction-set description databases as well as the synthesizer. The command "`s1.`" actually begins the processing of UMS on this example.

The phrase "`Synthesize condition of control flow [].`" is part of the trace describing which parts of the controlflow are being synthesized. This phrase is the outermost instruction fetch-execute loop, described in ISPS as a REPEAT loop. This loop has a nil input condition "`[]`", also described by this line of the trace. The phrase "`Synthesize portion of REPEAT loop 67.`" means the next (and first) member of the repeat loop's body (determined by hierarchical, top-down controlflow) is the fragment containing node number 67. The fragments are chosen for synthesis in this order.

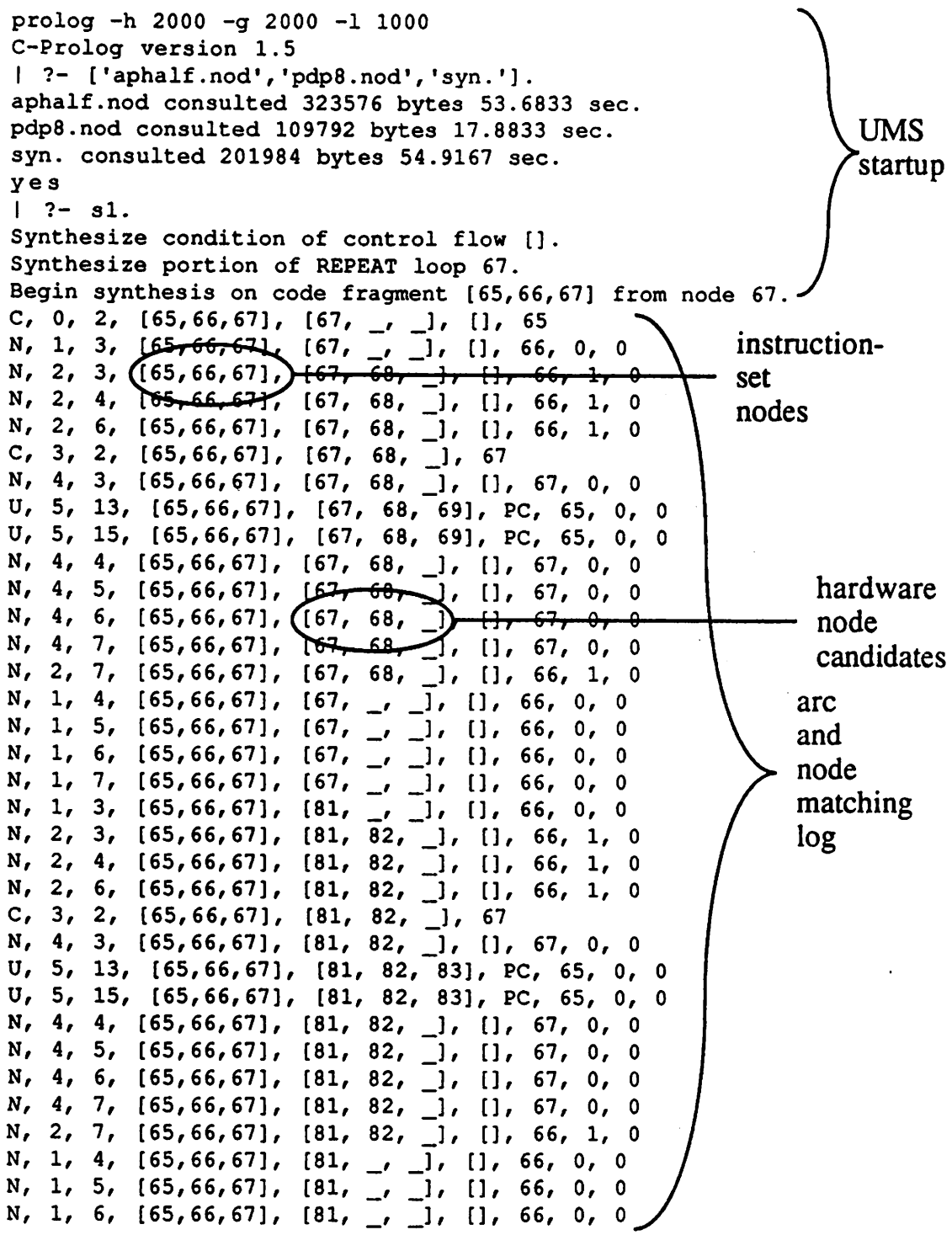


Figure A.1: Trace of UMS behavior while synthesizing PDP-8 microcode (see text).

A.2 Synthesis of First Fragment: "i = M[PC]"

The synthesis of this first fragment proceeds smoothly, demonstrating arc, node, and variable matching. The phrase "`Begin synthesis on code fragment [65,66,67] from node 67.`" in Figure A.1 indicates that the instruction-set fragment consisting of nodes 65, 66, and 67 receives controlflow input through node number 67 in the controlflow node hierarchy. Instruction-set description nodes 65, 66, and 67 are of type WORDS, ARRAYACCESS, and ASSIGN, respectively. They come from the ISPS source-code statement "`i=M[PC];`", which specifies how the instruction-set description reads from main memory "`M`", indexed by the value of the program counter "`PC`", and stores the value into "`i`". The variable "`i`" contains the value of the next instruction to be performed. Note that UMS has no preconceptions about what the intended use of either hardware or instruction-set description variables should be, although it does have descriptions of their characteristics (such as width and, in the case of "`M`", how many elements are in the array). At this point, the primary task of UMS is to determine how to access an array of memory with the appropriate size and shape somewhere in the hardware.

A.2.1 Arc and node matching

Trace lines of the form "`N, 1, 3, [65,66,67], [67, _, _], [], 66, 0, 0`" (shown in Figure A.1) describe the activity of the inference engine on one

node arc. The initial letter describes the type of arc processed; "C" means control input arc, "N" means nested (direct connection) arc, "U" means an arc with a previously unmapped variable, "M" means an arc with a previously mapped variable, and "H" means a hard-to-process nested arc (which may require expensive inferences or resynthesis). The next two integers are the count of the number of successful arcs matched and the numeric label of the arc matching heuristic, respectively.

The next vector, "[65, 66, 67]", in this trace line example is a list of synthesis-candidate node numbers from the instruction-set description. The second vector, which is "[67, _, _]" in this line, is a list of corresponding candidate hardware nodes to match the instruction-set description. The first vector element, instruction-set node number 65, is intended to be matched to hardware node 67. An underscore means that a hardware candidate has not yet been chosen. This means that in this trace line, instruction-set nodes 66 and 67 do not yet have corresponding candidate hardware nodes. The remaining numbers aids in the debugging of the inference engine.

The first hardware candidate for matching with "i=M[PC]" is "uMP.out=uMP[0];", which is composed of hardware nodes 67, 68, and 69. These hardware nodes describe that when power is first turned on, the hardware microsequencer begins execution by reading the microstore at location zero and performing that microword first. This behavior does not match

the intended meaning of the instruction-set fragment "i=M[PC]", so UMS rejects the match.

Arc matching continues until all three hardware-node candidates are chosen (listed near the middle of Figure A.1, when the second vector on a line is "[67, 68, 69]") and a total of five arcs have been matched successfully (indicated by the first integer 5). The second and third hardware-node candidates were chosen primarily because they are connected directly to the first "WORD" hardware node (number 67). Although the syntactic structure of these two expressions is similar, the characteristics of the variables are not. "uMP" and "uMP.out" are the variables defining the 48-bit-wide microstore and microinstruction-buffer hardware, while "M" and "i" are the variables defining the 12-bit-wide PDP-8 main memory and instruction buffer. After determining that the variable characteristics are incompatible, limited search tries to find alternative ways to match the arcs. These alternatives fail, and the hardware-node candidates are gradually "unchosen," indicated in subsequent trace lines in Figure A.1 that contain underscores in the second vector.

The second set of candidate hardware nodes consists of 81, 82, and 83 (shown in the bottom half of Figure A.1). These nodes describe the expression "Din=mask[MaskSel]" within the hardware description. The value of the constant ROM called "mask" is read in by these nodes, as indexed by the "MaskSel" field of the microword, and temporarily stored in the ALU data-input variable called

"Din". Although this expression of the hardware description structurally matches the instruction-set description expression "i=M[PC];", the characteristics of the variables do not match. In particular, the variable "mask" is of type read-only (determined by the dataflow analyzer of UMS) while "M" is of type read/write (PDP-8 main memory can be both read and written). When this part of the search fails, UMS disgards these candidate nodes and searches for new hardware-node candidates.

The third set of candidate hardware nodes that can structurally match the instruction-set fragment are shown in the continued trace of Figure A.2. These are hardware nodes 119, 120, and 121, which come from the source-code expression "uMP.out=uMP[uMP.addr]". This part of the microengine hardware reads in the next microword to be executed. This match fails for the same reasons (incompatible variable characteristics) that the first candidate set was unsuccessful, and this candidate set is abandoned.

A.2.2 Successful variable binding

The final hardware candidate set, consisting of nodes 289, 290, and 291 (shown in the middle of Figure A.2), is a correct choice in this case. When UMS tries to match the instruction-set description fragment "i=M[PC];" with the hardware fragment "MBR=MP[MAR]", it first tries to map the variable "PC" to the hardware variable "MAR". Here, both variables possess the same width (12 bits) and use (read/write) characteristics. To aid further synthesis of this fragment,

```
N, 1, 7, [65,66,67], [81, _, _], [], 66, 0, 0
N, 1, 3, [65,66,67], [119, _, _], [], 66, 0, 0
N, 2, 3, [65,66,67], [119, 120, _], [], 66, 1, 0
N, 2, 4, [65,66,67], [119, 120, _], [], 66, 1, 0
N, 2, 6, [65,66,67], [119, 120, _], [], 66, 1, 0
C, 3, 2, [65,66,67], [119, 120, _], 67
N, 4, 3, [65,66,67], [119, 120, _], [], 67, 0, 0
U, 5, 13, [65,66,67], [119, 120, 121], PC, 65, 0, 0
U, 5, 15, [65,66,67], [119, 120, 121], PC, 65, 0, 0
N, 4, 4, [65,66,67], [119, 120, _], [], 67, 0, 0
N, 4, 5, [65,66,67], [119, 120, _], [], 67, 0, 0
N, 4, 6, [65,66,67], [119, 120, _], [], 67, 0, 0
N, 4, 7, [65,66,67], [119, 120, _], [], 67, 0, 0
N, 2, 7, [65,66,67], [119, 120, _], [], 66, 1, 0
N, 1, 4, [65,66,67], [119, _, _], [], 66, 0, 0
N, 1, 5, [65,66,67], [119, _, _], [], 66, 0, 0
N, 1, 6, [65,66,67], [119, _, _], [], 66, 0, 0
N, 1, 7, [65,66,67], [119, _, _], [], 66, 0, 0
N, 1, 3, [65,66,67], [289, _, _], [], 66, 0, 0
N, 2, 3, [65,66,67], [289, 290, _], [], 66, 1, 0
N, 2, 4, [65,66,67], [289, 290, _], [], 66, 1, 0
N, 2, 6, [65,66,67], [289, 290, _], [], 66, 1, 0
C, 3, 2, [65,66,67], [289, 290, _], 67
N, 4, 3, [65,66,67], [289, 290, _], [], 67, 0, 0
U, 5, 13, [65,66,67], [289, 290, 291], PC, 65, 0, 0
Hw/var "MAR" mapped to inst/var "PC".
U, 6, 13, [65,66,67], [289, 290, 291], M, 66, 0, 0
Hw/var "MP" mapped to inst/var "M".
U, 7, 13, [65,66,67], [289, 290, 291], I, 67, 1, 0
Hw/var "MBR" mapped to inst/var "I".
Begin synthesis on code fragment [70,71,72] from node 72.
synthstack([74,75,76,77,78])
hwinodmap([291,67,0])
hwinodmap([290,66,0])
hwinodmap([289,65,0])
varmap([MAR,10,PC,3,[0,_,_],_,[[WIDTH,11,0,0,11]],1,[65]])
varmap([MP,2,M,2,[1,_,_],_,[[INDEX],[WIDTH,11,0,0,11]],1,[66]])
varmap([MBR,32,I,12,[0,_,_],_,[[WIDTH,11,0,0,11]],1,[67]])
hwdatapath([WORDS,289])
hwdatapath([ARRAYACCESS,290])
hwdatapath([ASSIGN,291])
C, 0, 2, [70,71,72], [94, _, _], [], 71
N, 1, 3, [70,71,72], [94, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [94, 95, _], [], 71, 1, 0
```

arc and node matching log

variable match log

Figure A.2: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

this successful variable mapping is posted to the UMS database, and the message "Hw/var "MAR" mapped to inst/var "PC"." is displayed for the user (shown in the middle of Figure A.2). Similar processing maps the hardware variables "MP" and "MBR" with the instruction-set description variables "M" and "i".

The mapping of these variables is shown in more detail toward the bottom of Figure A.2 in the lines beginning with "varmap", which are Prolog database records describing the mapping. These records also specify (indicated by "[WIDTH,11,0,0,11]") that bits numbered 11 through 0 of "MP" correspond to bits 0 through 11 of "M".

These variable mappings are (temporarily) correct, given the amount of analysis the synthesizer has performed. In the long run, however, the hardware memory-address register cannot be allowed to act only as a program counter. As will be shown in this example, when UMS detects additional use constraints about either of these variables, this mapping is changed (this is called variable remapping), and any previous synthesis activity that relied on this inference is updated. Here the hardware and instruction-set memory arrays are mapped correctly during this attempt; they will not be changed later. This is not too *surprising*, given that both are array variables within the two descriptions that with distinctive characteristics.

A.3 Synthesis of Second Fragment: "cpage=PC<0:4>"

The next fragment of the instruction-set description to be synthesized is "cpage=PC<0:4>", which extracts the five high-order bits of the address in the program counter (shown in Figure 1.3; note in Figure A.3 the source-code differences of the code published in book form). This fragment is represented by instruction-set nodes 70, 71, and 72. The most distinctive feature of "cpage=PC<0:4>" is the implicit bitfield extraction, specified in ISPS with the two integers surrounded by angle brackets. The UMS inference engine searches first for bitfield-extraction operators. Unfortunately, the hardware does not support this type of bit extraction, and the search eventually fails (shown in the bottom of Figure A.2, Figure A.4, and A.5).

A.3.1 Bitfield-extraction operator search failure

The first hardware-node candidate for bitfield extraction is node 94, which is part of the expression "'0 @ ALUport<5:3>" (the trace is shown at the bottom of Figure A.2 as "C, 0, 2, [70,71,72], [94, _, _], [], 71"). This hardware expression truncates a single zeroed bit to three data bits (labeled 5 through 3) from the microword going to the input of the AM2901 description. The value comes from the microinstruction (after some renaming of variables and relabeling of bits, which were detected by the UMS dataflow analyzer) and controls which RAM register is read from the A port of the dual-ported register

Effective-Address Calculation from Carnegie-Mellon University:

```
** Current Page Calculation **

cpage = PC<0:4>

. . .

** 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,
```

Effective-Address Calculation in [SIEBN82]:

```
** Current Page Calculation **

last.pc = PC

. . .

** Address.Calculation **

MA\effective.address<0:11> :=
        begin
        MA = '00000 @ PA next                        ! Zero page
        IF pb => MA<0:4> = LAST.PC<0:4> next         ! Current page
        IF ib =>                                     ! Indirect bit
             begin
             IF MA<0:8> eql #001 =>                  ! Auto index
                 MP[MA] = M[MA] + 1 next
             MA = M[MA]                              ! Indirect Address
             end
         end
```

Figure A.3: Differences in instruction-set specification source code of PDP-8 effective-address calculation between the Carnegie-Mellon University and [SIEBN82] versions.

```
N, 2, 4, [70,71,72], [94, 95, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [94, 95, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [94, 95, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [94, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [94, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [94, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [94, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [100, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [100, 101, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [100, 101, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [100, 101, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [100, 101, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [100, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [100, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [100, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [100, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [109, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [109, 110, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [109, 110, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [109, 110, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [109, 110, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [109, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [109, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [109, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [109, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [114, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [114, 115, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [114, 115, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [114, 115, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [114, 115, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [114, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [114, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [114, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [114, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [149, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [149, 150, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [149, 150, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [149, 150, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [149, 150, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [149, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [149, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [149, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [149, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [306, _, _], [], 71, 0, 0
```

Figure A.4: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
N, 2, 3, [70,71,72], [306, 307, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [306, 307, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [306, 307, _], [], 71, 1, 0
C, 3, 2, [70,71,72], [306, 307, _], 72
N, 4, 3, [70,71,72], [306, 307, _], [], 72, 0, 0
M, 5, 11, [70,71,72], [306, 307, 308], PC, 71, 0, 0
U, 5, 14, [70,71,72], [306, 307, 308], 0, 70, 0, 0
U, 5, 14, [70,71,72], [306, 307, 308], 4, 70, 0, 0
U, 5, 13, [70,71,72], [306, 307, 308], CPAGE, 72, 1, 0
Hw/var "IR" mapped to inst/var "CPAGE".
H, 6, 18, [70,71,72], [306, 307, 308], 0, 70, 0, 0
H, 6, 21, [70,71,72], [306, 307, 308], 0, 70, 0, 0
H, 6, 22, [70,71,72], [306, 307, 308], 0, 70, 0, 0
H, 6, 23, [70,71,72], [306, 307, 308], 0, 70, 0, 0
U, 5, 15, [70,71,72], [306, 307, 308], CPAGE, 72, 1, 0
U, 5, 24, [70,71,72], [306, 307, 308], 4, 70, 0, 0
U, 5, 24, [70,71,72], [306, 307, 308], 0, 70, 0, 0
N, 4, 4, [70,71,72], [306, 307, _], [], 72, 0, 0
N, 4, 5, [70,71,72], [306, 307, _], [], 72, 0, 0
N, 4, 6, [70,71,72], [306, 307, _], [], 72, 0, 0
N, 4, 7, [70,71,72], [306, 307, _], [], 72, 0, 0
N, 2, 7, [70,71,72], [306, 307, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [306, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [306, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [306, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [306, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [341, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [341, 342, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [341, 342, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [341, 342, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [341, 342, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [341, _, _], [], 71, 0,
N, 1, 5, [70,71,72], [341, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [341, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [341, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [689, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [689, 690, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [689, 690, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [689, 690, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [689, 690, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [689, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [689, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [689, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [689, _, _], [], 71, 0, 0
```

Figure A.5: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

array (see Figure 7.2). In this particular microengine, only 8 (numbered 0 through 7) of the possible 16 register array locations are used, so that the locations 8 through 15 are ignored (by zeroing the previously mentioned high-order bit of the register address). This technique was used to make the microstore one bit narrower.

Processing on the first candidate hardware node fails (shown near the top of Figure A.4) for two reasons. First, the output of bit extraction goes to an ISPS concatenation operator "@" instead of being assigned to a variable. Secondly, the hardware and instruction-set variable bitfields are of unequal widths (5 versus 3).

The next candidate is node 100 from the expression "'0 @ ALUport<2:0>" of the hardware description. This expression similarly controls the other port (called B) of the dual-ported memory array and fails processing for the same reasons as the previous candidate (as shown in Figure A.4).

The next hardware-node candidates are 109 and 114. These are part of the expressions "uIR<3:0>" and "'000 @ nxtAddr<8:0>", which extract part of the microinstruction register to pass to the AM2910 microsequencer chip. These matches are similarly unappropriate, and fail in Figure A.4.

Hardware node candidate 149, from the expression "'0 @ CCsel<4:0>", possesses the correct variable bit width but does not flow into an assignment statement. This hardware fragment extracts the selected code-jump condition

from the microinstruction word. This match is also unappropriate, failing near the bottom of Figure A.4.

A.3.2 An insufficiently close structural match

The hardware fragment "IR=MB<0:4>", which uses node 306, is a very close structural match to the current instruction-set fragment "cpage=PC<0:4>". Many of its arcs are matched by the inference engine (shown in the bottom of Figure A.4 and the top half of Figure A.5). The bit widths are compatible, and the match goes so far as to tentatively map the instruction-set description assignment-destination variable "cpage" to "IR" of the hardware (shown one-third down from the top in Figure A.5). As part of the instruction fetch, the hardware statement stores the five high-order bits of the memory-buffer register into the instruction register.

Consider the previous hardware to instruction-set mappings: "MBR":"i", "MP":"M", "MAR":"PC", "MBR=MP[MAR]":"i=M[PC]" (these are displayed near the bottom of Figure A.2 as part of the "varmap" record) and current and tentative mapping "IR=MB<0:4>":"cpage=PC<0:4>". Note that "MB" and "MBR" are defined as equivalent in the variable definition section of the hardware description. These previous and proposed mappings are inconsistent for a subtle reason.

The instruction-set fragments "i=M[PC]" and "cpage=PC<0:4>" require the object of bitfield extraction "PC" in the second fragment to process the same array-access index value as in the first fragment. Note that "PC" is on the right

side of the two equal signs for the two instruction-set fragments. The corresponding hardware fragments ("MBR=MP[MAR]" and "IR=MB<0:4>") require the bitfield-extraction object "MB" in the second fragment to match the array-access output value of the first fragment. Note that "MB" (or equivalently "MBR", since the hardware description declares the variables "MB" and "MBR" to be equivalent) is on both the left and right sides of the two equal signs. These mappings seem to be inconsistent.

These variables could be consistent only when "i" and "PC" of the instruction set ("MBR" and "MAR" of hardware) are of the same value. Since data-dependency analysis determines that this is not true, the entire mapping is invalid. The remainder of the first half of Figure A.5 shows the failure of this proposed mapping.

A.3.3 More unsuccessful search

Hardware-node candidate 341, from the expression "Full@enable<2:0>", does not match the correct bit width or flow into an assignment statement, and processing fails toward the bottom of Figure A.5. This hardware fragment combines bits for the microsequencer.

The hardware-node candidates 689, 694, 700, 705, 713, 717, 723, and 727, shown in Figures A.6 and A.7 originate from the input parameters to subroutines that calculate carry ("c.out") and overflow ("OVR") in the ALU logic operations "exclusive or" and "exclusive nor." Hardware-node candidates 742,

