Retargetable Microcode Synthesis

ROBERT A. MUELLER
Colorado State University
and
JOSEPH VARGHESE
Microelectronics and Computer Technology Corporation

Most work on automating the translation of high-level microprogramming languages into microcode has dealt with lexical and syntactic analysis and the use of manually produced macro tables for code generation. We describe an approach to and some results on the formalization and automation of the more difficult problem of retargeting local code generation in a machine-independent, optimizing microcode synthesis system. Whereas this problem is similar in many ways to that of retargeting local code generation in high-level language compilers, there are some major differences that call for new approaches.

The primary issues addressed in this paper are the representation of target microprogrammable machines, the intermediate representation of local microprogram function, and general algorithmic methods for deriving local microcode from target machine and microcode function specifications. Of particular interest are the use of formal semantics and data flow principles in achieving both a general and reasonably efficient solution. Examples of the modeling of a representative horizontal machine (the PUMA) and the generation of microcode for the PUMA machine model from our working implementation are presented.

Categories and Subject Descriptors: B.1.4 [**Control Structures and Microprogramming**]: Microprogram Design Aids—*languages and compilers, machine-independent microcode generation*; D.3.4 [**Programming Languages**]: Processors—*code generation, compilers, translator writing systems and compiler generators*

General Terms: Design, Languages

Additional Key Words and Phrases: Data antidependency, data dependency, flow graph, machine description, microcode compaction, microcode generation, microinstruction set processors, microprogramming.

1. INTRODUCTION

The evolution of microprogramming since Wilkes has seen a growth in the sophistication of development tools [44]. As with general-purpose programming, microprogramming can often be simplified and its reliability improved through the use of symbolic languages and automatic translators. For machine-

This work was supported in part by National Science Foundation grants MCS-8107481 and DCR-8503941 and by the Evans and Sutherland Corporation.
Authors' addresses: R. A. Mueller, Department of Computer Science, Colorado State University, Fort Collins, CO 80523; J. Varghese, Microelectronics and Computer Technology Corporation, Austin, TX 78759.

independent languages we require translation systems that, ideally, can be ported with modest effort and high reliability.

As pointed out in [9], much progress has been made in automating the generation of parsers, whereas comparatively little progress has been made toward solving the more difficult problem of automating code generation. By *code (microcode) generation*, we mean the translation of an intermediate language representation produced by a parser and enhanced by flow analysis and optimization to a machine code (microcode) representation.

One approach to automating code (microcode) generation is the use of table-driven methods, in which machine-dependent templates map intermediate language constructs to machine code [21, 39] or microcode [5, 24, 40]. To extend the degree of automation, the generation of the machine-dependent code generation tables from machine descriptions was proposed by [8], [16], and [18]. The automated generation of microcode generators from machine descriptions was subsequently proposed by [25], [26], and [43].

The common objective of such research is the ability to produce code generators directly and reliably from machine descriptions, which are capable of exploiting the underlying architecture in a reasonably efficient fashion. The automated generation of microcode generators is even more difficult than the automated generation of code generators since the underlying architectures tend to be much more diverse and can offer considerable concurrency in the execution of micro-operations. Thus, the feasibility of such systems in production environments remains an open question.

Our general view of the problem, shown in Figure 1, is similar to that of Cattell [8]. However, both our representations and algorithmic methods differ owing to the different underlying environments. The translation process begins with some high-level language representation of a microprogram, which is syntactically analyzed and translated into an internal form for flow analysis procedures. The flow analysis procedures decompose the program into a flow graph with nodes corresponding to basic blocks. Finally, the program is symbolically executed to statically optimize the machine-independent representation and obtain nonprocedural functional specifications of basic blocks called *symbolic assertions*. All this can be efficiently done using well-known methods [3, 20, 38].

The remainder of the translation, being *machine dependent*, is considerably more difficult. Note first that any machine-independent system for performing code generation must utilize a specification of the target machine. Our view of such a system begins with a nonbinding assignment of target machine registers to variables in the symbolic assertion specifications, followed by the phase-coupled selection and optimization of target machine microcode from the machine-dependent form.

The microcode generation algorithm uses procedural machine-independent semantic knowledge to facilitate efficient exploitation of the target machine in generating microcode from the symbolic assertion specifications. Whereas most previously reported microcode generation systems employ functional operator-driven macroexpansion, our algorithm is *target machine architecture driven*. It is the data path structure of the target machine that determines how information is transferred and transformed, and thus our *flow-graph algorithm* functions by algebraically simulating the flow of data in synthesizing the microcode [31].