```
N, 1, 3, [70,71,72], [694, _, _], [], 71, 0,
N, 2, 3, [70,71,72], [694, 695, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [694, 695, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [694, 695, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [694, 695, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [694, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [694, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [694, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [694, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [700, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [700, 701, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [700, 701, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [700, 701, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [700, 701, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [700, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [700, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [700, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [700, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [705, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [705, 706, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [705, 706, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [705, 706, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [705, 706, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [705, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [705, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [705, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [705, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [713, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [713, 714, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [713, 714, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [713, 714, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [713, 714, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [713, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [713, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [713, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [713, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [717, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [717, 718, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [717, 718, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [717, 718, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [717, 718, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [717, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [717, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [717, _, _], [], 71, 0, 0
```

Figure A.6: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
N, 1, 7, [70,71,72], [717, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [723, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [723, 724, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [723, 724, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [723, 724, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [723, 724, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [723, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [723, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [723, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [723, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [727, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [727, 728, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [727, 728, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [727, 728, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [727, 728, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [727, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [727, _, _], [], 71, 0,
N, 1, 6, [70,71,72], [727, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [727, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [742, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [742, 743, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [742, 743, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [742, 743, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [742, 743, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [742, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [742, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [742, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [742, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [768, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [768, 769, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [768, 769, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [768, 769, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [768, 769, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [768, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [768, _, _], [], 71, 0, 0
N, 1, 6, [70,71,72], [768, _, _], [], 71, 0, 0
N, 1, 7, [70,71,72], [768, _, _], [], 71, 0, 0
N, 1, 3, [70,71,72], [783, _, _], [], 71, 0, 0
N, 2, 3, [70,71,72], [783, 784, _], [], 71, 1, 0
N, 2, 4, [70,71,72], [783, 784, _], [], 71, 1, 0
N, 2, 6, [70,71,72], [783, 784, _], [], 71, 1, 0
N, 2, 7, [70,71,72], [783, 784, _], [], 71, 1, 0
N, 1, 4, [70,71,72], [783, _, _], [], 71, 0, 0
N, 1, 5, [70,71,72], [783, _, _], [], 71, 0, 0
```

Figure A.7: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

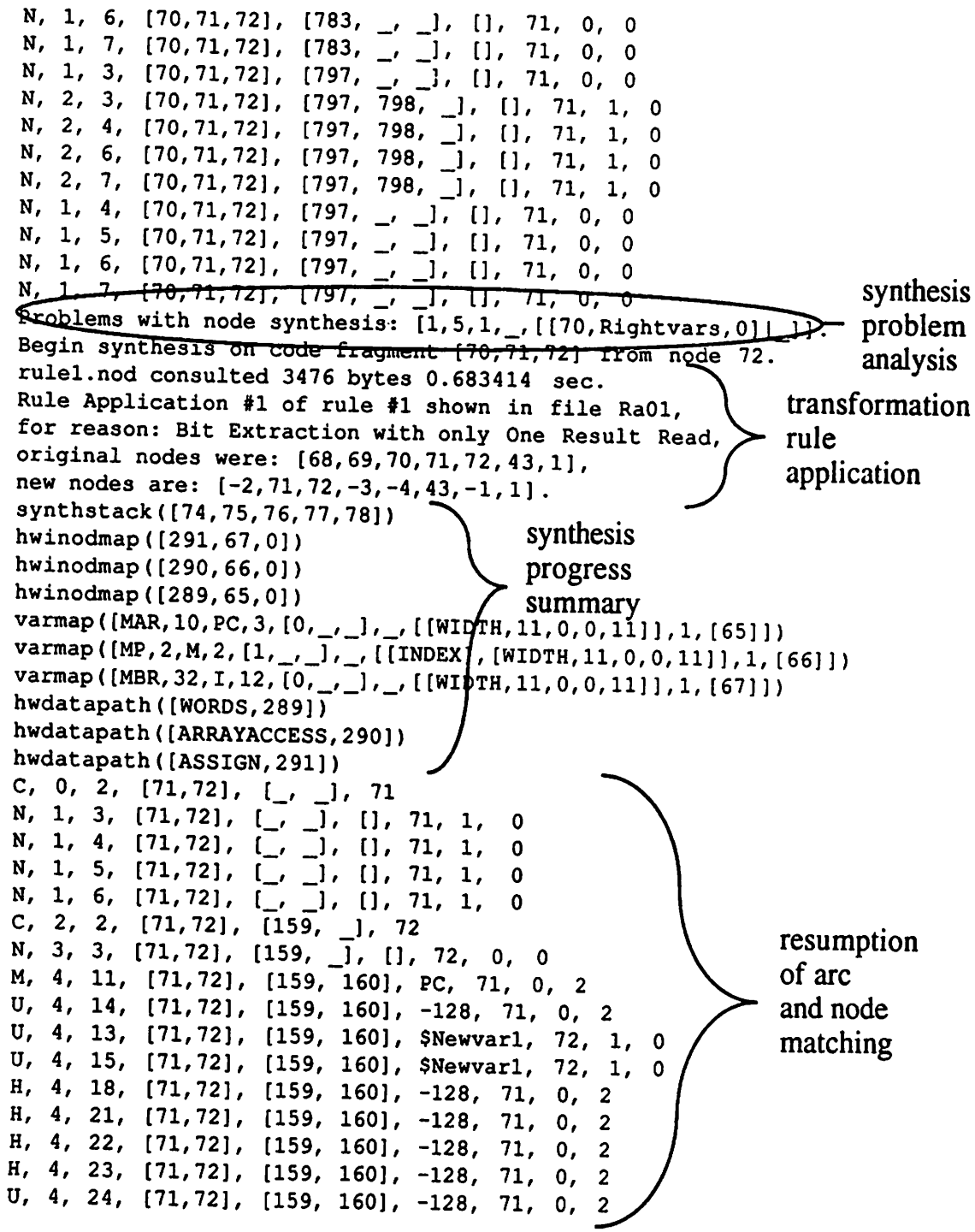


Figure A.8: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

768, 783, and 797 (see [SIEBN82] page 185 for these) shown in Figures A.7 and A.8 are from the body of these subroutines. All of these bitfield extractions fail because they do not provide the correct width bitfields.

A.3.4 Failure analysis after exhaustive search

The synthesis trace of Figure A.8 (about one third from the top) shows that after performing exhaustive search on all possible hardware bitfield-extraction nodes, each candidate node has been rejected. The trace line "`Problems with node synthesis: [1,5,1,_,[[70,Rightvars,0]|_]]`." in Figure A.8 means that in the best partial candidate-set match (candidate nodes 306, 307, and 308), five arcs were matched correctly (indicated by the second integer in the trace line). It also shows that the problem in matching the sixth arc was within instruction node 70 (the fourth integer), since the right variable "`Rightvars`" cannot be accessed. Previous discussion described why the 306, 307, and 308 hardware-node candidate set failed.

A.3.5 Transformation of the instruction-set fragment

When triggered by search failure, UMS attempts to synthesize the same instruction-set description fragment after trying to transform the problem-causing instruction-set node (in this case 70) into something else with a semantics-preserving transformation. UMS sequentially tries each transformation within its knowledge base. Before it is performed, each transformation is checked to determine if it would affect the problem-causing

```
/*                                                                    */
/* Bit Extraction with later Concatenation as Logical Anding Rule     */
/*                                                                    */
/* A<M:N> ==> A and -1<M:N>                                           */
/*                                                                    */
        rule('Bit Extraction with only One Result Read',
             Ruletype,
        [
        rulenode('CONSTANT',
                        [],
                        [[M,[],[R3]]],
                        _,
                        _,
                        R1),
        rulenode('CONSTANT',
                        [],
                        [[N,[],[R3]]],
                        _,
                        _,
                        R2),
        rulenode('BITFIELD',
                        [
                         [M,[],[R1]],
                         [N,[],[R2]]
                        ],
                        [[[],[],[R4]]],
                        _,
                        _,
                        R3),
        rulenode('EXTRACTFIELD',
                        [
                         [Var1,Varlatt,BitSrc1],
                         [[],[],[R3]]
                        ],
                        [[[],[],[R5]]],
                        _,
                        _,
                        R4),
```

Figure A.9: A UMS rule-input specification (see text).

```
rulenode('ASSIGN',
                    [
                     [_,[['CHOICE',_,Ch1]],[Cho1]],
                     [[],[],[R4]]
                    ],
                    [[Var2,Aatt,[R8]]],
                    _,
                    _,
                    R5),
rulenode('CONCAT',
                    [
                     [_,[['CHOICE',_,Ch2]],[Cho2]],
                     [Var2,[],[R5]],
                     [Var3,[],[R9]]
                    ],
                    [[[],[],R10]],
                    _,
                    (
                     notequal(Var2,0),
                     makebits(instnode,R4,Var1,M,N,Mask1),
                     fcntrl(instnode,BitDest,[Ch2,Ch3]),
                     findcontrol(Ruletype,R8,Cntrl,_),
                     findvardef(Ruletype,Var1,VD1,Cntrl),
                     findvardef(Ruletype,Var3,VD2,Cntrl),
                     vardefatt(Ruletype,VD1,'WIDTH',[M,NN]),
                     findwidesib(Ruletype,M,NN,VD2,Var33,_),
                     vardefatt(Ruletype,VD2,'WIDTH',[W1,W2]),
                     Remain is NN-N, OtherWD is W1-W2,
                     abs(OtherWD,OtherW),
                     abs(Remain,Rem), Rem is OtherW+1,
                    newvardef(Var4,Var1,R4,_,_,NewN1,NewN2,N2),
                     vardefatt(Ruletype,VD2,'PARTOF',
                               [_,[['WIDTH',_,[RW1,RW2]]]]),
                     makebits(instnode,R4,Var1,RW1,RW2,Mask2)
                    ),
                    R8),
rulenode('VARDEF',
                    _,
                    _,
                    _,
                    _,
                    N2)
        ],
```

Figure A.10: Continued UMS rule-input specification (see text).

```
[
rulenode('CONSTANT',
                  [],
                  [[Mask1,[],[R4]]],
                  _,
                  _,
                  NewR1),
rulenode('AND',
                  [
                   [_,[['CHOICE',_,Ch1]],[Cho1]],
                   [Var1,Var1att,BitSrc1],
                   [Mask1,[],[NewR1]]
                  ],
                  [[[],[],[R5]]],
                  _,
                  _,
                  R4),
rulenode('ASSIGN',
                  [
                   [_,[['CHOICE',_,Ch1]],[Cho1]],
                   [[],[],[R4]]
                  ],
                  [[Var4,Aatt,[R8]]],
                  _,
                  _,
                  R5),
rulenode('CONSTANT',
                  [],
                  [[Mask2,[],[R6]]],
                  _,
                  _,
                  NewR3),
rulenode('AND',
                  [
                   [_,[['CHOICE',_,Ch2]],[Cho2]],
                   [Var33,[],[R9]],
                   [Mask2,[],[NewR3]]
                  ],
                  [[[],[],[R8]]],
                  _,
                  _,
                  R6),
```

Figure A.11: A UMS rule-output specification (see text).

```
rulenode('OR',
                    [
                     [_,[['CHOICE',_,Ch2]],[Cho2]],
                     [Var4,[],[R5]],
                     [[],[],[R6]]
                    ],
                    [[[],[],R10]],
                    _,
                    _,
                    R8),
NewN1,
NewN2
],
1
).
```

Figure A.12: Continued UMS rule-output specification (see text).

node. If so, then the transformation is performed and synthesis is reattempted as if the transformed code was the original. Thus, UMS performs breadth-first search for one application of any transformation. However, if the transformed fragment cannot be synthesized, the code is restored to its original state and another transformation is attempted. After attempting to perform all transformations in the knowledge base, UMS tries sets of multiple transformations: first two, then three, and so forth.

Figures A.9 through A.12 show the source code for the first UMS transformation applied in this example, which is also shown in the synthesis traced in the middle of Figure A.8 (a similar transformation was discussed in the example of Chapter I). This rule describes an alternative technique to using ISPS angle brackets for bit extraction. This alternative performs an AND operation on the value with the appropriate constant, which contains 1's where

the desired field is located, 0's elsewhere. The application of this rule is constrained to where the field does not have to be realigned within its destination, which would require shifting. The rule is further constrained to where the desired value is used only once by a bitfield concatenation operation. In this case, the undesired parts of the required bitfield are masked out by an AND operation, the unwanted bits of the destination masked out with a similar AND operation, and the two values merged with an OR operation. This transforms "`cpage=PC<0:4> next eadd=cpage@pa`" into "`$Newvar1 = (PC and '111110000000) next eadd = ($Newvar1 or (i and '000001111111))`", where "`pa`" had been specified in the variable definitions to be equivalent to "`i<5:11>`". This is the same microprogramming trick shown in Figure 1.4.

Figures A.9 and A.12 include the set of nodes that specify the pattern and include variable-definition nodes (called "VARDEF") so that size and bit-numbering information may be accessed easily. Some Prolog code added to the rule (this is a form of procedural attachment) performs the size and bit-numbering tests as well as making the appropriate constant.

A.3.6 The transformation trace

Figures A.13 through A.16 show a trace of the transformation application stored on a disk by UMS for later analysis. UMS can use this trace to restore the instruction-set nodes to their original state. Figure A.13 and the first half of Figure A.14 show the nodes of concern before applying the

```
/* Rule Application #1 of rule #1 */
/* Original Nodes: [68,69,70,71,72,43,1] */
ruleapplication(1,1,[68,69,70,71,72,43,1],[
        instnode('CONSTANT',
                        [],
                        [[0,[],[70]]],
                        _,
                        _,
                        68),
        instnode('CONSTANT',
                        [],
                        [[4,[],[70]]],
                        _,
                        _,
                        69),
        instnode('BITFIELD',
                        [
                         [0,[],[68]],
                         [4,[],[69]]
                        ],
                        [[[],[],[71]]],
                        _,
                        _,
                        70),
        instnode('EXTRACTFIELD',
                        [
                         ['PC',[['UndefRead',_,[3]]],
                               [3,75,84,114,127,129,193]],
                         [[],[],[70]]
                        ],
                        [[[],[],[72]]],
                        _,
                        _,
                        71),
        instnode('ASSIGN',
                        [
                         [_,[['CHOICE',_,'REPEAT']],[64]],
                         [[],[],[71]]
                        ],
                        [['CPAGE',[['WIDTH',_,[0,4]]],[43]]],
                        _,
                        _,
                        72),
```

Figure A.13: A UMS rule-application trace (see text).

```
        instnode('CONCAT',
                        [
                         [_,[['CHOICE',_,1]],[39]],
                         ['CPAGE',[],[72]],
                         ['PA',[],[67]]
                        ],
                        [[[],[],[44]]],
                        _,
                        _,
                        43),
        instnode('VARDEF',
                        [[_,[['CHOICE',_,'SUBRDEF'],
                         ['IndexReadWrite',_,[0,0,0]],
                         ['ParentVar',_,['PDP8',1,[1]]]],[0,1]]],
                        [['PDP8',[['WIDTH',_,[1,1]],
                         ['SUBCODE',_,[2,3,4,5,6,7,8,9,10,11,12,
                          13,14,15,16,17,18,19,20,21,22,23,24,
                          25,26,27,28,29,30,31,32,33,34,35,36,
                          37,38,63,85,132,144,145,194]]],[]]],
                        _,
                        _,
                        1)
],
/* New Nodes: [-2,71,72,-3,-4,43,-1,1] */
[-2,71,72,-3,-4,43,-1,1],[
        instnode('CONSTANT',
                        [],
                        [[-128,[],[71]]],
                        _,
                        _,
                        -2),
        instnode('AND',
                        [
                         [_,[['CHOICE',_,'REPEAT']],[64]],
                         ['PC',[['UndefRead',_,[3]]],
                               [3,75,84,114,127,129,193]],
                         [-128,[],[-2]]
                        ],
                        [[[],[],[72]]],
                        _,
                        _,
                        71),
```

Figure A.14: Continued UMS rule-application trace (see text).

```
instnode('ASSIGN',
                    [
                     [_,[['CHOICE',_,'REPEAT']],[64]],
                     [[],[],[71]]
                    ],
                    [['$Newvar1',[['WIDTH',_,[0,4]]],[43]]],
                    _,
                    _,
                    72),
instnode('CONSTANT',
                    [],
                    [[127,[],[-4]]],
                    _,
                    _,
                    -3),
instnode('AND',
                    [
                     [_,[['CHOICE',_,1]],[39]],
                     ['I',[],[67]],
                     [127,[],[-3]]
                    ],
                    [[[],[],[43]]],
                    _,
                    _,
                    -4),
instnode('OR',
                    [
                     [_,[['CHOICE',_,1]],[39]],
                     ['$Newvar1',[],[72]],
                     [[],[],[-4]]
                    ],
                    [[[],[],[44]]],
                    _,
                    _,
                    43),
```

Figure A.15: Continued UMS rule-application trace (see text).

```
        instnode('VARDEF',
                        [[_,[['CHOICE',_,'SUBRDEF'],
                             ['IndexReadWrite',_,[0,1,1]],
                             ['ParentVar',_,['$Newvar1',-1,
                              [1,1,1,1,1,1,1,1,1,1,1,1]]]],[1]]],
                        [['$Newvar1',[['PARTOF',_,[]],
                                      ['WIDTH',_,[0,11]]],[]]],
                        _,
                        _,
                        -1),
        instnode('VARDEF',
                        [[_,[['CHOICE',_,'SUBRDEF'],
                             ['IndexReadWrite',_,[0,0,0]],
                             ['ParentVar',_,['PDP8',1,[1]]]],[0,1]]],
                        [['PDP8',[['WIDTH',_,[1,1]],
                             ['SUBCODE',_,[-1,2,3,4,5,6,7,8,9,10,11,12,
                              13,14,15,16,17,18,19,20,21,22,23,24,25,
                              26,27,28,29,30,31,32,33,34,35,36,37,38,
                              63,85,132,144,145,194]]],[]]],
                        _,
                        _,
                        1)
]).
```

Figure A.16: Continued UMS rule-application trace (see text).

transformation. The second half of Figure A.14 and all of Figures A.15 and A.16 show their replacement nodes. As shown in Figures A.14 through A.16, when additional nodes are created during application of a transformation, the nodes are labeled with negative node numbers.

A.3.7 Resumption of synthesis

In an effort to match the new instruction-set description node number 71 (the first AND) with the hardware description, UMS now looks for a corresponding hardware AND operator node instead of the previously unsuccessful bit-extraction node. The first one found is node 159 shown in the trace of Figure A.8. This node corresponds in the hardware description to the fragment "`interrupt.request AND interrupt.enable`", which processes the values of the two microengine state bits. These two one-bit-wide hardware registers do not match the twelve-bit-wide operands created by the transformation, so the match fails.

The next hardware candidate is node 368 (as shown in Figure A.17), which comes from the fragment "`fail=(not CCEN) and CC`". This similarly fails, as shown in the beginning of Figure A.18.

A.3.8 Successful node match of modified code

The next candidate for matching with instruction-set description node 71 from "`$Newvar1 = PC and '111110000000`", as shown in Figure A.18, is the hardware-description AND operator node 666 from "`alu = R and S`", which is

```
M, 4, 11, [71,72], [159, 160], PC, 71, 0, 1
U, 4, 14, [71,72], [159, 160], -128, 71, 0, 1
U, 4, 13, [71,72], [159, 160], $Newvar1, 72, 1, 0
U, 4, 15, [71,72], [159, 160], $Newvar1, 72, 1, 0
H, 4, 18, [71,72], [159, 160], PC, 71, 0, 1
H, 4, 21, [71,72], [159, 160], PC, 71, 0, 1
H, 4, 22, [71,72], [159, 160], PC, 71, 0, 1
H, 4, 23, [71,72], [159, 160], PC, 71, 0, 1
U, 4, 24, [71,72], [159, 160], -128, 71, 0, 1
M, 4, 11, [71,72], [159, 160], PC, 71, 0, 1
U, 4, 14, [71,72], [159, 160], -128, 71, 0, 1
U, 4, 13, [71,72], [159, 160], $Newvar1, 72, 1, 0
U, 4, 15, [71,72], [159, 160], $Newvar1, 72, 1, 0
H, 4, 18, [71,72], [159, 160], PC, 71, 0, 1
H, 4, 21, [71,72], [159, 160], PC, 71, 0, 1
H, 4, 22, [71,72], [159, 160], PC, 71, 0, 1
U, 4, 24, [71,72], [159, 160], -128, 71, 0, 1
N, 3, 4, [71,72], [159, _], [], 72, 0, 0
N, 3, 5, [71,72], [159, _], [], 72, 0, 0
N, 3, 6, [71,72], [159, _], [], 72, 0, 0
N, 3, 7, [71,72], [159, _], [], 72, 0, 0
N, 3, 8, [71,72], [159, _], [], 72, 0, 0
C, 2, 2, [71,72], [368, _], 72
N, 3, 3, [71,72], [368, _], [], 72, 0, 0
M, 4, 11, [71,72], [368, 369], PC, 71, 0, 2
U, 4, 14, [71,72], [368, 369], -128, 71, 0, 2
U, 4, 13, [71,72], [368, 369], $Newvar1, 72, 1, 0
U, 4, 15, [71,72], [368, 369], $Newvar1, 72, 1, 0
H, 4, 18, [71,72], [368, 369], -128, 71, 0, 2
H, 4, 21, [71,72], [368, 369], -128, 71, 0, 2
H, 4, 22, [71,72], [368, 369], -128, 71, 0, 2
H, 4, 23, [71,72], [368, 369], -128, 71, 0, 2
U, 4, 24, [71,72], [368, 369], -128, 71, 0, 2
M, 4, 11, [71,72], [368, 369], PC, 71, 0, 1
U, 4, 14, [71,72], [368, 369], -128, 71, 0, 1
U, 4, 13, [71,72], [368, 369], $Newvar1, 72, 1, 0
U, 4, 15, [71,72], [368, 369], $Newvar1, 72, 1, 0
H, 4, 18, [71,72], [368, 369], PC, 71, 0, 1
H, 4, 21, [71,72], [368, 369], PC, 71, 0,
H, 4, 22, [71,72], [368, 369], PC, 71, 0, 1
H, 4, 23, [71,72], [368, 369], PC, 71, 0, 1
U, 4, 24, [71,72], [368, 369], -128, 71, 0, 1
M, 4, 11, [71,72], [368, 369], PC, 71, 0, 1
U, 4, 14, [71,72], [368, 369], -128, 71, 0, 1
```

Figure A.17: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
U, 4, 13, [71,72], [368, 369], $Newvar1, 72, 1, 0
U, 4, 15, [71,72], [368, 369], $Newvar1, 72, 1, 0
H, 4, 18, [71,72], [368, 369], PC, 71, 0, 1
H, 4, 21, [71,72], [368, 369], PC, 71, 0, 1
H, 4, 22, [71,72], [368, 369], PC, 71, 0,
U, 4, 24, [71,72], [368, 369], -128, 71, 0, 1
N, 3, 4, [71,72], [368, _], [], 72, 0, 0
N, 3, 5, [71,72], [368, _], [], 72, 0,
N, 3, 6, [71,72], [368, _], [], 72, 0,
N, 3, 7, [71,72], [368, _], [], 72, 0, 0
N, 3, 8, [71,72], [368, _], [], 72, 0, 0
C, 2, 2, [71,72], [666, _], 72
N, 3, 3, [71,72], [666, _], [], 72, 0, 0
M, 4, 11, [71,72], [666, 667], PC, 71, 0, 2
U, 4, 14, [71,72], [666, 667], -128, 71, 0, 2
U, 5, 13, [71,72], [666, 667], $Newvar1, 72, 1, 0
U, 5, 15, [71,72], [666, 667], $Newvar1, 72, 1, 0
H, 5, 18, [71,72], [666, 667], PC, 71, 0,
H, 5, 21, [71,72], [666, 667], PC, 71, 0, 2
ALU.OUT _
TEMP.AM2901 _
ALU.LSB _
ALU.MSB _
AM2901 _
OUT _
CN4 _
C.OUT _
0 _
Y _
F _
A.LATCH _
H, 6, 19, [71,72], [666, 667], $Newvar1, 72, 1, 0
Hw/var "F" mapped to inst/var "$Newvar1".
Begin synthesis on code fragment [74,75] from node 74.
history([instnode,70,[68,69,70,71,72,43,1],
                    [-2,71,72,-3,-4,43,-1,1],1,1])
synthstack([75,76,77,78])
hwinodmap([666,71,2])
hwinodmap([667,72,0])
hwinodmap([291,67,0])
hwinodmap([290,66,0])
hwinodmap([289,65,0])
varmap([A.LATCH,522,PC,3,[0,_,_],_,[[WIDTH,11,0,0,11]],1,[65,71]])
varmap([MP,2,M,2,[1,_,_],_,[[INDEX],[WIDTH,11,0,0,11]],1,[66]])
```

movement log of "PC" upstream

record of previously moved mapping of "PC" to "A.LATCH"

Figure A.18: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

part of the microengine ALU. Structurally, these two code fragments are alike, and they are an appropriate synthesis match.

A.3.9 Synthesis-triggered variable remapping

Note that in early synthesis, the instruction-set variable "PC" was mapped to the hardware memory-address register "MAR" (shown in the bottom right of Figure 7.2). For the current candidate node set to be accepted, the value of "PC" must also be accessible as an input operand to the expression "$Newvar1 = PC and '111110000000". This means that it must be capable of being an input to the ALU.

The variable names in the middle of Figure A.18 list alternate candidate hardware bindings for "PC" as it is moved "upstream" from the memory-address register (some of these names are shown in the microengine dataflow diagram of Figure 7.2 and Figure 1.5). Specifically, "ALU.OUT" and "TEMP.AM2901" from Figure A.18 are descriptive variables for the AM2901 output. "ALU.LSB", "ALU.MSB", and "C.OUT" are the least- and most-significant bits and the carry bit, respectively, of the ALU output and are inappropriate for "PC" because they are only one bit wide. "OUT" and "CN4" are inappropriate because they are not twelve bits wide. "0", the zero constant input of the ALU, is inappropriate because it cannot be written to (dataflow analysis has determined that "PC" must be able to be written to). "F" and "Y" are not appropriate because they are both read and written from/to a section of the hardware bus, and therefore cannot

store state. The variable checked for "PC", "A.LATCH", is remapped to this part of the hardware. When the arc for "$Newvar1" is processed, it is the variable's first application, and simple-minded processing maps it to "F", although this was rejected earlier for "PC" at a much smaller computational cost. "F" is the output of the ALU. When more is known about how "$Newvar1" is used in the microprogram, later processing will remap it to something reasonable.

The hardware ALU output is to the 13-bit value "alu", where "$Newvar1" of the instruction-set description expects a 12-bit output. This extra bit in the hardware description is the ALU carry bit. The match is close, but not close enough. A heuristic within the UMS inference engine is triggered when the hardware-match variable is wider than the instruction-set description requires. The reasoning is that if the lower-order bits (of the width required by the instruction-set description) are available somewhere "downstream" on the hardware bus, then the instruction-set variable can be bound at that point. UMS chooses the closest viable hardware variable ("downstream" from "alu"). Now, "F" can be bound to the instruction-set variable "$Newvar1", shown in Figure A.19 as "varmap([F,519,$Newvar1,-1,[0,1,1],_,[[WIDTH,11,0,0,11]], 2,[72]])".

```
varmap([MBR,32,I,12,[0,_,_],_,[[WIDTH,11,0,0,11]],1,[67]])
varmap([F,519,$Newvar1,-1,[0,1,1],_,[[WIDTH,11,0,0,11]],2,[72]])
constants([MASK,24,-128,0])
hwdatapath([WORDS,289])
hwdatapath([ARRAYACCESS,290])
hwdatapath([ASSIGN,291])
hwdatapath([AND,666])
hwdatapath([ASSIGN,667])
constreg([MASK,24])
busnode([S,560,_])
busnode([ALU.OUT,104,_])
busnode([TEMP.AM2901,-104,_])
busnode([TEMP.AM2901,104,_])
busnode([AM2901,539,_])
busnode([OUT,573,_])
busnode([OUT,-573,_])
busnode([Y,573,_])
busnode([R,561,_])
busnode([DIN,83,_])
C, 0, 2, [74,75], [_, _], 74
N, 1, 3, [74,75], [_, _], [], 74, 1, 0
N, 1, 4, [74,75], [_, _], [], 74, 1,
N, 1, 5, [74,75], [_, _], [], 74, 1,
N, 1, 6, [74,75], [_, _], [], 74, 1, 0
C, 2, 2, [74,75], [346, _], 75
N, 3, 3, [74,75], [346, _], [], 75, 0, 0
M, 4, 11, [74,75], [346, 347], PC, 74, 0, 2
M, 4, 11, [74,75], [346, 347], PC, 75, 1, 0
U, 4, 14, [74,75], [346, 347], 1, 74, 0, 2
H, 4, 18, [74,75], [346, 347], 1, 74, 0, 2
H, 4, 21, [74,75], [346, 347], 1, 74, 0, 2
H, 4, 22, [74,75], [346, 347], 1, 74, 0, 2
H, 4, 23, [74,75], [346, 347], 1, 74, 0, 2
U, 4, 24, [74,75], [346, 347], 1, 74, 0, 2
M, 4, 11, [74,75], [346, 347], PC, 74, 0, 1
M, 4, 11, [74,75], [346, 347], PC, 75, 1, 0
U, 4, 14, [74,75], [346, 347], 1, 74, 0, 1
H, 4, 18, [74,75], [346, 347], PC, 74, 0, 1
H, 4, 21, [74,75], [346, 347], PC, 74, 0, 1
ALU.LSB _
ALU.MSB _
CN4 _
C.OUT _
0 _
```

Figure A.19: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
F _
A.LATCH _
RAM _
OVR _
OVR67 _
H, 4, 22, [74,75], [346, 347], PC, 74, 0, 1
H, 4, 23, [74,75], [346, 347], PC, 74, 0, 1
U, 4, 24, [74,75], [346, 347], 1, 74, 0, 1
M, 4, 11, [74,75], [346, 347], PC, 74, 0, 1
M, 4, 11, [74,75], [346, 347], PC, 75, 1, 0
U, 4, 14, [74,75], [346, 347], 1, 74, 0, 1
H, 4, 18, [74,75], [346, 347], PC, 74, 0, 1
H, 4, 21, [74,75], [346, 347], PC, 74, 0, 1
H, 4, 22, [74,75], [346, 347], PC, 74, 0, 1
U, 4, 24, [74,75], [346, 347], 1, 74, 0, 1
N, 3, 4, [74,75], [346, _], [], 75, 0, 0
N, 3, 5, [74,75], [346, _], [], 75, 0, 0
N, 3, 6, [74,75], [346, _], [], 75, 0, 0
N, 3, 7, [74,75], [346, _], [], 75, 0, 0
C, 2, 2, [74,75], [489, _], 75
N, 3, 3, [74,75], [489, _], [], 75, 0, 0
M, 4, 11, [74,75], [489, 490], PC, 74, 0, 2
M, 4, 11, [74,75], [489, 490], PC, 75, 1, 0
U, 4, 14, [74,75], [489, 490], 1, 74, 0, 2
H, 4, 18, [74,75], [489, 490], 1, 74, 0, 2
H, 4, 21, [74,75], [489, 490], 1, 74, 0, 2
H, 4, 22, [74,75], [489, 490], 1, 74, 0, 2
H, 4, 23, [74,75], [489, 490], 1, 74, 0, 2
U, 4, 24, [74,75], [489, 490], 1, 74, 0, 2
M, 4, 11, [74,75], [489, 490], PC, 74, 0, 1
M, 4, 11, [74,75], [489, 490], PC, 75, 1, 0
U, 4, 14, [74,75], [489, 490], 1, 74, 0, 1
H, 4, 18, [74,75], [489, 490], PC, 74, 0, 1
H, 4, 21, [74,75], [489, 490], PC, 74, 0, 1
H, 4, 22, [74,75], [489, 490], PC, 74, 0, 1
H, 4, 23, [74,75], [489, 490], PC, 74, 0, 1
U, 4, 24, [74,75], [489, 490], 1, 74, 0, 1
M, 4, 11, [74,75], [489, 490], PC, 74, 0, 1
M, 4, 11, [74,75], [489, 490], PC, 75, 1, 0
U, 4, 14, [74,75], [489, 490], 1, 74, 0, 1
H, 4, 18, [74,75], [489, 490], PC, 74, 0, 1
H, 4, 21, [74,75], [489, 490], PC, 74, 0, 1
H, 4, 22, [74,75], [489, 490], PC, 74, 0, 1
U, 4, 24, [74,75], [489, 490], 1, 74, 0, 1
```

Figure A.20: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
N, 3, 4, [74,75], [489, _], [], 75, 0, 0
N, 3, 5, [74,75], [489, _], [], 75, 0, 0
N, 3, 6, [74,75], [489, _], [], 75, 0, 0
N, 3, 7, [74,75], [489, _], [], 75, 0, 0
C, 2, 2, [74,75], [614, _], 75
N, 3, 3, [74,75], [614, _], [], 75, 0, 0
M, 4, 11, [74,75], [614, 615], PC, 74, 0, 2
M, 4, 11, [74,75], [614, 615], PC, 75, 1, 0
U, 4, 14, [74,75], [614, 615], 1, 74, 0, 2
H, 4, 18, [74,75], [614, 615], 1, 74, 0, 2
H, 4, 21, [74,75], [614, 615], 1, 74, 0, 2
H, 4, 22, [74,75], [614, 615], 1, 74, 0, 2
H, 4, 23, [74,75], [614, 615], 1, 74, 0, 2
U, 4, 24, [74,75], [614, 615], 1, 74, 0, 2
M, 4, 11, [74,75], [614, 615], PC, 74, 0, 1
M, 4, 11, [74,75], [614, 615], PC, 75, 1, 0
U, 4, 14, [74,75], [614, 615], 1, 74, 0, 1
H, 5, 18, [74,75], [614, 615], PC, 74, 0, 1
H, 5, 21, [74,75], [614, 615], PC, 74, 0, 1
H, 5, 22, [74,75], [614, 615], PC, 74, 0, 1
H, 5, 23, [74,75], [614, 615], PC, 74, 0, 1
H, 4, 18, [74,75], [614, 615], PC, 74, 0, 1
H, 4, 21, [74,75], [614, 615], PC, 74, 0, 1
H, 4, 22, [74,75], [614, 615], PC, 74, 0, 1
U, 4, 24, [74,75], [614, 615], 1, 74, 0, 1
M, 4, 11, [74,75], [614, 615], PC, 74, 0, 1
M, 4, 11, [74,75], [614, 615], PC, 75, 1, 0
U, 4, 14, [74,75], [614, 615], 1, 74, 0, 1
H, 4, 18, [74,75], [614, 615], PC, 74, 0, 1
H, 4, 21, [74,75], [614, 615], PC, 74, 0, 1
H, 4, 22, [74,75], [614, 615], PC, 74, 0, 1
U, 4, 24, [74,75], [614, 615], 1, 74, 0, 1
N, 3, 4, [74,75], [614, _], [], 75, 0, 0
N, 3, 5, [74,75], [614, _], [], 75, 0, 0
N, 3, 6, [74,75], [614, _], [], 75, 0, 0
N, 3, 7, [74,75], [614, _], [], 75, 0, 0
N, 1, 7, [74,75], [_, _], [], 74, 1, 0
Problems with node synthesis: [1,4,1,_,[[74,Rightvars,1]|_]].
Begin synthesis on code fragment [74,75] from node 74.
rule2.nod consulted 1588 bytes 0.283659 sec.
rule3.nod consulted 1828 bytes 0.350163 sec.
rule4.nod consulted 1184 bytes 0.216895 sec.
```

Figure A.21: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
Rule Application #2 of rule #4 shown in file Ra02,
for reason: Adding a Zero Maintains a Value,
original nodes were: [74,75],
new nodes are: [74,-5,-6,75].
history([instnode,70,[68,69,70,71,72,43,1],
                     [-2,71,72,-3,-4,43,-1,1],1,1])
synthstack([75,76,77,78])
hwinodmap([666,71,2])
hwinodmap([667,72,0])
hwinodmap([291,67,0])
hwinodmap([290,66,0])
hwinodmap([289,65,0])
varmap([A.LATCH,522,PC,3,[0,_,_],_,[[WIDTH,11,0,0,11]],1,[65,71]])
varmap([MP,2,M,2,[1,_,_],_,[[INDEX],[WIDTH,11,0,0,11]],1,[66]])
varmap([MBR,32,I,12,[0,_,_],_,[[WIDTH,11,0,0,11]],1,[67]])
varmap([F,519,$Newvar1,-1,[0,1,1],_,[[WIDTH,11,0,0,11]],2,[72]])
constants([1,_,1,_])
constants([MASK,24,-128,0])
hwdatapath([WORDS,289])
hwdatapath([ARRAYACCESS,290])
hwdatapath([ASSIGN,291])
hwdatapath([AND,666])
hwdatapath([ASSIGN,667])
constreg([MASK,24])
busnode([ALU.CIN,87,_])
busnode([S,560,_])
busnode([ALU.OUT,104,_])
busnode([TEMP.AM2901,-104,_])
busnode([TEMP.AM2901,104,_])
busnode([AM2901,539,_])
busnode([OUT,573,_])
busnode([OUT,-573,_])
busnode([Y,573,_])
busnode([R,561,_])
busnode([DIN,83,_])
C, 0, 2, [74,-6,75], [_, _, _], 74
N, 1, 3, [74,-6,75], [_, _, _], [], 74, 1, 0
N, 1, 4, [74,-6,75], [_, _, _], [], 74, 1, 0
N, 1, 5, [74,-6,75], [_, _, _], [], 74, 1, 0
N, 1, 6, [74,-6,75], [_, _, _], [], 74, 1, 0
C, 2, 2, [74,-6,75], [613, _, _], -6
N, 3, 3, [74,-6,75], [613, _, _], [], -6, 0, 1
N, 4, 3, [74,-6,75], [613, 614, _], [], -6, 1, 0
N, 4, 4, [74,-6,75], [613, 614, _], [], -6, 1, 0
```

Figure A.22: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

A.3.10 Controlflow

Controlflow synthesis is not implemented in UMS. However, the controlflow requirements for this fragment could be determined in a straightforward manner, similar to what is shown in Figure 1.6.

A.4 Synthesis of Third Fragment: "PC = PC+1"

Figure A.19 shows the beginning of synthesis of the instruction-set description fragment "PC=PC+1". The most distinctive type of node in this fragment is the addition node numbered 74.

A.4.1 Exhaustive search failure

As shown in Figures A.19, A.20, and A.21, a successful candidate set is not found. The ALU hardware performs addition in the expression "alu=R+S+Cn", where the first two operands to the right of the equals sign are the conventional inputs, the third operand is the carry input. The addition node at the end of Figure A.21 is shown to be causing the problem.

A.4.2 Transformation of problem node

A transformation traced in Figure A.22 changes "PC=PC+1" into "PC=PC+1+0" and allows the hardware carry bit to become part of the calculation. The location of the program counter "PC" has been moved to the hardware random-access memory scratchpad "RAM", as traced in the middle of Figure

A.23 and shown at the bottom as "varmap([RAM,521,PC,3,[0,_,_],0,[[WIDTH,11,0,0,11]],1,[65,71,74,75]])".

A.5 Synthesis of Fourth Fragment: "eadd='00000@pa"

The next instruction-set fragment to be synthesized is "eadd='00000@pa", which calculates for the instruction the effective address for the zero-page data-access mode (this occurs when bit 4 of the instruction is zero, as represented in Figure 7.1).

A.5.1 Exhaustive search failure

This is similar to the earlier case in this example, in which the microengine datapath does not directly support this type of bit manipulation. Figures A.24 and A.25 show search failure for the required bit-concatenation operation in the hardware. The top of Figure A.26 shows the UMS-deduced failure localized to instruction-set node 41, the concatenation operator.

A.5.2 Transformation of problem node

Figure A.26 shows applying a transformation rule to modify the concatenation operator to use a bit mask instead. Applying this programming trick requires attempting to set the high-order bits to zero (in "eadd='00000@pa"). An AND operator with the appropriate constant (which contains ones only in the right part of the word) will set the high-order bits of "pa" to zero (while using the

analysis that the instruction-set variable "pa" is actually an alternative name for the seven low-order bits of the variable "i"). The arc-matching proceeds quickly until the bottom of Figure A.27.

Remember that at the beginning of synthesis for this fragment, the instruction-set variable "i" was mapped to the memory-buffer register "MBR" (recorded in the middle of Figure A.27). The use of the variable "i" now has two constraints: it must be used as an output for the previous expression "i=M[PC];" and it must be accessible as an input for the expression "eadd='000001111111 and i". These constraints conflict because the present mapping of "i" to "MBR", and UMS cannot find a way to move "i" "upstream" or "downstream" to solve the conflict.

A.5.3 Revision of previous synthesis

Because mapping has proceeded as well as it has so far, UMS applies cost-benefit analysis to decide to continue using this hardware-node candidate set instead of abandoning it and trying other parts of the hardware. UMS proceeds based on the hypothesis that the early mapping of "i" to "MBR" may have been in error. The previous mapping of "i" to "MBR" is deleted temporarily, and synthesis for the current instruction-set fragment (shown near the bottom of Figure A.27) maps "i" to "s" (which is one of the inputs to the hardware ALU, shown in Figure 7.2). Figure A.28 shows the synthesis state.

```
N, 4, 6, [74,-6,75], [613, 614, _], [], -6, 1, 0
C, 5, 2, [74,-6,75], [613, 614, _], 75
N, 6, 3, [74,-6,75], [613, 614, _], [], 75, 0, 0
M, 7, 11, [74,-6,75], [613, 614, 615], PC, 74, 0, 2
M, 7, 11, [74,-6,75], [613, 614, 615], PC, 75, 1, 0
U, 7, 14, [74,-6,75], [613, 614, 615], 1, 74, 0, 2
U, 8, 14, [74,-6,75], [613, 614, 615], 0, -6, 0, 1
H, 9, 18, [74,-6,75], [613, 614, 615], PC, 74, 0, 2
H, 10, 19, [74,-6,75], [613, 614, 615], PC, 75, 1, 0
H, 10, 20, [74,-6,75], [613, 614, 615], PC, 75, 1, 0
H, 10, 21, [74,-6,75], [613, 614, 615], PC, 75, 1, 0
ALU.LSB _
ALU.MSB _
CN4 _
C.OUT _
0 _
F _
A.LATCH _
RAM _
RAM 521
RAM 521
```

movement of "PC"

```
Begin subroutine datapath synthesis from node 76.
Synthesize subroutine body [86].
Single input variable condition of control flow.
Begin subroutine datapath synthesis from node 87.
Synthesize subroutine body [39,45].
Single input variable condition of control flow.
Begin synthesis on code fragment [41,42] from node 41.
history([instnode,70,[68,69,70,71,72,43,1],
                    [-2,71,72,-3,-4,43,-1,1],1,1])
history([instnode,74,[74,75],[74,-5,-6,75],4,2])
synthstack([42,43,44])
synthstack([90,91,93,96,97,98,99,103,104,110,111,115,116,
            120,121,122,126,127,128,129,130,131])
synthstack([77,78])
hwinodmap([613,74,2])
hwinodmap([615,75,0])
hwinodmap([614,-6,1])
hwinodmap([666,71,2])
hwinodmap([667,72,0])
hwinodmap([291,67,0])
hwinodmap([290,66,0])
hwinodmap([289,65,0])
varmap([RAM,521,PC,3,[0,_,_],0,[[WIDTH,11,0,0,11]],1,[65,71,74,75]])
```

Figure A.23: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
varmap([MP,2,M,2,[1,_,_],_,[[INDEX],[WIDTH,11,0,0,11]],1,[66]])
varmap([MBR,32,I,12,[0,_,_],_,[[WIDTH,11,0,0,11]],1,[67]])
varmap([F,519,$Newvar1,-1,[0,1,1],_,[[WIDTH,11,0,0,11]],2,[72]])
constants([0,_,0,_])
constants([MASK,24,1,1])
constants([1,_,1,_])
constants([MASK,24,-128,0])
hwdatapath([WORDS,289])
hwdatapath([ARRAYACCESS,290])
hwdatapath([ASSIGN,291])
hwdatapath([AND,666])
hwdatapath([ASSIGN,667])
hwdatapath([ADD,613])
hwdatapath([ADD,614])
hwdatapath([ASSIGN,615])
constreg([MASK,24])
busnode([F,615,_])
busnode([A.LATCH,543,_])
busnode([ALU.CIN,89,_])
busnode([ALU.CIN,87,_])
busnode([S,560,_])
busnode([ALU.OUT,104,_])
busnode([TEMP.AM2901,-104,_])
busnode([TEMP.AM2901,104,_])
busnode([AM2901,539,_])
busnode([OUT,573,_])
busnode([OUT,-573,_])
busnode([Y,573,_])
busnode([R,561,_])
busnode([DIN,83,_])
C, 0, 2, [41,42], [_, _], 41
N, 1, 3, [41,42], [_, _], [], 41, 1, 0
N, 1, 4, [41,42], [_, _], [], 41, 1, 0
N, 1, 5, [41,42], [_, _], [], 41, 1, 0
N, 1, 6, [41,42], [_, _], [], 41, 1, 0
C, 2, 2, [41,42], [136, _], 42
N, 3, 3, [41,42], [136, _], [], 42, 0, 0
U, 4, 14, [41,42], [136, 137], 0, 41, 0, 0
U, 4, 24, [41,42], [136, 137], 0, 41, 0, 0
N, 3, 4, [41,42], [136, _], [], 42, 0, 0
N, 3, 5, [41,42], [136, _], [], 42, 0, 0
N, 3, 6, [41,42], [136, _], [], 42, 0, 0
N, 3, 7, [41,42], [136, _], [], 42, 0, 0
C, 2, 2, [41,42], [344, _], 42
```

Figure A.24: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
N, 3, 3, [41,42], [344, _], [], 42, 0, 0
U, 4, 14, [41,42], [344, 345], 0, 41, 0, 0
U, 4, 24, [41,42], [344, 345], 0, 41, 0, 0
N, 3, 4, [41,42], [344, _], [], 42, 0, 0
N, 3, 5, [41,42], [344, _], [], 42, 0, 0
N, 3, 6, [41,42], [344, _], [], 42, 0, 0
N, 3, 7, [41,42], [344, _], [], 42, 0, 0
C, 2, 2, [41,42], [582, _], 42
N, 3, 3, [41,42], [582, _], [], 42, 0, 0
U, 4, 14, [41,42], [582, 583], 0, 41, 0, 0
U, 4, 24, [41,42], [582, 583], 0, 41, 0, 0
U, 5, 13, [41,42], [582, 583], PA, 41, 0, 0
U, 5, 15, [41,42], [582, 583], PA, 41, 0, 0
N, 3, 4, [41,42], [582, _], [], 42, 0, 0
N, 3, 5, [41,42], [582, _], [], 42, 0, 0
N, 3, 6, [41,42], [582, _], [], 42, 0, 0
N, 3, 7, [41,42], [582, _], [], 42, 0, 0
C, 2, 2, [41,42], [591, _], 42
N, 3, 3, [41,42], [591, _], [], 42, 0, 0
U, 4, 14, [41,42], [591, 592], 0, 41, 0, 0
U, 4, 24, [41,42], [591, 592], 0, 41, 0, 0
U, 5, 13, [41,42], [591, 592], PA, 41, 0, 0
U, 5, 15, [41,42], [591, 592], PA, 41, 0, 0
N, 3, 4, [41,42], [591, _], [], 42, 0, 0
N, 3, 5, [41,42], [591, _], [], 42, 0, 0
N, 3, 6, [41,42], [591, _], [], 42, 0, 0
N, 3, 7, [41,42], [591, _], [], 42, 0, 0
C, 2, 2, [41,42], [597, _], 42
N, 3, 3, [41,42], [597, _], [], 42, 0, 0
U, 4, 14, [41,42], [597, 598], 0, 41, 0, 0
U, 4, 24, [41,42], [597, 598], 0, 41, 0, 0
N, 3, 4, [41,42], [597, _], [], 42, 0, 0
N, 3, 5, [41,42], [597, _], [], 42, 0, 0
N, 3, 6, [41,42], [597, _], [], 42, 0, 0
N, 3, 7, [41,42], [597, _], [], 42, 0, 0
C, 2, 2, [41,42], [606, _], 42
N, 3, 3, [41,42], [606, _], [], 42, 0, 0
U, 4, 14, [41,42], [606, 607], 0, 41, 0, 0
U, 4, 24, [41,42], [606, 607], 0, 41, 0, 0
N, 3, 4, [41,42], [606, _], [], 42, 0, 0
N, 3, 5, [41,42], [606, _], [], 42, 0, 0
N, 3, 6, [41,42], [606, _], [], 42, 0, 0
N, 3, 7, [41,42], [606, _], [], 42, 0, 0
N, 1, 7, [41,42], [_, _], [], 41, 1, 0
```

Figure A.25: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
Problems with node synthesis: [1,5,1,_,[[41,Rightvars,PA]|_]].
Begin synthesis on code fragment [41,42] from node 41.
rule5.nod consulted 1824 bytes 0.334049 sec.
Rule Application #3 of rule #5 shown in file Ra03,
for reason: Concatination of Zero with Value,
original nodes were: [40,41,42],
new nodes are: [40,41,42].
history([instnode,70,[68,69,70,71,72,43,1],
                      [-2,71,72,-3,-4,43,-1,1],1,1])
history([instnode,74,[74,75],[74,-5,-6,75],4,2])
synthstack([42,43,44])
synthstack([90,91,93,96,97,98,99,103,104,110,111,115,116,120,
            121,122,126,127,128,129,130,131])
synthstack([77,78])
hwinodmap([613,74,2])
hwinodmap([615,75,0])
hwinodmap([614,-6,1])
hwinodmap([666,71,2])
hwinodmap([667,72,0])
hwinodmap([291,67,0])
hwinodmap([290,66,0])
hwinodmap([289,65,0])
varmap([RAM,521,PC,3,[0,_,_],0,[[WIDTH,11,0,0,11]],1,[65,71,74,75]])
varmap([MP,2,M,2,[1,_,_],_,[[INDEX],[WIDTH,11,0,0,11]],1,[66]])
varmap([MBR,32,I,12,[0,_,_],_,[[WIDTH,11,0,0,11]],1,[67]])
varmap([F,519,$Newvar1,-1,[0,1,1],_,[[WIDTH,11,0,0,11]],2,[72]])
constants([0,_,0,_])
constants([MASK,24,1,1])
constants([1,_,1,_])
constants([MASK,24,-128,0])
hwdatapath([WORDS,289])
hwdatapath([ARRAYACCESS,290])
hwdatapath([ASSIGN,291])
hwdatapath([AND,666])
hwdatapath([ASSIGN,667])
hwdatapath([ADD,613])
hwdatapath([ADD,614])
hwdatapath([ASSIGN,615])
constreg([MASK,24])
busnode([F,615,519])
busnode([L,279,_])
busnode([A.LATCH,543,_])
busnode([ALU.CIN,89,_])
busnode([ALU.CIN,87,_])
```

Figure A.26: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
busnode([S,560,_])
busnode([ALU.OUT,104,_])
busnode([TEMP.AM2901,-104,_])
busnode([TEMP.AM2901,104,_])
busnode([AM2901,539,_])
busnode([OUT,573,_])
busnode([OUT,-573,_])
busnode([Y,573,_])
busnode([R,561,_])
busnode([DIN,83,_])
C, 0, 2, [41,42], [_, _], 41
N, 1, 3, [41,42], [_, _], [], 41, 1, 0
N, 1, 4, [41,42], [_, _], [], 41, 1, 0
N, 1, 5, [41,42], [_, _], [], 41, 1, 0
N, 1, 6, [41,42], [_, _], [], 41, 1, 0
C, 2, 2, [41,42], [666, _], 42
N, 3, 3, [41,42], [666, _], [], 42, 0, 0
M, 4, 11, [41,42], [666, 667], I, 41, 0, 2
U, 4, 14, [41,42], [666, 667], 127, 41, 0, 2
U, 4, 13, [41,42], [666, 667], EADD, 42, 1, 0
U, 4, 15, [41,42], [666, 667], EADD, 42, 1, 0
H, 4, 18, [41,42], [666, 667], I, 41, 0, 2
H, 5, 18, [41,42], [666, 667], 127, 41, 0, 2
H, 5, 21, [41,42], [666, 667], 127, 41, 0, 2
H, 5, 22, [41,42], [666, 667], 127, 41, 0, 2
H, 5, 23, [41,42], [666, 667], 127, 41, 0, 2
H, 4, 21, [41,42], [666, 667], I, 41, 0, 2
H, 4, 22, [41,42], [666, 667], I, 41, 0, 2
U, 4, 24, [41,42], [666, 667], 127, 41, 0, 2
M, 4, 11, [41,42], [666, 667], I, 41, 0, 1
U, 4, 14, [41,42], [666, 667], 127, 41, 0, 1
U, 5, 13, [41,42], [666, 667], EADD, 42, 1, 0
U, 5, 15, [41,42], [666, 667], EADD, 42, 1, 0
H, 5, 18, [41,42], [666, 667], I, 41, 0, 1
H, 5, 21, [41,42], [666, 667], I, 41, 0, 1
H, 5, 22, [41,42], [666, 667], I, 41, 0, 1
U, 5, 13, [41,42], [666, 667], I, 41, 0, 1
Hw/var "S" mapped to inst/var "I".
Rule Application #4 of rule #2 shown in file Ra04,
for reason: Adding a Zero and Carry Zero Maintains
a Value to an Array read,
original nodes were: [66,67],
new nodes are: [66,-7,-8,-9,-10,67].
```

suspend and remap "i"

Figure A.27: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
history([instnode,70,[68,69,70,71,72,43,1],
                      [-2,71,72,-3,-4,43,-1,1],1,1])
history([instnode,74,[74,75],[74,-5,-6,75],4,2])
synthstack([42,43,44])
synthstack([90,91,93,96,97,98,99,103,104,110,111,115,116,120,
            121,122,126,127,128,129,130,131])
synthstack([77,78])
hwinodmap([666,41,1])
hwinodmap([667,42,0])
hwinodmap([613,74,2])
hwinodmap([615,75,0])
hwinodmap([614,-6,1])
hwinodmap([666,71,2])
hwinodmap([667,72,0])
varmap([F,519,$Newvar1,-1,[0,1,1],_,[[WIDTH,11,0,0,11]],2,[72]])
varmap([RAM,521,PC,3,[0,_,_],0,[[WIDTH,11,0,0,11]],1,[71,74,75]])
varmap([S,518,I,12,[0,_,_],_,[[WIDTH,11,0,0,11]],4,[41]])
constants([MASK,24,127,2])
constants([0,_,0,_])
constants([MASK,24,1,1])
constants([1,_,1,_])
constants([MASK,24,-128,0])
hwdatapath([WORDS,289])
hwdatapath([ARRAYACCESS,290])
hwdatapath([ASSIGN,291])
hwdatapath([AND,666])
hwdatapath([ASSIGN,667])
hwdatapath([ADD,613])
hwdatapath([ADD,614])
hwdatapath([ASSIGN,615])
constreg([MASK,24])
busnode([F,615,519])
busnode([L,279,_])
busnode([A.LATCH,543,_])
busnode([ALU.CIN,89,_])
busnode([ALU.CIN,87,_])
busnode([S,560,_])
busnode([ALU.OUT,104,_])
busnode([TEMP.AM2901,-104,_])
busnode([TEMP.AM2901,104,_])
busnode([AM2901,539,_])
busnode([OUT,573,_])
busnode([OUT,-573,_])
busnode([Y,573,_])
```

Figure A.28: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

UMS then suspends its current synthesis effort to determine whether the old uses of "i" can be made to work with its new mapping to "s". This did not succeed (at just one location, the output arc of the expression "i=M[PC];"). When this mapping fails, UMS decides the cost-benefits are in favor of again suspending its current work, as shown near the top of Figure A.29. Applying transformation rules using the "i" variable modifies experimentally the arc. The expression "i=M[PC];" is changed into "i=M[PC]+0+0;" by applying a general rule to simply add a value zero and a zero carry input to a value, which in this case was the memory access ("M[PC]") of the new value for "i". Arc and node matching for these new nodes proceeds as shown in Figure A.29.

A.5.4 Partitioning code into multiple microwords

Arc mapping proceeds for the new expression "i=M[PC]+0+0;" as shown in Figures A.28 and A.29. Because this expression is too complex for the microengine, synthesis is suspended again so that a special rule (labeled zero, which is directly coded into UMS) is used at the top of Figure A.30. This rule creates a temporary variable, so the calculation "i=M[PC]+0+0;" becomes "$Newvar3=M[PC] next i=$Newvar3+0+0;", which is successfully synthesized. Rule number zero can be used to spread complex calculations across multiple microwords.

After these synthesis suspensions, UMS finishes synthesis of the original fragment "eadd='000001111111 and i", which is shown as completed as

"H, 6, 9, [41,42], [666, 667], EADD, 42, 1, 0" in Figure A.30. Although synthesis resumes with node numbers -4, 43, and 44, the example ends here.

```
busnode([R,561,_])
busnode([DIN,83,_])
currenttask([41,42])
```

suspend synthesis to solve subproblem

```
C, 0, 2, [-8,-10,65,66,67], [_, _, _, _, _], -8
N, 1, 3, [-8,-10,65,66,67], [_, _, _, _, _], [], -8, 0, 1
N, 1, 4, [-8,-10,65,66,67], [_, _, _, _, _], [], -8, 0, 1
N, 1, 5, [-8,-10,65,66,67], [_, _, _, _, _], [], -8, 0, 1
N, 1, 6, [-8,-10,65,66,67], [_, _, _, _, _], [], -8, 0, 1
N, 1, 7, [-8,-10,65,66,67], [_, _, _, _, _], [], -8, 0, 1
N, 2, 7, [-8,-10,65,66,67], [613, _, _, _, _], [], -8, 0, 2
N, 2, 8, [-8,-10,65,66,67], [613, _, _, _, _], [], -8, 0, 2
N, 2, 3, [-8,-10,65,66,67], [613, _, _, _, _], [], -8, 1, 0
N, 2, 4, [-8,-10,65,66,67], [613, _, _, _, _], [], -8, 1, 0
N, 2, 6, [-8,-10,65,66,67], [613, _, _, _, _], [], -8, 1, 0
C, 3, 2, [-8,-10,65,66,67], [613, _, _, _, _], -10
N, 4, 3, [-8,-10,65,66,67], [613, _, _, _, _], [], -10, 0, 1
N, 5, 3, [-8,-10,65,66,67], [613, 614, _, _, _], [], -10, 1, 0
N, 5, 4, [-8,-10,65,66,67], [613, 614, _, _, _], [], -10, 1, 0
N, 5, 6, [-8,-10,65,66,67], [613, 614, _, _, _], [], -10, 1, 0
N, 6, 3, [-8,-10,65,66,67], [613, 614, _, _, _], [], 65, 1, 0
N, 6, 4, [-8,-10,65,66,67], [613, 614, _, _, _], [], 65, 1, 0
N, 6, 5, [-8,-10,65,66,67], [613, 614, _, _, _], [], 65, 1, 0
N, 6, 6, [-8,-10,65,66,67], [613, 614, _, _, _], [], 65, 1, 0
N, 7, 3, [-8,-10,65,66,67], [613, 614, 289, _, _], [], 66, 0, 0
N, 8, 3, [-8,-10,65,66,67], [613, 614, 289, 290, _], [], 66, 1, 0
N, 8, 4, [-8,-10,65,66,67], [613, 614, 289, 290, _], [], 66, 1, 0
C, 9, 2, [-8,-10,65,66,67], [613, 614, 289, 290, _], 67
N, 10, 3, [-8,-10,65,66,67], [613, 614, 289, 290, _], [], 67, 0, 0
M, 11, 11, [-8,-10,65,66,67], [613, 614, 289, 290, 615], PC, 65, 0, 0
M, 11, 11, [-8,-10,65,66,67], [613, 614, 289, 290, 615], I, 67, 1, 0
U, 11, 14, [-8,-10,65,66,67], [613, 614, 289, 290, 615], 0, -8, 0, 1
U, 12, 14, [-8,-10,65,66,67], [613, 614, 289, 290, 615], 0, -10, 0, 1
U, 13, 13, [-8,-10,65,66,67], [613, 614, 289, 290, 615], M, 66, 0, 0
Hw/var "MP" mapped to inst/var "M".
H, 14, 18, [-8,-10,65,66,67], [613, 614, 289, 290, 615], PC, 65, 0, 0
H, 15, 19, [-8,-10,65,66,67], [613, 614, 289, 290, 615], I, 67, 1, 0
H, 15, 20, [-8,-10,65,66,67], [613, 614, 289, 290, 615], I, 67, 1, 0
H, 15, 21, [-8,-10,65,66,67], [613, 614, 289, 290, 615], I, 67, 1, 0
Q _
B, 16, 25, [-8,-10,65,66,67], [613, 614, 289, 290, 615], [], 66, 1, 0
B, 16, 26, [-8,-10,65,66,67], [613, 614, 289, 290, 615], [], 66, 1, 0
DIN 54
```

Figure A.29: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

```
Rule Application #5 of rule #0 shown in file Ra05,
for reason: Create Intermediate Value,
original nodes were: [1,64,66,-8],
new nodes are: [-12,1,64,66,-13,-8].
N, 10.5, 3, [-8,66,-13], [613, 290, _], [], 66, 0, 0
N, 11.5, 3, [-8,66,-13], [613, 290, _], [], 66, 1, 0
N, 11.5, 4, [-8,66,-13], [613, 290, _], [], 66, 1, 0
N, 11.5, 6, [-8,66,-13], [613, 290, _], [], 66, 1, 0
C, 12.5, 2, [-8,66,-13], [613, 290, _], -13
N, 13.5, 3, [-8,66,-13], [613, 290, _], [], -13, 0, 0
U, 14.5, 13, [-8,66,-13], [613, 290, 291], $Newvar3, -8, 0, 1
Hw/var "R" mapped to inst/var "$Newvar3".
U, 15.5, 15, [-8,66,-13], [613, 290, 291], $Newvar3, -13, 1, 0
H, 15.5, 19, [-8,66,-13], [613, 290, 291], $Newvar3, -13, 1, 0
H, 15.5, 20, [-8,66,-13], [613, 290, 291], $Newvar3, -13, 1, 0
H, 6, 19, [41,42], [666, 667], EADD, 42, 1, 0
Hw/var "F" mapped to inst/var "EADD".
Begin synthesis on code fragment [-4,43,44] from node 43.
history([instnode,70,[68,69,70,71,72,43,1],
                    [-2,71,72,-3,-4,43,-1,1],1,1])
history([instnode,74,[74,75],[74,-5,-6,75],4,2])
history([instnode,1,[1,64,66,-8],[-12,1,64,66,-13,-8],0,5])
history([instnode,67,[66,67],[66,-7,-8,-9,-10,67],2,4])
history([instnode,41,[40,41,42],[40,41,42],5,3])
synthstack([44])
synthstack([90,91,93,96,97,98,99,103,104,110,111,115,116,120,
            121,122,126,127,128,129,130,131])
synthstack([77,78])
hwinodmap([291,-13,0])
hwinodmap([615,67,0])
hwinodmap([290,66,0])
hwinodmap([289,65,0])
hwinodmap([614,-10,1])
hwinodmap([613,-8,1])
hwinodmap([666,41,1])
hwinodmap([667,42,0])
hwinodmap([613,74,2])
hwinodmap([615,75,0])
hwinodmap([614,-6,1])
hwinodmap([666,71,2])
hwinodmap([667,72,0])
varmap([R,320,$Newvar3,-12,[0,_,_],_,[[WIDTH,11,0,0,11]],6,[-13,-8]])
varmap([Q,520,I,12,[0,_,_],_,[[WIDTH,11,0,0,11]],4,[41,67]])
varmap([RAM,521,PC,3,[0,_,_],0,[[WIDTH,11,0,0,11]],1,[65,71,74,75]])
```

suspention of suspention

resume original synthesis

Figure A.30: Continued trace of UMS behavior while synthesizing PDP-8 microcode (see text).

BIBLIOGRAPHY

[ABUKL79] Abu-sufah, W., Kuck, D., and Lawrie, D., "Automatic Program Transformation for Virtual Memory Computers", *National Computer Conference*, Vol. 48, 1979, Pages 969 to 974.

[ADA78] Adams, P. H., "Microprogrammable Microprocessor Survey", *SIGMICRO*, Vol. 9, No. 1, 1-Mar-78, Pages 7 to 38.

[AMD81] Advanced Micro Devices, *Bipolar Microprocessor Logic and Interface Data Book*, 1981, Sunnyvale, California.

[AGE76] Agerwala, T., "Microprogram Optimization: A Survey", *IEEE Transactions on Computers*, Vol. 25, No. 10, 1-Oct-76, Pages 962 to 973.

[AGRR76] Agrawala, A. K., and Rauscher, T. G., *Foundations of Microprogramming Architecture, Software, and Applications*, Academic Press, New York, New York, 1976, Pages 1 to 416.

[AHOJ74] Aho, A. V., and Johnson, S. C., "LR Parsing", *Computing Surveys*, Vol. 6, No. 2 ,1-Jun-74, Pages 99 to 124.

[AHOSU86] Aho, A. V., Sethi, R., and Ullman, J. D., *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, Massachusetts, 1986, Pages 1 to 796.

[AHOU77] Aho, A. V., and Ullman, J. D., *Principles of Compiler Design*, Addison-Wesley, Reading, Massachusetts, 1977, Pages 1 to 604.

[AHOU76] Aho, A. V., and Ullman, J. D., "Optimal Code Generation for Expression Trees", *Journal of the ACM*, Vol. 23, No. 3, 1-Jul-76, Pages 488 to 501.

[AHOU72] Aho, A. V., and Ullman, J. D., "Optimization of Straight Line Programs", *SIAM Journal of Computation*, Vol. 1, No. 1, 1-Mar-72, Pages 1 to 19.

[ALP81] Altman, A. H., and Parker, A. C., "The SLIDE Simulator: A Facility for the Design and Analysis of Computer Interconnections", Carnegie-Mellon University Technical Report, 1-Jun-81, Pages 1 to 4, also published as: *17th Design Automation Conference Proceedings*, 23Jun-80 Pages 148 to 155.

[AMM77] Amman, U., "On Code Generation in a Pascal Compiler", *Software - Practice and Experience*, Vol. 7, No. 3, 1-Jun-77, Pages 391 to 423.

[AND80] Andrews, M., *Principles of Firmware Engineering in Microprogram Control*, Computer Science Press, Potomac, Maryland, 1980, Pages 1 to 347.

[AND79] Andrews, R. B., *Proving Programs Correct*, John Wiley and Sons, New York, New York, 1979, Pages 1 to 184.

[ANKCHM82] Anklam, P., Cutler, D., Heinen Jr., R., and MacLaren, M. D., *Engineering a Compiler VAX-11 Code Generation and Optimization*, Digital Press, Bedford, Massachusetts, 1982, Pages 1 to 269.

[ARM77] Armstrong, C. V. W., "Multimicroprocessor Architecture and the Use of Multi-Level Encoding in Microinstruction Formats", *Micros, Minis, and Maxis*, COMPCON Fall 1977.

[ATK84] Atkins, R. P., "Improved Instruction Formulation in Exhaustive Local Microcode Compaction Algorithm", *17th Annual Microprogramming Workshop*, ACM , 30-Oct-84, Pages 105 to 111.

[BABH81] Baba, T., and Hagiwara, B., "The MPG System: A Machine-Independent Efficient Microprogram Generator", *IEEE Transactions on Computers*, Vol. 30, No. 6, 1-Jun-81, Pages 373 to 395.

[BAEK79] Baer, J., and Koyama, B., "On the Minimization of the Width of the Control Memory of Microprogrammed Processors", *IEEE Transactions on Computers*, Vol. 28, No. 4, 1-Apr-79, Pages 310 to 317.

[BAK80] de Bakker, J., *Mathematical Theory of Program Correctness*, Prentice-Hall, Englewood Cliffs, New Jersey, 1980, Pages 1 to 505.

[BALL85] Balbin, I., Lecot, K., *Logic Programming: a Classified Bibliography*, Wildgrass Books, Fitzroy, Victoria, Australia, 1985, Pages 1 to 360.

[BAL85A] Balzer, R., "Automated Enhancement of Knowledge Representation", *IJCAI*, William Kaufmann, Los Altos, California, 18-Aug-85, Pages 203 to 207.

[BAL85B] Balzer, R., "A 15 Year Perspective on Automatic Programming", *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, 1-Nov-85, Pages 1257 to 1268.

[BAL81] Balzer, R., "Transformational Implementation: An Example", *IEEE Transactions on Software Engineering*, Vol. 7, No. 1, 1981, Pages 3 to 14.

[BANCKT79] Banerjee, U., Chen, S. C., Kuck, D. J., and Towle, R. A., "Time and Parallel Processor Bounds for FORTRAN-like Loops", *IEEE Transactions on Computers*, Vol. 28, No. 9, 1-Sep-79, Pages 660 to 670.

[BANR82] Banerji, D. K., and Raymond, J., *Elements of Microprogramming*, Prentice-Hall, Englewood Cliffs, New Jersey, 1982, Pages 1 to 434.

[BAR79A] Barbacci, M. R., "An ISPS Primer for the Instruction Set Processor Notation", in: Bell, Mudge, and McNamara, *Computer Engineering A DEC View of Hardware Engineering*, 1-Sep-78, Pages 519 to 536.

[BAR79B] Barbacci, M. R., "Instruction Set Processor Specifications (ISPS): The Notation and Its Applications", *IEEE Transactions on Computers*, Vol. 30, No. 1, 1-Jan-81, Pages 24 to 40, also published as: Carnegie-Mellon University Technical Report, 17-May-79, Pages 1 to 37.

[BAR79C] Barbacci, M. R., "Instruction Set Processor Specifications for Simulation, Evaluation, and Synthesis", *16th Design Automation Conference Proceedings*, 25-Jun-79, Pages 64 to 72.

[BARBCS78] Barbacci, M. R., Barnes, G. E., Cattell, R. G., and Siewiorek, D. P., "The ISPS Computer Description Language", Department of Computer Science, Carnegie-Mellon University Technical Report, 6-Mar-78.

[BARN78] Barbacci, M. R., and Nagle, A. W., "The Symbolic Manipulation of Computer Descriptions ISPS Application Note", Carnegie-Mellon University Technical Report, 7-Mar-78, Pages 1 to 36.

[BARS82] Barbacci, M. R., Siewiorek, D. P., *The Design and Analysis of Instruction Set Processors*, McGraw-Hill, New York, New York, 1982, Pages 1 to 243.

[BARM86] Barney, C. and Manuel, T., "RISC: Is It a Good Idea or Just Another Hype?", *Electronics*, 5-May-86, Pages 28 to 31.

[BARR81] Barr, A., and Feigenbaum, E. A., (eds.), *The Handbook of Artificial Intelligence, Vol. 1*, William Kaufmann, Los Altos, California, 1981, Pages 1 to 409.

[BARR82] Barr, A., and Feigenbaum, E. A., (eds.), *The Handbook of Artificial Intelligence, Vol. 2*, William Kaufmann, Los Altos, California, 1982, Pages 1 to 427.

[BARS85A] Barstow, D. R., "On the Convergence Toward a Database of Program Transformations", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 1, 1-Jan-85, Pages 1 to 9.

[BARS85B] Barstow, D. R., "Automatic Programming for Streams", *IJCAI*, William Kaufmann, Los Altos, California, 18-Aug-85, Pages 232 to 237.

[BARS85C] Barstow, D. R., "Domain-Specific Automatic Programming", *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, 1-Nov-85, Pages 1321 to 1336.

[BARS83] Barstow, D. R., "A Perspective on Automatic Programming", *IJCAI*, William Kaufmann, Los Altos, California, 8-Aug-83, Pages 1170 to 1179.

[BARS80A] Barstow, D. R., 'Remarks on "A Synthesis of Several Sorting Algorithms" by John Darlington', *ACTA Informatica*, Vol. 13, 1980, Pages 225 to 227.

[BARS80B] Barstow, D. R., "The Roles of Knowledge and Deduction in Algorithm Creation", Yale University Technical Report, No. 178, 1-Apr-80, Pages 1 to 27.

[BARS79] Barstow, D. R., *Knowledge-Based Program Construction*, 1979, Elsevier North-Holland, New York, New York, Pages 1 to 262. An earlier version is: Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules, Stanford Artificial Intelligence Laboratory, Memo AIM-308, 1-Nov-77.

[BAUE76] Bauer, F. L., and Eickel, J., (eds.), *Compiler Construction, an Advanced Course*, 2nd Edition, Springer-Verlag, New York, New York, 1976, Pages 1 to 638.

[BELLMM78] Bell, G. C., Mudge, J. C., and McNamara, J. E., *Computer Engineering A DEC View of Hardware Engineering*, Digital Press, Bedford, Massachusetts, 1-Sep-78, Pages 1 to 585.

[BELN71] Bell, G. C., and Newell, A., *Computer Structures: Readings and Examples*, McGraw-Hill, New York, New York, 1971, Pages 1 to 668.

[BIEG83] Biermann, A. W., and Guiho, G., (eds.), *Computer Program Synthesis Methodologies*, D. Reidel, Boston, Massachusetts, 1983, Pages 1 to 374.

[BOYM79] Boyer, R. S., and Moore, J. S., *A Computational Logic*, Academic Press, New York, New York, 1979, Pages 1 to 384.

[BREH81] Breuer, M. and Hartenstein, R., (eds.), *Computer Hardware Descriptions and their Applications*, Elsevier North-Holland, New York, New York, 7-Sep-81, Pages 1 to 302.

[BRO83] Broy, M., "Program Construction by Transformations: A Family Tree of Sorting Programs", in: *Computer Program Synthesis Methodologies*, A. W. Biermann and G. Guiho, (eds.), 1983, Pages 1 to 49.

[BURD77] Burstall, R. M., and Darlington, J., "A Transformational System for Developing Recursive Programs", *Journal of the ACM*, Vol. 24, No. 1, 1977, Pages 44 to 67.

[CARJB78] Carter, W. C., Joyner, W. H., and Brand, D., "Microprogram Verification Considered Necessary", *AFIPS Conference Proceedings*, 1978, Pages 657 to 664.

[CASJK80] Casavant, A. E., Gajski, D. D., and Kuck, D. J., "Automatic Design with Dependence Graphs", *17th Design Automation Conference*, 23-Jun-80, Pages 506 to 515.

[CATT80] Cattell, R. G., "Automatic Derivation of Code Generators from Machine Descriptions", *ACM Transactions on Programming Languages*, Vol. 2, No. 2, 1-Apr-80, Pages 173 to 190.

[CATT79] Cattell, R. G., "Code Generation and Machine Descriptions", Xerox PARC Technical Report, 1-Oct-79, Pages 1 to 48.

[CATT78] Cattell, R. G., "Formalization and Automatic Derivation of Code Generators", Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University Technical Report, 1-Apr-78, Pages 1 to 126.

[CATNL79] Cattell, R. G., Newcomer, J. M., and Leverett, B. W., "Code Generation in a Machine-independent Compiler", *SIGPLAN Notices*, Vol. 14, No. 8, 1-Aug-79, Pages 65 to 75.

[CHENK75] Chen, S. C., and Kuck, D. J, "Time and Parallel Processor Bounds for Linear Recurrence Systems", *IEEE Transactions on Computers*, Vol. 24, No. 7, 1-Jul-75, Pages 701 to 717.

[CHRM80] Chroust, G., and Mühlbacher, J. R., (eds.), *Firmware, Microprogramming, and Restructurable Hardware*, Elsevier North-Holland, New York, New York, 1980, Pages 1 to 310.

[CHU65] Chu, Y., "An ALGOL-like Computer Design Language", *Communications of the ACM*, Vol. 8, No. 10, 1-Oct-65, Pages 607 to 615.

[CLAL82] Clark, D. W., and Levy, H. M., "Measurement and Analysis of Instruction Use in the VAX-11/780", *9th Annual Symposium on Computer Architecture*, 26-Apr-82, Pages 9 to 17.

[CLAS80] Clark, D. W., and Strecker, W. D., 'Comments on "The Case for the Reduced Instruction Set Computer" by Patterson and Ditzel', *Computer Architecture News*, Vol. 8, No. 6, 15-Oct-80, Pages 34 to 38.

[CLAT82] Clark, K. L., and Tarnlund, S. A., *Logic Programming*, Academic Press, New York, New York, 1982, Pages 1 to 366.

[CLA85] Clarke Jr., E. M., "The Characterization Problem for Hoare Logics", in: *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, (eds.), 1985, Pages 89 to 103.

[CLOM81] Clocksin, W. F., and Mellish, C. S., *Programming in PROLOG*, Springer-Verlag, New York, New York, 1-Jun-81, Pages 1 to 279.

[CLOUT80] Cloutier, R. J., "Control Allocation: the Automated Design of Digital Controllers", M.S. Thesis, Carnegie-Mellon University, Electrical Engineering Department, 18-Apr-80, Pages 1 to 53.

[COFF76] Coffman, E. G. Jr. (Ed.), *Computer and Job-Shop Scheduling Theory*, John Wiley and Sons, New York, New York, 1976.

[COH83] Cohen, D., "Symbolic Execution of the Gist Specification Language", *IJCAI*, William Kaufmann, Los Altos, California, 8-Aug-83, Pages 17 to 20.

[COHF82] Cohen, P. R., and Feigenbaum, E. A., (eds.), *The Handbook of Artificial Intelligence, Vol. 3*, William Kaufmann, Los Altos, California, 1982, Pages 1 to 639.

[COL78] Colmerauer, A., *Metamorphosis Grammars*, Lecture Notes in Computer Science: Springer-Verlag, New York, New York, Vol. 63, 1978, Pages 133 to 189.

[COLHJ83] Colwell, R. P., Hitchcock III, C. Y., and Jensen, E. D., "Peering through the RISC/CISC Fog: an Outline of Research", *Computer Architecture News*, Vol. 11, No. 1, 1-Mar-83, Pages 44 to 50.

[CON79] Constable, R., "A Discussion of Program Verification", in: *Research Directions in Software Technology*, P. Wegner, (ed.), 1979, Pages 393 to 404.

[CORDV79A] Cory, W. E., Duley, J. R., and van Cleemput, W. M., "An Introduction to the DDL-P Language", Stanford Computer Systems Laboratory Technical Report, No. 163, 1-Mar-79, Pages 1 to 97.

[CORDV79B] Cory, W. E., Duley, J. R., and van Cleemput, W. M., "DDL-P Command Language Manual", Stanford Computer Systems Laboratory Technical Report, No. 164, 1-Mar-79, Pages 1 to 39.

[CROMV80A] Crocker, S. D., Marcus, L., and van-Mierop, D., "Microcode Verification Project Final Report", University of Southern California Information Sciences Institute, 1-Feb-80, Pages 1 to 134.

[CROMV80B] Crocker, S. D., Marcus, L., and van-Mierop, D., "The ISI Microcode Verification System", in: G. Chroust and J. R. Mühlbacher, (eds.), *Firmware, Microprogramming, and Restructurable Hardware*, 1980, Pages 89 to 102.

[DAR81] Darlington, J., "An Experimental Program Transformation and Synthesis System", *Artificial Intelligence*, Vol. 16, 1981, Pages 1 to 46.

[DAR78] Darlington, J., "A Synthesis of Several Sorting Algorithms", *ACTA Informatica*, Vol. 11, 1978, Pages 1 to 30.

[DARB73] Darlington, J., and Burstall, R. M., "A System Which Automatically Improves Programs", *IJCAI*, William Kaufmann, Los Altos, California, 1973, Pages 479 to 485.

[DARF80] Darlington, J., and Feather, M., "A Transformational Approach to Modification", Imperial College of Science and Technology Technical Report, 1980, Pages 1 to 22.

[DASBC73] Das, S. R., Banerji, D. K., and Chattopadhyay, A., "On Control Memory Minimization in Microprogrammed Digital Computers", *IEEE Transactions on Computers*, Vol. 22, No. 9, 1-Sep-73, Pages 945 to 948.

[DAS84] Dasgupta, S., *The Design and Description of Computer Architectures*, John Wiley & Sons, New York, New York, 1984, Pages 1 to 300.

[DAS80A] Dasgupta, S., "Towards a Microprogramming Language Schema", *Proceedings of the 11th Annual Microprogramming Workshop*, ACM, 1-Nov-78, Pages 144 to 153.

[DAS80] Dasgupta, S., "Some Aspects of High Level Microprogramming", *ACM Computing Surveys*, Vol. 12, No. 3, 1-Sep-80, Pages 295 to 323.

[DAS79] Dasgupta, S., "The Optimization of Microprogram Stores", *Computing Surveys*, Vol. 11, No. 1, 1-Mar-79, Pages 40 to 65.

[DAVF80] Davidson, J. W., and Fraser, C. W., "The Design and Application of a Retargetable Peephole Optimizer", *ACM Transactions on Programming Languages*, Vol. 2, No. 2, 1-Apr-80, Pages 191 to 202.

[DAVF79] Davidson, J. W., and Fraser, C. W., "The Retargetable Peephole Optimizer PO*", University of Arizona Technical Report, 1-Jun-79, Pages 1 to 25.

[DAV83] Davidson, S, "High Level Microprogramming - Current Usage, Future Prospects", *16th Annual Microprogramming Workshop*, ACM, 11-Oct-83, Pages 193 to 200.

[DAVLSM81] Davidson, S., Landskov, D., Shriver, B. D., and Mallett, P. W., "Some Experiments in Local Microcode Compaction for Horizontal Machines", *IEEE Transactions on Computers*, Vol. 30, No. 7, 1-Jul-81, Pages 460 to 477.

[DAVS80A] Davidson, S., and Shriver, B. D., "Firmware Engineering: An Extensive Update", in: *Firmware, Microprogramming, and Restructurable Hardware*, G. Chroust and J. R. Mühlbacher, (eds.), 1980, Pages 1 to 36.

[DAVS80B] Davidson, S., and Shriver, B. D., "MARBLE: A High Level Machine Independent Language for Microprogramming", in: *Firmware, Microprogramming, and Restructurable Hardware*, G. Chroust and J. R. Mühlbacher, (eds.), 1980, Pages 253 to 263.

[DAVS78] Davidson, S., and Shriver, B. D., "An Overview of Firmware Engineering", *Computer*, IEEE, 1-May-78, Pages 21 to 33.

[DAV84] Davis, R., "Amplifying Expertise with Expert Systems", in: *The AI Business*, by P. H. Winston and K. A. Prendergast, 1984, Pages 17 to 40.

[DAVK77] Davis, R., and J. King, "An Overview of Production Systems", *Machine Intelligence*, Vol. 8, 1977, Pages 300 to 332.

[DAVL82] Davis, R., and Lenat, D. B., *Knowledge-Based Systems in Artificial Intelligence*, McGraw-Hill, New York, New York, 1982, Pages 1 to 490.

[DEA80] Deak, E. G., "A Transformational Approach to the Development and Verification of Programs in a Very High Level Language", Ph.D. Dissertation, New York University,1-Nov-80.

[DER85] Dershowitz, N., "Synthesis by Completion", *IJCAI*, William Kaufmann, Los Altos, California, 18-Aug-85, Pages 208 to 214.

[DEW76] Dewitt, D. J., "A Machine Independent Approach to the Production of Optimized Horizontal Microcode", Ph.D. Dissertation, Department of Computer, Information, and Control Engineering, University of Michigan, 1-Jun-76.

[DIE77] Dietmeyer, D. L, "Translation of DDL Descriptions of Digital Systems", University of Wisconsin-Madison Electrical and Computer Engineering Technical Report, 1-Sep-77, Pages 1 to 46.

[DITP80] Ditzel, D. R., and Patterson, D. A., "Retrospective on High-level Language Computer Architecture", *7th Annual Symposium on Computer Architecture*, 6-May-80, Pages 97 to 104.

[DON76] Donahue, J. E., *Complementary Definitions of Programming Language Semantics*, Springer-Verlag, New York, New York, 1976, Pages 1 to 172.

[DONNF79] Donegan, M. K., Noonan, R. E., and Feyock, S., "A Code Generator Generator Language", *Sigplan Notices*, Vol. 14, No. 8, 1-Aug-79, Pages 58 to 64.

[ECK71] Eckhouse, R. H., "A High-level Microprogramming Language (MPL)", *Proceedings of the 1971 AFIPS Spring Joint Computer Conference*, Vol. 38, Pages 169 to 177, also see "A High-level Microprogramming Language (MPL)", Ph.D. Dissertation, State University of New York at Buffalo, 1971.

[EVE81] Eveking, H., "The Application of CONLAN Assertions to the Correct Description of Hardware", in: *Computer Hardware Descriptions and their Applications*, M. Breuer and R. Hartenstein, (eds.), 7-Sep-81, Pages 37 to 50.

[FALK74] Falk, H., "Hard-soft Tradeoffs", *IEEE Spectrum*, Vol. 11, No. 2, 1-Feb-74, Page 34.

[FEA82] Feather, M. S., "A System for Assisting Program Transformation", *ACM Transactions on Programming Languages and Systems*, Vol. 4, No. 1, 1-Jan-82, Pages 1 to 20.

[FEA79] Feather, M. S., "A System for Developing Programs by Transformation", 1979, Ph.D. Dissertation, University of Edinburgh.

[FEIM83] Feigenbaum, E. A., and McCorduck, P., *The Fifth Generation Artificial Intelligence and Japan's Computer Challenge to the World*, Addison-Wesley, Reading, Massachusetts, 1983, Pages 1 to 275.

[FERB73] Fernandez, E. B., and Bussell, B., "Bounds on the Number of Processors and Time for Multiprocessor Optimal Schedule", *IEEE Transactions on Computers*, Vol. 22, No. 8, 1-Aug-73, Pages 745 to 751.

[FIS81] Fisher, J. A., "Trace Scheduling: A Technique for Global Microcode Compaction", *IEEE Transactions on Computers*, Vol. 30, No. 7, 1-Jul-81, Pages 478 to 490.

[FIS79] Fisher, J. A., "The Optimization of Horizontal Microcode Within Basic Blocks and Beyond", Ph.D. Dissertation, Courant Institute, New York University, 1-Oct-79.

[FISLS81] Fisher, J. A., Landskov, D., and Shriver, B., "Microcode Compaction: Looking Backward and Looking Forward", *Proceedings NCC*, 1981, Pages 95 to 102.

[FIZFKLPPPSSV81] Fitzpatrick, D. T., Foderaro, J. K., Katevenis, M. G., Landman, H. A., Patterson, D. A., Peek, J. B., Peshkess, Z., Sequin, C. H., Sherburne, R. W., and Van Dyke, K. S., "A RISCy Approach to VLSI", *Computer Architecture News*, Vol. 10, No. 1, 1-Mar-81, Pages 28 to 32, also in: VLSI Design, 1-Sep-81, Pages 14 to 20.

[FODVP82] Foderaro, J. K., Van Dyke, K. S., and Patterson, D. A., "Running RISCs", *VLSI Design*, 1-Sep-82, Pages 27 to 32.

[FOSI85] Foster, C. C., and Iberall, T., *Computer Architecture*, Van Nostrand Reinhold, New York, New York, 1985, Pages 1 to 386.

[FRA79] Fraser, C. W., "A Compact, Machine-Independent Peephole Optimizer", *Conference Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, 1979, Pages 1 to 6.

[FRA77] Fraser, C. W., "Automatic Generation of Code Generators", Ph.D. Dissertation, Yale University, 1-Jul-77, Pages 1 to 53.

[FRIS76] Frieder, G., and Saal, H. J., "A Process for the Determination of Addresses in Variable Length Addressing", *Communications of the ACM*, Vol 19, No. 6, 1-Jun-76, Pages 96 to 103.

[FREG82] Freud Jr., V. R., and Guerin, J. A., "Automated Conversion of Design Data for Building the IBM 3081", *19th Design Automation Conference Proceedings*, 14-Jun-82, Pages 202 to 212.

[GAJPKK82] Gajski, D. D., Padua, D. A., Kuck, D. J., and Kuhn, R. H., "A Second Opinion on Data Flow Machines and Languages", *Computer*, 1-Feb-82, Pages 58 to 69.

[GANA80] Ganapathi, M., "Retargetable Code Generation and Optimization using Attribute Grammars", University of Wisconsin Technical Report, 1-Dec-80, Pages 1 to 133.

[GANAF85] Ganapathi, M., and Fischer, C. N., "Affix Grammar Driven Code Generation", *ACM Transactions on Programming Languages and Systems*, Vol. 7, No. 4, 1-Oct-85, Pages 560 to 599.

[GANAF84] Ganapathi, M., and Fischer, C. N., "Attributed Linear Representations for Retargetable Code Generators", *Software - Practice and Experience*, Vol. 14, No. 4, 1-Apr-84, Pages 347 to 364.

[GANAF81A] Ganapathi, M., and Fischer, C. N., "A Review of Automatic Code Generation Techniques", University of Wisconsin Technical Report, 1981, Pages 1 to 33.

[GANAF81B] Ganapathi, M., and Fischer, C. N., "Bibliography on Automated Retargetable Code Generation", *SIGPLAN Notices*, Vol. 16, No. 10, 1-Oct-81, Pages 9 to 12.

[GANAFH82] Ganapathi, M., Fischer, C. N., and Hennessy, J. L., "Retargetable Compiler Code Generation", *Computing Surveys*, Vol. 14, No. 4, 1-Dec-82, Pages 573 to 592.

[GANRW77] Ganzinger, H., Ripken, K., and Wilhelm, R., "Automatic Generation of Optimizing Multipass Compilers", *Information Processing 1977*, Elsevier North-Holland, New York, New York.

[GEO79] Georgeff, M., "A Framework for Control in Production Systems", *IJCAI*, William Kaufmann, Los Altos, California, 1-Aug-79, Pages 328 to 334, also found in: Stanford Artificial Intelligence Laboratory Memo, No. 322, 1-Jan-79.

[GER78] Gerhart, S. L., "Program Verification in the 1980's", *Proceedings of the Conference on Computing in the 1980's*, IEEE, Pages 80 to 89.

[GIES82] Gieser, J. L., and Sheraga, R. J., "Microarchitecture Description Techniques", *15th Annual Microprogramming Workshop*, ACM, 5-Oct-82, Pages 23 to 32.

[GLA77] Glanville, R. S., "A Machine Independent Algorithm for Code Generation and Its Use in Retargetable Compilers", Ph.D. Dissertation, 1-Dec-77, University of California at Berkeley.

[GLAG78] Glanville, R. S., and Graham, S. L., "A New Method for Compiler Code Generation", *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, 23-Jan-78, Pages 231 to 240.

[GOO85] Good, D. I., "Mechanical Proofs about Computer Programs", in: *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, (eds.), 1985, Pages 55 to 74.

[GOR81] Gordon, M. J. C., "Register Transfer Systems and their Behavior", in: *Computer Hardware Descriptions and their Applications*, M. Breuer and R. Hartenstein, (eds.), 7-Sep-81, Pages 23 to 36.

[GRA75] Graham, R. M., *Principles of Systems Programming*, John Wiley and Sons, New York, New York, 1975, Pages 1 to 422.

[GRA80] Graham, S. L., "Table Driven Code Generation", *IEEE Computer*, Vol. 13, No. 8, 1-Aug-80, Pages 25 to 34.

[GRAHS82] Graham, S. L., Henry, R. R., and Schulman, R. A., "An Experiment in Table Driven Code Generation", *ACM Sigplan Notices*, Vol. 17, No. 6, 23-Jun-82, Pages 32 to 42.

[GRAM70] Grasselli, A., and Montanari, U., "On the Minimization of Read-only Memories in Microprogrammed Digital Computers", *IEEE Transactions on Computers*, 1-Nov-70, Pages 1111 to 1114.

[GRA81] Gray, J. P., (ed.), VLSI 81 Very Large Scale Integration", Academic Press, New York, New York, Pages 1 to 363.

[GRE79] Greenwood, S. R., "Macro: A Programming Language", *SIGPLAN Notices*, Vol. 14, No. 12, 1-Dec-79, Pages 80 to 91.

[GRI83] Gresse, C., "Automatic Programming from Data Types Decomposition Patterns", *IJCAI*, William Kaufmann, Los Altos, California, 8-Aug-83, Pages 37 to 39.

[GRI71] Gries, D., *Compiler Construction for Digital Computers*, John Wiley and Sons, New York, New York, 1971, Pages 1 to 493.

[GOR79] Gordon, M. J. C., *The Denotational Description of Programming Languages An Introduction*, Springer-Verlag, New York, New York, 1979, Pages 1 to 160.

[GUI83] Guiho, G., "Automatic Programming using Abstract Data Types", *IJCAI*, William Kaufmann, Los Altos, California, 8-Aug-83, Pages 1 to 9.

[GUPT83] Gupta, A., and Toong, H. D., "An Architectural Comparison of 32-Bit Microprocessors", *IEEE Micro*, 1-Feb-83, Pages 9 to 22.

[HAPA81] Hafer, L., and Parker, A. C., "A Formal Method for the Specification, Analysis, and Design of a Register-Transfer Level Digital Logic", *18th Design Automation Conference Proceedings*, 29-Jun-81, Pages 846 to 853.

[HAPA78] Hafer, L. J., and Parker, A., "Register-Transfer Level Digital Design Automation: the Allocation Process", *15th Design Automation Conference Proceedings*, 19-Jun-78, Pages 213 to 219.

[HANLMMP82] Hansen, P. M., Linton, M. A., Mayo, R. N., Murphy, M., and Patterson, D. A., "A Performance Evaluation of the Intel IAPX 432", *Computer Architecture News*, Vol. 10, No. 4, 1-Jun-82, Pages 17 to 26.

[HASJ79] Hashizume, B., and Johnson, W. N., "The LSI-11/23 Control Store Microarchitecture", *Proceedings of the Fall COMPCON Conference*, 1979.

[HAY84] Hayes-Roth, F., "The Knowledge-Based Expert System: A Tutorial", *Computer*, 1-Sep-84, Pages 11 to 28.

[HAYWL83] Hayes-Roth, F., Waterman, D. A., Lenat, D. B., (eds.), *Building Expert Systems*, Addison-Wesley, Reading, Massachusetts, 1983, Pages 1 to 444.

[HAYWL78] Hayes-Roth, F, Waterman, D. A., and Lenat, D. B., "Principles of Pattern-Directed Inference Systems", in: *Pattern Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (eds.), Pages 577 to 602.

[HEC77] Hecht, M. S., *Flow Analysis of Computer Programs*, Elsevier North-Holland, New York, New York, 1977.

[HEN84] Henry, R. R., "Graham-Glanville Code Generators", Ph.D. Dissertation, University of California at Berkeley, 1-May-84, Pages 1 to 280.

[HENMA83] Henry, S. G., Mueller, R. A., and Andrews, M., "Local and Global Microcode Compaction Using Reduction Operators", *16th Annual Microprogramming Workshop*, ACM, 11-Oct-83, Pages 34 to 43.

[HITT83] Hitchcock III, C. Y., and Thomas, D. E., "A Method of Automatic Data Path Synthesis", *20th Design Automation Conference Proceedings*, 27-Jun-83, Pages 484 to 489.

[HILP73] Hill, F.J., and Peterson, G. R., *Digital Systems: Hardware Organization and Design*, John Wiley and Sons, New York, New York, 1973, Pages 1 to 481.

[HILSMN81] Hill, F. J., Swanson, R. E., Masud, M., and Navabi, Z., "Structure Specification with a Procedural Hardware Description Language", *IEEE Transactions on Computers*, Vol. 30, No. 2, 1-Jan-81, Pages 157 to 161.

[HOAS85] Hoare, C. A. R., and Shepherdson, J. C., (eds.), *Mathematical Logic and Programming Languages*, Prentice-Hall, Englewood Cliffs, New Jersey, 1985, Pages 1 to 184.

[HOPHA85] Hopkins, W. C., Horton, M. J., and Arnold, C. S., "Target-Independent High-Level Microprogramming", *18th Annual Microprogramming Workshop*, ACM, 3-Dec-85, Pages 137 to 143.

[HUEO80] Huet, G., and Oppen, D. C., "Equations and Rewrite Rules: A Survey", SRI Technical Report, 1-Jan-80, Pages 1 to 52.

[HUL80] Hullot, J. M., "A Catalogue of Canonical Term Rewriting Systems", SRI Technical Report, 1-Apr-80, Pages 1 to 33.

[HUS70] Husson, S. S., *Microprogramming Principles and Practices*, Prentice-Hall, Englewood Cliffs, New Jersey, 1970, Pages 1 to 614.

[HWA79] Hwang, K., *Computer Arithmetic Principles, Architecture, and Design*, John Wiley and Sons, New York, New York, 1979, Pages 1 to 423.

[JEFF80] Jefferson, D. R., "Type Reduction and Program Verification", Ph.D. Dissertation, Carnegie-Mellon University, 1-Apr-80, Pages 1 to 418.

[JET79] Jette, C. L., "Heuristic Control of Design-directed Program Transformations", *AFIPS*, Vol. 48, 1979, Pages 1071 to 1077.

[JOH78] Johnson, S. C., "A Portable Compiler Theory and Practice", *Proceedings of the 5th ACM Symposium on Principles of Programming Languages*, 23-Jan-78, Pages 97 to 104.

[JON80] Jones, N. D., (ed.), *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science, Springer-Verlag, New York, New York, No. 94, 14-Jan-80, Pages 1 to 489.

[JONS80] Jones, N. D., and Schmidt, D. A., "Compiler Generation from Denotational Semantics", in: *Semantics-Directed Compiler Generation*, Jones, N. D. (ed.), 14-Jan-80, Pages 70 to 93.

[JOYCL76] Joyner, W. H., Carter, W. C., and Leeman, G. B., "Automated Proofs of Microprogram Correctness", *Proceedings of the 9th Annual Workshop on Microprogramming*, 1-Sep-76, Pages 51 to 56.

[KAN85] Kant, E., "Understanding and Automating Algorithm Design", *IJCAI*, William Kaufmann, Los Altos, California, 18-Aug-85, Pages 1243 to 1253, also in: *IEEE Transactions on Software Engineering*, Vol. 11, No. 11, 1-Nov-85, Pages 1361 to 1374.

[KAN79A] Kant, E., "Efficiency Considerations in Program Synthesis: A Knowledge-Based Approach", Ph.D. Dissertation, Stanford Artificial Intelligence Laboratory Memo, No. 331, 1-Sep-79, Pages 1 to 155.

[KAN79B] Kant, E., "A Knowledge-Based Approach to Using Efficiency Estimation in Program Synthesis", *IJCAI*, William Kaufmann, Los Altos, California, 1-Aug-79, Pages 457 to 462.

[KAN77] Kant, E., "The Selection of Efficient Implementations for a High Level Language", *Proceedings of the ACM SIGART-SIGPLAN Symposium on Artificial Intelligence and Programming Languages*, 1-Aug-77, Pages 140 to 146.

[KANB81] Kant, E., and Barstow, D. R., "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis", *IEEE Transactions on Software Engineering*, Vol. 7, No. 5, 1-Sep-81, Pages 458 to 471.

[KANN83] Kant, E., and Newell, A., "An Automatic Algorithm Designer: An Initial Implementation", *AAAI*, William Kaufmann, Los Altos, California, 22-Aug-83, Pages 177 to 181.

[KIB78] Kibler, D. F., "Power, Efficiency, and Correctness of Transformation Systems", Ph.D. Dissertation, Computer Science Department, University of California at Irvine, 1978, Pages 1 to 205.

[KIBNS77] Kibler, D. F., Neighbors, J. M., and Standish, T. A., "Program Manipulation via an Efficient Production System", Proceedings of the Symposium on Artificial Intelligence and Programming Languages, *SIGPLAN Notices*, Vol. 12, 1977, Pages 163 to 173.

[KIDD81] Kidder, T., *The Soul of a New Machine*, Alantic-Little, Brown and Company, 1981, Pages 1 to 293.

[KIN83] King, M., *Parsing Natural Language*, Academic Press, New York, New York, 1983, Pages 1 to 308.

[KLAD81] Klassen, A., and Dasgupta, S., "S*(QM-1): An Instantiation of the High Level Microprogramming Language Schema S* for the Nanodata QM-1", *Proceedings of the 14th Annual Microprogramming Workshop*, ACM, 12-Oct-81, Pages 124 to 130.

[KLER71] Kleir, R. L., and Ramamoorthy, C. V., "Optimization Strategies for Microprograms", *IEEE Transactions on Computers*, Vol. 20, No. 7, 1-Jul-71, Pages 783 to 794.

[KNUB70] Knuth, D. E., and Bendix, P., "Simple Word Problems in Universal Algebras", in J. Leech, (ed.), *Computational Problems in Abstract Algebra*, Oxford, England, 1970, Pages 263 to 297.

[KOG81] Kogge, P. M., *The Architecture of Pipelined Computers*, McGraw-Hill, New York, New York, 1981, Pages 1 to 334.

[KOW79] Kowalski, R., *Logic for Problem Solving*, Elsevier North-Holland, New York, New York, 1979, Pages 1 to 287.

[KOWT85] Kowalski, T. J., and Thomas, D. E., "The VLSI Design Automation Assistant: Prototype System", *22nd Design Automation Conference Proceedings*, 23-Jun-85, Pages 252 to 258.

[KOWT83] Kowalski, T. J., and Thomas, D. E., "The VLSI Design Automation Assistant: Prototype System", *20th Design Automation Conference Proceedings*, 27-Jun-83, Pages 479 to 483.

[KUC77] Kuck, D. J., "A Survey of Parallel Machine Organization and Programming", *Computing Surveys*, Vol. 9, No. 1, 1-Mar-77, Pages 29 to 59.

[KUC78] Kuck, D. J., *The Structure of Computers and Computations*, Volume 1, John Wiley and Sons, New York, New York, 1978, Pages 1 to 611.

[KUCBC74] Kuck, D. J. Budnik, P. P. Chen, S. C. Lawrie, D. H., Towle, R. A., Strebendt, R. E., Davis, E. W., and Han, J., "Measurements of Parallelism in Ordinary FORTRAN Programs", *Computer*, 1974, Pages 37 to 46.

[KUCMC72] Kuck, D. J., Muraoka, Y., and Chen S., "On the Number of Operations Simultaneously Executable in FORTRAN-like Programs and Their Resulting Speedup", *IEEE Transactions on Computers*, Vol. 21, No. 12, 1-Dec-72, Pages 1293 to 1309.

[LANDSM80] Landskov, D., Davidson, S., Shriver, B., and Mallett, P. W., "Local Microcode Compaction Techniques", *Computing Surveys*, Vol. 12, No. 3, 1-Sep-80, Pages 261 to 294.

[LAR82] Larus, J. R., "A Comparison of Microcode, Assembly Code, High-Level Languages on the VAX-11 and RISC I", *Computer Architecture News*, Vol. 10, No. 5, 1-Sep-82, Pages 10 to 15.

[LEIT81] Leive, G. W., and Thomas, D. E., "A Technology Relative Logic Synthesis and Module Selection System", *18th Design Automation Conference Proceedings*, 29-Jun-81, Pages 479 to 485.

[LEVS80] Leverett, B., Szymanski, T., "Chaining Span-Dependent Jump Instructions", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 3, 1-Jul-80, Pages 274 to 289.

[LEV82] Leverett, B. W., "Topics in Code Generation and Register Allocation", Carnegie-Mellon University Technical Report, 28-Jul-82, Pages 1 to 25.

[LEVCHNRSW79] Leverett, B. W., Cattell, R. G., Hobbs, S. O., Newcomer, J. M., Reiner, A. H., Schatz, B. R., Wulf, W. A., "An Overview of the Production Quality Compiler-compiler Project", Carnegie-Mellon University Technical Report, 1-Feb-79, Pages 1 to 60.

[LEV84] Levy, B., "Microcode Verification Using SDVS - The Method and a Case Study", *17th Annual Microprogramming Workshop*, ACM, 30-Oct-84, Pages 234 to 245.

[LEVYE80] Levy, H. M., Eckhouse, R. H., *Computer Programming and Architecture the VAX-11*, Digital Press, Bedford, Massachusetts, 1-Apr-80, Pages 1 to 407.

[LEWS81] Lewis, T. G., and Shriver, B. D., "Introduction to Special Issue on Microprogramming Tools and Techniques", *IEEE Transactions on Computers*, Vol. 30, No. 7, 1-Jul-81, Pages 457 to 459.

[LIN83] Linn, J. L., "SRDAG Compaction - A Generalization of Trace Scheduling to Increase the Use of Global Context Information", *16th Annual Microprogramming Workshop*, ACM, 11-Oct-83, Pages 11 to 22.

[LIP78] Lipovski, G. J., "Hardware Description Languages", in: *Encyclopedia of Computer Science and Technology*, Marcel Dekker, Inc., New York, New York, 1978.

[LIP83] Lipp, H. M., "Methodical Aspects of Logic Synthesis", *Proceedings of the IEEE*, Vol. 71, No. 1, 1983, Pages 88 to 97.

[LLO74] Lloyd, G. R., "PUMPKIN - (Another Microprogramming Language", *SIGMICRO Newsletter*, Vol. 5, 1-Apr-74, Pages 15 to 44.

[LON79] London, R. L., "Program Verification", in: *Research Directions in Software Technology*, P. Wegner, (ed.), 1979, Pages 302 to 315.

[LOV77] Loveman, D. B., "Program Improvement by Source-to-Source Transformation", *Journal of the Association for Computing Machinery*, Vol. 24, No. 1, 1-Jan-77, Pages 121 to 145.

[MA78] Ma, P. R., "Optimizing the Microcode Produced by a High Level Microprogramming Language", Ph.D. Dissertation, Oregon State University, 1-Oct-78.

[MAL81] Ma, P. R., and Lewis, T. G., "On the Design of a Microcode Compiler for a Machine-Independent High-Level Language", *IEEE Transactions on Software Engineering*, Vol. 7, No. 3, 1-May-81, Pages 261 to 274.

[MAL80] Ma, P. R., and Lewis, T. G., "Design of a Machine-Independent Optimizing System for Emulator Development", *ACM Transactions on Programming Languages and Systems*, Vol. 2, No. 2, 1-Apr-80, Pages 239 to 262.

[MAL78] Mallett, P. W., "Methods of Compacting Microprograms", Ph.D. Dissertation, University of Southern Louisiana, 1-Dec-78, Pages 1 to 175.

[MALL75] Mallett, P. W., and Lewis, T. G., "Considerations for Implementing a High Level Microprogramming Language Translation System", *Computer*, 1-Aug-75, Pages 40 to 52.

[MANW85] Manna, Z., and Waldinger, R., *The Logical Basis for Computer Programming Volume 1: Deductive Reasoning*, Addison-Wesley, Reading, Massachusetts, 1985, Pages 1 to 618.

[MANW77] Manna, Z., and Waldinger, R., *Studies in Automatic Programming Logic*, Elsevier North-Holland, New York, New York, 1977, Pages 1 to 192.

[MARCL84] Marcus, L., Crocker, S. D., Landauer, J. R., "SDVS: A System for Verifying Microcode Correctness", *17th Annual Microprogramming Workshop*, ACM, 30-Oct-84, Pages 246 to 255.

[MAR84] Marwedel, P., "A Retargetable Compiler for a High-Level Microprogramming Language", *Proceedings of the 17th Annual Microprogramming Workshop*, ACM, 30-Oct-84, Pages 267 to 274.

[MAR82] Marwedel, P., "The Integrated Design of Computer Systems with MIMOLA", *Informatik Fachberichte*, 1982, Pages 45 to 57.

[MAR81A] Marwedel, P., "A Retargetable Microcode Generation System for a High-Level Microprogramming Language", *Proceedings of the 14th Annual Microprogramming Workshop*, ACM, 12-Oct-81, Pages 115 to 123.

[MAR81B] Marwedel, P., "Statistical Studies of Horizontal Microprograms", in: *Computer Hardware Descriptions and their Applications*, M. Breuer and R. Hartenstein, (eds.), 7-Sep-81, Pages 281 to 192.

[MAR80A] Marwedel, P., "Hardware Allocation for Horizontal Microinstructions in the MIMOLA Software System", Christian-Albrechts-Universitat Technical Report, 1-Aug-80, Pages 1 to 32.

[MAR80B] Marwedel, P., "The Design of a Subprocessor with Dynamic Microprogramming", *Informatik Fachberichte*, 1980, Pages 164 to 177.

[MAR79] Marwedel, P., "The MIMOLA Design System: Detailed Description of the Software System", *16th Design Automation Conference Proceedings*, 25-Jun-79, Pages 59 to 63.

[MARZ79] Marwedel, P., Zimmermann, G., "MIMOLA Report and MIMOLA Software System User Manual", Christian-Albrechts-Universitat Technical Report, 1-May-79, Pages 1 to 113.

[MCDF78] McDermott, J., and Forgy, C., "Production System Conflict Resolution Strategies", in: *Pattern Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (eds.), Pages 177 to 202.

[MCDNM78] McDermott, J., Newell, A., and Moore, J., "The Efficiency of Certain Production System Implementations", in: *Pattern Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (eds.), Pages 155 to 176.

[MILL71] Miller, P. L., "Automatic Creation of a Code Generator from a Machine Description", M.S. Thesis, Massachusetts Institute of Technology, 1971.

[MIL85] Milner, R., "The Use of Machines to Assist in Rigorous Proof", in: *Mathematical Logic and Programming Languages*, C. A. R. Hoare and J. C. Shepherdson, (eds.), 1985, Pages 77 to 87.

[MIZ83] Mizoguchi, F., "Prolog Based Expert System", *New Generation Computing*, Vol. 1, No. 1, 1983, Pages 99 to 104.

[MONT74] Montangero, C., "An Approach to the Optimal Specification of Read-only Memories in Microprogrammed Digital Computers", *IEEE Transactions on Computers*, Vol. 23, No. 4, 1-Apr-74, Pages 375 to 389.

[MUCJ81] Muchnick, S. S., Jones, N. D., (eds.), *Program Flow Analysis Theory and Applications*, 1981, Prentice-Hall, Englewood Cliffs, New Jersey, Pages 1 to 418.

[MUE80A] Mueller, R. A., "Automated Microprogram Synthesis", Ph.D. Dissertation, Colorado State University, 1980, Pages 1 to 185, reprinted by UMI Research Press, Ann Arbor, Michigan, 1984.

[MUE80B] Mueller, R. A., "Formalization and Automated Synthesis of Microprograms", *Proceedings of the 13th Annual Microprogramming Workshop*, ACM, 30-Nov-80, Pages 45 to 53.

[MUEDO84] Mueller, R. A., Duda, M. R., and O'Haire, S. M., "A Survey of Resource Allocation Methods in Optimizing Microcode Compilers", *17th Annual Microprogramming Workshop*, ACM, 30-Oct-84, Pages 285 to 295.

[MUEJ81] Mueller, R. A., and Johnson, G. R., "Contrasting Translation, Verification, and Synthesis in Software and Firmware Engineering", *Proceedings of the 14th Annual Microprogramming Workshop*, ACM, 12-Oct-81, Pages 17 to 22.

[MUEV83] Mueller, R. A., and Varghese, J., "Flow Graph Machine Models in Microcode Synthesis", *16th Annual Microprogramming Workshop*, ACM, 11-Oct-83, Pages 159 to 167.

[MUEV82] Mueller, R. A., and Varghese, J., "Formal Semantics for the Automated Derivation of Micro-code", *19th Design Automation Conference Proceedings*, 14-Jun-82, Pages 815 to 824.

[MUEVA84] Mueller, R. A., Varghese, J., and Allan, V. H., "Global Methods in the Flow Graph Approach to Retargetable Microcode Generation", *17th Annual Microprogramming Workshop*, ACM, 30-Oct-84, Pages 275 to 284.

[NAGLE81] Nagle, A. W., "Automated Design of Digital-system Control Sequencers from Register-transfer Specifications", Ph.D. Dissertation, Carnegie-Mellon University Electrical Engineering Department, 1981, Pages 1 to 150.

[NAGLE78] Nagle, A. W., "Automatic Synthesis of Microcontrollers", *Proceedings of the 11th Annual Microprogramming Workshop*, ACM, 1-Nov-78, Pages 112 to 117.

[NAGP81] Nagle, A. W., and Parker, A. C., "Algorithms for Multiple-Criterion Design of Microprogrammed Control Hardware", *18th Design Automation Conference Proceedings*, 29-Jun-81, Pages 486 to 493.

[NEW75] Newcomer, J. M., "Machine Independent Generation of Optimized Local Code", Ph.D. Dissertation, Carnegie-Mellon University, 1975.

[NEWC80] Newcomer, J. M., Personal Communication, 1980.

[NIL80] Nilsson, N. N., *Principles of Artificial Intelligence*, Tioga Publishing Co., Palo Alto, California, 1980, Pages 1 to 476.

[OAK79] Oakley, J. D., "The Symbolic Execution of Formal Machine Descriptions", Ph.D. Dissertation, Department of Computer Science, Carnegie-Mellon University, 1-Apr-79, Pages 1 to 144.

[ORGA83] Organick, E. I., *A Programmer's View of the Intel 432 System*, McGraw-Hill, New York, New York, 1983, Pages 1 to 418.

[OKE83] O'Keefe, R. A., "PROLOG Compared to LISP?", *SIGPLAN Notices*, Vol. 18, No. 5, 1-May-83, Pages 46 to 56.

[PAG81] Pagan, F. G., *Formal Specification of Programming Languages, a Panoramic Primer*, Prentice-Hall, Englewood Cliffs, New Jersey, 1981, Pages 1 to 245.

[PAR75] Parker, A. C., "A Generalized Approach to Digital Interfacing", Ph.D. Dissertation, North Carolina State University, 1975, Pages 1 to 257.

[PAHA81] Parker, A. C., Hafer, L. J., "Automating the Design of Testable Hardware", in: *VLSI 81 Very Large Scale Integration*, J. P. Gray (ed.), Academic Press, New York, New York, 18-Aug-81,Pages 357 to 363.

[PARH79] Parker, A. C., Hafer, L., "Automated Synthesis of Digital Hardware", Carnegie-Mellon University Technical Report, 1-May-79, Pages 1 to 54.

[PTSBHLK79] Parker, A., Thomas, D., Siewiorek, D., Barbacci, M., Hafer, L., Leive, G., Kim, J,. "The CMU Design Automation System An Example of Automated Data Path Design", *16th Design Automation Conference Proceedings*, 25-Jun-79, Pages 73 to 80.

[PARTCC79] Parker, A. C., Thomas, D. E., Crocker, S., Cattell, R., G. G., "ISPS: A Retrospective View", *4th International Symposium on Computer Hardware Description Languages*, 8-Oct-79, Pages 33 to 39.

[PARW81] Parker, A. C., Wallace, J. J., "SLIDE: An I/O Hardware Description Language", *IEEE Transactions on Computers*, Vol. 10, No. 6, 1-Jun-81, Pages 423 to 439.

[PAT83] Patterson, D. A., "Microprogramming", *Scientific American*, 1-Mar-83, Pages 50 to 57.

[PAT82] Patterson, D. A., "A Performance Evaluation of the Intel 80286", *Computer Architecture News*, Vol. 10, No. 5, 1-Sep-82, Pages 16 to 18.

[PAT81] Patterson, D. A., "An Experiment in High Level Language Microprogramming and Verification", *Communications of the ACM*, Vol. 24, No. 10, 1-Oct-81, Pages 699 to 709.

[PAT78] Patterson, D. A., "An Approach to Firmware Engineering", Proceedings of the NCC, 1-Jun-78, Pages 643 to 647.

[PAT77] Patterson, D. A., "Verification of Microprograms", Ph.D. Dissertation, University of California at Los Angles, 1977, Pages 1 to 297.

[PAT76] Patterson, D. A., "Strum: Structured Microprogram Development System for Correct Firmware", *IEEE Transactions on Computers*, Vol. 25, No. 10, 1-Oct-76, Pages 974 to 985.

[PATD80] Patterson, D. A., Ditzel, D. R., "The Case for the Reduced Instruction Set Computer", *Computer Architecture News*, Vol. 8, No. 6, 15-Oct-80, Pages 25 to 33.

[PATGPS81] Patterson, D., Goodell, R., Poe, M. D., and Steely, S., "V-Compiler: A Next Generation Tool for Microprogramming", *Proceedings NCC*, 1981, Pages 103 to 109.

[PATLT79] Patterson, D., Lew, K., and Tuck, R., "Towards an Efficient Machine-Independent Language for Microprogramming", *12th Annual Microprogramming Workshop*, ACM, 1979, Pages 22 to 35.

[PATP82] Patterson, D. A., Piepho, R. S., "RISC Assessment: A High-level Language Experiment", *9th Annual Symposium on Computer Architecture*, 26-Apr-8, Pages 3 to 8.

[PATS82] Patterson, D. A., Sequin, C. H., "A VLSI RISC", *Computer*, 1-Sep-82, Pages 8 to 21.

[PATS81] Patterson, D. A., Sequin, C. H., "RISC I: A Reduced Instruction Set VLSI Computer", *8th Annual Symposium on Computer Architecture*, 12-May-81, Pages 443 to 457.

[PEA84] Pearl, J., *Heuristics Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, Reading, Massachusetts, 1984, Pages 1 to 382.

[PER84] Pereira, F. C., (ed.), "C-Prolog User's Manual Version 1.5", Technical Report, University of Edinburgh Department of Architecture, 1-Feb-84, Pages 1 to 37.

[PERW80] Pereira, F. C., Warren, D. H., "Definite Clause Grammars for Language Analysis - A Survey of the Formalism and a Comparison with Augmented Transition Networks", *Artificial Intelligence*, Vol. 13, 1980, Pages 231 to 278.

[PER79] Pereira, L. M., "Backtracking Intelligently in And/or Tree"s, Universidade Nova de Lisboa Technical Report, 1-Jun-79, Pages 1 to 25.

[PERP79A] Pereira, L. M., Porto, A., "Intelligent Backtracking and Sidetracking in Horn Clause Programs the Theory", Universidade Nova de Lisboa Technical Report, No. 2, 1-Oct-79, Pages 1 to 66.

[PERP79B] Pereira, L. M., Porto, A., "Intelligent Backtracking and Sidetracking in Horn Clause Programs the Implementation", Universidade Nova de Lisboa Technical Report, No. 13, 1-Dec-79, Pages 1 to 42.

[PERP82] Pereira, L. M., Porto, A., "Selective Backtracking", In: *Logic Programming*, Clark and Tarnlund, (eds.), 1982, Pages 107 to 116.

[PETS81] Peterson, G. E., Stickel, M. E., "Complete Sets of Reductions for Some Equational Theories", *Journal of the ACM*, Vol. 28, No. 2, 1-Apr-81, Pages 233 to 264.

[PILB82] Piloty, R., and Borrione, D., "The CONLAN Project: Status and Future Plans", *19th Design Automation Conference Proceedings*, 14-Jun-82, Pages 202 to 212.

[POE84] Poe, M. D., "Control of Heuristic Search in a PROLOG-based Microcode Synthesis Expert System", *International Conference on Fifth Generation Computer Systems*, 6-Nov-84, Pages 589 to 595.

[POE80] Poe, M. D., "Heuristics for the Global Optimization of Microprograms", *Proceedings of the 13th Annual Microprogramming Workshop*, ACM, 30-Nov-80, Pages 13 to 22.

[POE81A] Poe, M. D., Goodell, R., Steely, S., "Issues of the Design of a Low Level Microprogramming Language for Global Microcode Compaction", *Proceedings of the 14th Annual Microprogramming Workshop*, ACM, 12-Oct-81, Pages 88 to 94.

[POE81B] Poe, M. D., "Measurement and Manipulation of Potential Parallelism in Microcode", 8-Sep-81, Euromicro81, in: *Implementing Functions*, by L. Richer, P. Le Beux, G. Chroust, G. Noguez, (eds.), Pages 351 to 360.

[POENPS84] Poe, M. D., Nasr, R., Potter, J., Slinn, J., "A KWIC (Key Word In Context) Bibliography on PROLOG and Logic Programming", *Journal of Logic Programming*, Vol. 1, No. 1, 1-Jun-84, Pages 81 to 142.

[PRO84] Proulx, D. M., "Applications of Pipelining to Firmware", *Proceedings of the 17th Annual Microprogramming Workshop*, ACM, 30-Oct-84, Pages 37 to 46.

[RADI82] Radin, G., "The 801 Minicomputer", *Proceedings of the Symposium on Architectural Support for Programming Languages and Operating Systems*, 1-Mar-82, Pages 39 to 47.

[RAJT85] Rajan, J. V., and Thomas, D. E., "Synthesis By Delayed Binding of Decisions", *22nd Design Automation Conference Proceedings*, ACM, 23-Jun-85, Pages 367 to 373.

[RAML77] Ramamoorthy, C. V., and Li, H. F., "Pipeline Architecture", *ACM Computing Surveys*, Vol. 9, No. 1, 1-Mar-77, Pages 61 to 102.

[RIC83] Rich, E., *Artificial Intelligence*, McGraw-Hill, New York, New York, 1983, Pages 1 to 436.

[RICLCN81] Richer, L., Le Beux, P., Chroust, G., Noguez, G., (eds.), *Implementing Functions*, Elsevier North-Holland, New York, New York, 8-Sep-81, Pages 1 to 488.

[RIP75] Ripken, K., "Generating an Intermediate-Code Generator in a Compiler-Writing System", *International Computing Symposium 1975*, Elsevier North-Holland, New York, New York, 1975, Pages 121 to 127.

[ROBE79A] Robertson, E. L., "Code Generation and Storage Allocation for Machines with Span-Dependent Instructions", *ACM Transactions on Programming Languages and Systems*, Vol. 1, No. 1, 1-Jul-79, Pages 71 to 83.

[ROBE79] Robertson, E. L., "Microcode Bit Optimization is NP Complete", *IEEE Transactions on Computers*, Vol. 28, No. 4, 1-Apr-79, Pages 316 to 319.

[ROB79] Robinson, J. A., *Logic: Form and Function The Mechanization of Deductive Reasoning*, Elsevier North-Holland, New York, New York, 1979, Pages 1 to 312.

[ROS83] Rosner, M., "Production Systems", in: *Parsing Natural Language* by M. King, 1983, Pages 35 to 58.

[RUBI82] Rubinfeld, R. I., "Two-chip Supermicroprocessor Outperforms PDP-11 Minicomputers", *Electronics*, Vol. 55, No. 25, 15-Dec-82, Pages 131 to 136.

[RUS80] Russell, B., "Correctness of the Compiling Process Based on Axiomatic Semantics", *Acta Informatica*, Springer-Verlag, New York, New York, 1980, Pages 1 to 20.

[SAC77] Sacerdoti, E. D., *A Structure for Plans and Behavior*, Elsevier North-Holland, New York, New York, 1977, Pages 1 to 126.

[SAL76] Salisbury, A. B., *Microprogrammable Computer Architectures*, Elsevier North-Holland, New York, New York, 1976, Pages 1 to 161.

[SAMK78] Sameh, A. H., Kuck, D. J., "On Stable Parallel Linear System Solvers", *Journal of the ACM*, Vol. 25, No. 1, 1978, Pages 81 to 91.

[SCH78] Schwartz, J. T., "A Survey of Program Proof Technology", Office of Naval Research, 1-Sep-78, Pages 1 to 33.

[SCHW68] Schwartz, S. J., "An Algorithm for Minimizing Read Only Memories for Machine Control", *IEEE Symposium on Switching and Automata Theory*, 1968, Pages 28 to 33.

[SET75] Sethi, R., "Complete Register Allocation Problems", *SIAM Journal of Computing*, Vol. 4, No. 3, 1-Sep-75, Pages 226 to 248.

[SHA74] Shaw, A. C., *The Logical Design of Operating Systems*, Prentice-Hall, Englewood Cliffs, New Jersey, 1974, Pages 1 to 306.

[SHIS83] Shimizu, T., and Sakamura, K., "MIXER: An Expert System for Microprogramming", *16th Annual Microprogramming Workshop*, ACM, 11-Oct-83, Pages 168 to 178.

[SHI83] Shiva, S. G., "Automatic Hardware Synthesis", *Proceedings of the IEEE*, Vol. 71, No. 1, 1983, Pages 76 to 87.

[SHI79] Shiva, S. G., "Computer Hardware Description Languages - A Tutorial", *Proceedings of the IEEE*, Vol. 67, No. 12, 1-Dec-79, Pages 1605 to 1615.

[SHIC82] Shiva, S. G., and Covington, J. A., "Modular Description/Simulation/ Synthesis Using DDL", *19th Design Automation Conference Proceedings*, 14-Jun-82, Pages 321 to 329.

[SHRO82] Shrobe, H. E., "The Data Path Generator", *Conference on Advanced Research in VLSI*, Massachusetts Institute of Technology, 27-Jan-82, Pages 175 to 181.

[SHRO79] Shrobe, H. E., "Explicit Control of Reasoning in the Programmer's Apprentice", *4th Workshop on Automated Deduction*, 1-Feb-79, Pages 97 to 102.

[SIEBAR76] Siewiorek, D. P., Barbacci M. R., "The CMU RT-CAD System - An Innovative Approach to Computer Aided Design", *Proceedings of the National Computer Conference*, 1976, Pages 643 to 655.

[SIEBN82] Siewiorek, D. P., Bell, C. G., Newell, A., *Computer Structures: Principles and Examples*, McGraw-Hill, New York, New York, 1982, Pages 1 to 926.

[SIE74A] Siewiorek, D. P., "Introducing ISP", *Computer*, IEEE, Vol. 7, No. 12, 1-Dec-74, Pages 39 to 41.

[SIE74B] Siewiorek, D. P., "Introducing PMS", *Computer*, IEEE, Vol. 7, No. 12, 1-Dec-74, Pages 42 to 44.

[SINGT81] Singh, A. K., and Tracey, J. H., "Development of Comparison Features for Computer Hardware Description Languages", in: *Computer Hardware Descriptions and their Applications*, M. Breuer and R. Hartenstein, (eds.), 7-Sep-81, Pages 247 to 263.

[SINT81] Sint, M., "MIDL - A Microinstruction Description Language", *Proceedings of the 14th Annual Microprogramming Workshop*, ACM, 12-Oct-81, Pages 95 to 106.

[SINT80] Sint, M., "A Survey of High Level Microprogramming Languages", *Proceedings of the 13th Annual Microprogramming Workshop*, ACM, 30-Nov-80, Pages 141 to 153.

[SMI83] Smith, D. R., "A Problem Reduction Approach to Program Synthesis", *IJCAI*, William Kaufmann, Los Altos, California, 8-Aug-83, Pages 32 to 36.

[SMIS79] Smith, D. A., and Standish, T. A., "Research on Interactive Program Manipulation Final Report", University of California at Irvine Technical Report, No. 146, 1-Dec-79, Pages 1 to 60.

[SNOW78] Snow, E. A., "Automation of Module Set Independent Register-Transfer Level Design", Ph.D. Dissertation, Electrical Engineering Department, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1978.

[SNSITH78] Snow, E. A., Siewiorek, D. P., Thomas D. E., "A Technology-Relative Computer-Aided Design System: Abstract Representations, Transformations, and Tradeoffs", *15th Design Automation Conference Proceedings*, 19-Jun-78, Pages 220 to 226.

[STAHKN76] Standish, T., Harriman, D., Kibler, D., and Neighbors, J., "The Irvine Program Transformation Catalogue", University of California at Irvine, Computer Science Department, 1-Jan-76, Pages 1 to 81.

[STAKN76] Standish, T. A., Kibler, D. F., and Neighbors, J. M., "Improving and Refining Programs by Program Manipulation", *Proceedings of the 1976 ACM Annual Conference*, 1-Oct-76, Pages 509 to 516.

[STA83] Starnes, T. W., "Design Philosophy behind Motorola's MC68000", *Byte*, Vol. 8, No. 3, 1-May-83, Pages 342 to 367.

[STEABBBHS83] Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., Sacerdoti, E., "The Architecture of Expert Systems", in: F. Hayes-Roth, D. A. Waterman, and D. B. Lenat, (eds.), *Building Expert Systems*, 1983, Pages 89 to 126.

[STEABBBHS82] Stefik, M., Aikins, J., Balzer, R., Benoit, J., Birnbaum, L., Hayes-Roth, F., Sacerdoti, E., "The Organization of Expert Systems: A Prescriptive Tutorial", Xerox PARC Technical Report, 1982, Pages 1 to 61.

[STEK85] Steiner, D., and Kant, E., "Symbolic Execution in Algorithm Design", *IJCAI*, William Kaufmann, Los Altos, California, 18-Aug-85, Pages 225 to 231.

[STO77] Stoy, J. E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, Cambridge, Massachusetts, 1977, Pages 1 to 414.

[STRWTOMS58] Strong, J., Wegstein, J., Tritter, A., Olsztyn, J., Mock, O., Steel, T., "The Problem of Programming Communication with Changing Machines a Proposed Solution", 1-Aug-58, *Communications of the ACM*, Vol. 1, No. 8, Pages 12 to 18.

[SWARTZL76] Swartzlander, E. E., (ed.), *Computer Design Development Principle Papers*, Hayden, Rochelle Park, New Jersey, 1976, Pages 1 to 310.

[SUD85] Su, B., and Ding, S., "Some Experiments in Global Microcode Compaction", *18th Annual Microprogramming Workshop*, ACM, 3-Dec-85, Pages 175 to 180.

[SUDJ84] Su, B., Ding, S., and Jin, L, "An Improvement of Trace Scheduling for Global Microcode Compaction", *17th Annual Microprogramming Workshop*, ACM , 30-Oct-84, Pages 78 to 85.

[SU74] Su, S. Y. H., "A Survey of Computer Hardware Description Languages in the U.S.A.", *Computer*, IEEE, Vol. 7, No. 12, 1-Dec-74, Pages 45 to 51.

[SZY78] Szymanski, T. G., "Assembling Code for Machines with Span-Dependent Instructions", *Communications of the ACM*, Vol. 21, No. 4, 1-Apr-78, Pages 300 to 308.

[TAKTBSK82] Takahashi, E., Takahashi, K., Bito, T., Sasaki, T., and Kitano, K., "Automatic Address Assignment of Horizontal Microprograms", *16th Annual Microprogramming Workshop*, ACM, 11-Oct-83, Pages 185 to 192.

[TAKTBAY82] Takahashi, K., Takahashi, E., Bito, T., Aoyama, T., and Yamada, A., "MDS: An Improved Total System for Firmware Development", *15th Annual Microprogramming Workshop*, ACM, 5-Oct-82, Pages 50 to 56.

[TANKE78] Tanaka, T., Kawada, T., and Emori, T., "Proposal on Efficient Address Allocation Algorithm for Horizontal Microprograms", *11th Annual Microprogramming Workshop*, ACM, 1978, Pages 40 to 40.

[TANVS82] Tanenbaum, A. S., Vanstaveren, H., Stevenson, J. W., "Using Peephole Optimization on Intermediate Code", *ACM Transactions on Programming Languages*, Vol. 4., No. 1, 1982, Pages 21 to 36.

[THOMAS77] Thomas, D. E., "The Design and Analysis of an Automated Design Style Selector", Ph.D. Dissertation, Department of Electrical Engineering, Carnegie-Mellon University, 1-Apr-77.

[TOKTTY78] Tokoro, M., Takizuka, T., Tamura, E., Yamaura, I., "A Technique of Global Optimization of Microprograms", *11th Annual Microprogramming Workshop*, ACM, 1-Nov-78, Pages 41 to 50.

[TOKTTT77] Tokoro, M., Tamura, E., Takase, K., and Tamaru, K., "An Approach to Micro-program Optimization Considering Resource Occupancy and Instruction Formats", *Proceedings 10th Annual Microprogramming Workshop*, ACM, 1977, Pages 92 to 100.

[TOKTT81] Tokoro, M., Tamura, E., and Takizuka, T., Optimization of Microprograms, *IEEE Transactions on Computers*, Vol. 30, No. 7, July, 1981, Pages 491 to 504.

[TSUG76] Tsuchiya, M., and Gonzalez, M. J., "Toward Optimization of Horizontal Microprograms", *IEEE Transactions on Computers*, Vol. 25, No. 10, 1-Oct-76, Pages 992 to 999.

[TSUG74] Tsuchiya, M., and Gonzalez, M. J., "An Approach to Optimization of Horizontal Microprograms", 7th Annual Microprogramming Workshop, ACM, 1974, Pages 85 to 90.

[UEHMSK81] Uehara, T., Maruyama, F., Saito, T., Kawato, N., "DDL Verifier", in: *Computer Hardware Descriptions and their Applications*, M. Breuer and R. Hartenstein, (eds.), 7-Sep-81, Pages 51 to 64.

[ULL73] Ullman, J. D., "Polynomial Complete Scheduling Problem", Fourth Symposium Operating System Principles, 1973, Pages 96 to 101, Published in *Operating System Review*, Vol. 7, No. 4, ACM, cited in [FIS79].

[ULR80] Ulrich, J. W., "The Derivation of Microcode by Symbolic Execution", *Proceedings of the 13th Annual Microprogramming Workshop*, ACM, 30-Nov-80, Pages 38 to 42.

[VANW86] Van Caneghem, M., Warren, D. H. D., *Logic Programming and Its Applications*, Ablex, Norwood, New Jersey, 1986, Pages 1 to 318.

[VEG85] Vegdahl, S. R., "The Design of an Interactive Compiler for Optimizing Microprograms", *18th Annual Microprogramming Workshop*, ACM, 3-Dec-85, Pages 129 to 135.

[VEG82A] Vegdahl, S. R., "Phase Coupling and Constant Generation in an Optimizing Microcode Compiler", *Proceedings of the 15th Annual Microprogramming Workshop*, ACM, 5-Oct-82, Pages 125 to 133.

[VEG82B] Vegdahl, S. R., "Local Code Generation and Compaction in Optimizing Microcode Compilers", Ph.D. Dissertation, Carnegie-Mellon University, 1-Dec-82, Pages 1 to 181.

[VERI84] Verity, J. W., "PROLOG vs. LISP", *Datamation*, 1-Jan-84, Pages 50 to 51.

[WAGD83] Wagner, A., and Dasgupta, S., "Axiomatic Proof Rules for a Machine-Specific Microprogramming Language", *16th Annual Microprogramming Workshop*, ACM, 11-Oct-83, Pages 151 to 158.

[WAI76] Waite, W. M., "Code Generation", in: *Compiler Construction, an Advanced Course*, 2nd Edition, F. L. Bauer and J. Eickel, (eds.), 1976, Pages 302 to 332.

[WAL77] Waldinger, R., "Achieving Several Goals Simultaneously", in: *Machine Intelligence 8*, John Wiley and Sons, New York, New York, 1977, also SRI Technical Note #107, 1-Jul-75, Pages 1 to 78.

[WALL77] Waldinger, R., and Levitt, K., "Reasoning about Programs", in: *Studies in Automatic Programming Logic*, by Z. Manna and R. Waldinger, 1977.

[WALT85] Walker, R. A., and Thomas, D. E., "A Model of Design Representation and Synthesis", *22nd Design Automation Conference Proceedings*, 23-Jun-85, Pages 453 to 459.

[WAL79A] Wallace, J. J., "The GLIDE/SLIDE Compiler Release 1", Carnegie-Mellon University Technical Report, 1-Jul-79, Pages 1 to 26.

[WAL79B] Wallace, J. J., "On the Automatic Verification of SLIDE Descriptions", Carnegie-Mellon University Technical Report, 1-Aug-79, Pages 1 to 71.

[WALP79] Wallace, J. J., and Parker, A. C., "SLIDE: An I/O Hardware Descriptive Language", *4th International Symposium on Computer Hardware Description Languages*, 8-Oct-79, Pages 82 to 88.

[WAR77] Warren, D. H., "Implementing Prolog - Compiling Predicate Logic Programs", Ph. D. Dissertation, Technical Report, University of Edinburgh, No. 39 & 40, 1977, also reprinted as: "Applied Logic - Its Use and Implementation as a Programming Tool", Technical Report SRI International, No. 290, 1983.

[WAR80] Warren, D. H., "Logic Programming and Compiler Writing", *Software - Practice and Experience*, Vol. 10, 1-Feb-80, Pages 97 to 125.

[WATH78] Waterman, D. A., Hayes-Roth, F., (eds.), *Pattern Directed Inference Systems*, Academic Press, New York, New York, 1978, Pages 1 to 658.

[WATH78B] Waterman, D. A., and Hayes-Roth, F., "An Overview of Pattern-Directed Inference Systems", in: *Pattern Directed Inference Systems*, D. A. Waterman and F. Hayes-Roth (eds.), Pages 3 to 24.

[WEB67] Weber, H., "A Microprogrammed Implementation of EULER on IBM System/360 Model 30", *Communications of the ACM*, Vol. 10, No. 9, 1-Sep-67, Pages 549 to 558.

[WEG76] Wegbreit, B., "Goal-Directed Program Transformation", *IEEE Transactions on Software Engineering*, Vol. 2, No. 2, 1-Jun-76, Pages 69 to 80.

[WEG79] Wegner, P., (ed.), *Research Directions in Software Technology*, 1979, MIT Press, Cambridge, Massachusetts, Pages 1 to 869.

[WEI80] Weidner, T. G., "CHAMIL A Case Study In Microprogramming Language Design", *SIGPLAN Notices*, Vol. 15, No. 1, 1-Jan-80, Pages 156 to166.

[WICK75] Wick, J., "Automatic Generation of Assemblers", Ph.D. Dissertation, Department of Computer Science, Yale University, 1975.

[WIL82] Wilkes, M. V., "The Processor Instruction Set", *15th Annual Microprogramming Workshop*, ACM, 5-Oct-82, Pages 3 to 8.

[WIL51] Wilkes, M. V., "The Best Way to Design an Automatic Calculating Machine", Report of the Manchester University Computer Inaugural Conference, Electrical Engineering Department of the Manchester University, Manchester, England, 1-Jul-51, Also reprinted in Swartzlander.

[WILS53] Wilkes, M. V., Stringer, J. B., "Microprogramming and the Design of the Control Circuits in an Electronic Digital Computer", *Proceedings of Cambridge Philosophy Society*, Part 2, Vol. 49, 1-Apr-53, Pages 230 to 238, Also reprinted in Siewiorek, Bell, and Newell.

[WILL79] Williams, M. H., "Long/Short Address Optimization in Assemblers", *Software - Practice and Experience*, Vol. 9, 1979, Pages 227 to 235.

[WINP84] Winston, P. H., and Prendergast, K. A., *The AI Business*, MIT Press, Cambridge, Massachusetts, 1984, Pages 1 to 324.

[WOSOLB84] Wos, L., Overbeek, R., Lusk, E., and Boyle, J., *Automated Reasoning Introduction and Applications*, Prentice-Hall, Englewood Cliffs, New Jersey, 1984, Pages 1 to 482.

[WOO79] Wood, G., "Global Optimization of Microprograms Through Modular Control Constructs", *12th Annual Microprogramming Workshop*, ACM, 1979, Pages 1 to 6.

[WOO78] Wood, G., "On the Packing of Micro-operations into Microinstruction Words", *11th Annual Microprogramming Workshop*, ACM, 1978, Pages 41 to 50.

[WULF82] *Electronics*, 30-Nov-82, Page 58.

[WULF80] Wulf, W. A., "PQCC: A Machine-Relative Compiler Technology", Carnegie-Mellon University Technical Report, 25-Sep-80, Pages 1 to 22.

[WULFWGH75] Wulf, W., Johnson, R. K., Weinstock, C. B., Hobbs, S. O., Geschke, C. M., *The Design of an Optimizing Compiler*, Elsevier North-Holland, New York, New York, 1975, Pages 1 to 165.

[ZIM80A] Zimmermann, G., "Computer Aided Design of Control Structures for Digital Computers", *IEEE International Conference on Circuits and Computers*, 1-Oct-80, Pages 103 to 106.

[ZIM80B] Zimmermann, G., "MDS - The MIMOLA Design Method", *Journal of Digital Systems*, Vol. 4, No. 3, 1-Sep-80, Pages 337 to 369.

[ZIM79A] Zimmermann, G., "The MIMOLA Design System A Computer Aided Digital Processor Design Method", *16th Design Automation Conference Proceedings*, 25-Jun-79, Pages 53 to 58.

[ZIM79B] Zimmermann, G., "Cost Performance Analysis and Optimization of Highly Parallel Computer Structures: First Results of a Structured Top-Down Design Method", *4th International Symposium on Computer Hardware Description Languages*, 8-Oct-79, Pages 33 to 39.