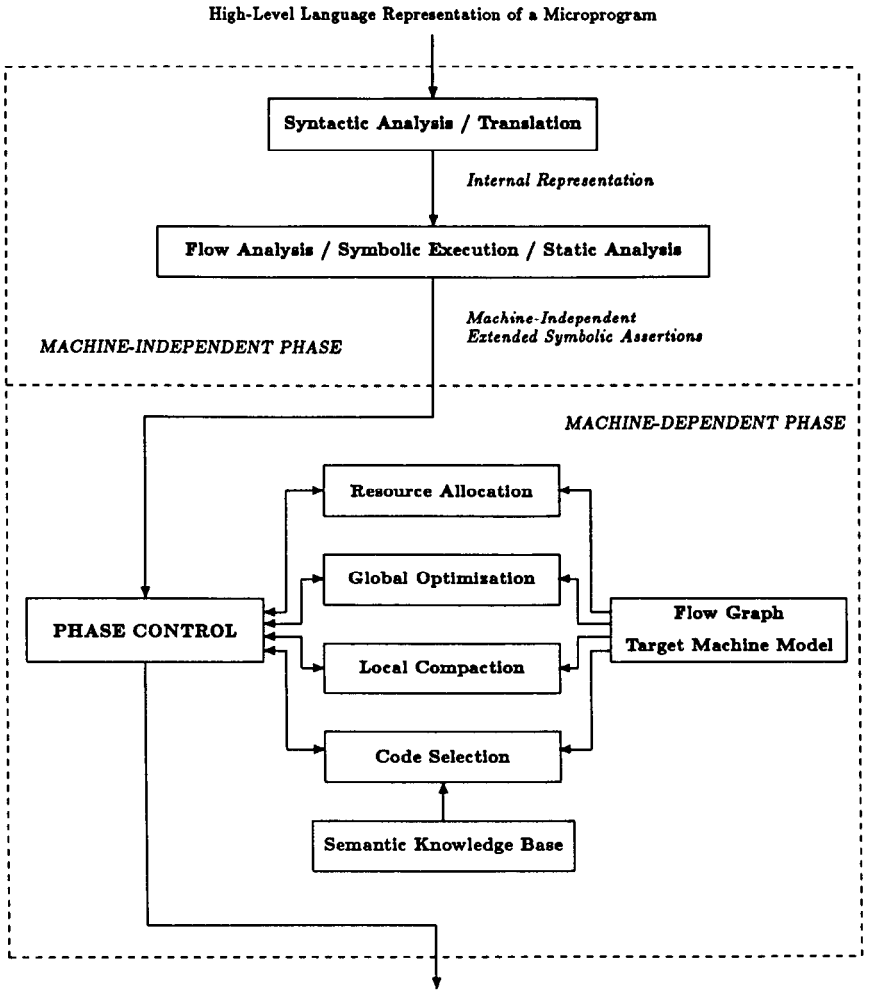


Fig. 1. CSU retargetable microcode generation system.

This paper focuses on flow-graph local microcode selection and grammar-based local compaction as the basis for the code generation component of a retargetable microcode compiler. Detailed discussions of the other components of the retargetable system shown in Figure 1 are given elsewhere.[1] The use of knowledge-based methods to achieve retargetability in the Colorado State University (CSU) system is the subject of [33].

This paper is organized into two major sections and a summary. The first section presents the flow-graph machine model and microgrammar compaction model of microarchitectures. A flow graph and microgrammar for the Processing

[1] There are a large number of technical system documents describing the CSU retargetable microcode synthesis system. Those interested in obtaining copies of the available documents should address their requests to R. A. Mueller.

Unit With Microprogrammed Arithmetic (PUMA) microarchitecture are given as examples. The second section presents retargetable local microcode selection and grammar-based compaction methods that extract target machine specifics from flow-graph and microgrammar models of the target microarchitecture. Several examples of algorithms targeted to the PUMA are provided. The paper is summarized with a discussion of problem areas and the current status of the Colorado State University retargetable microcode synthesis system.

First, we digress briefly in order to familiarize the reader with basic microprogramming terminology. Primitive operations of microprogrammable machines are known as *microoperations*. A collection of microoperations that are encoded into one word of the control store is termed a *microinstruction*, and a set of microinstructions that perform a well-defined task is a *microprogram*. For conciseness we use the abbreviations MO and MI for microoperation and microinstruction, respectively.

The execution time for an MI is termed a *basic clock cycle*. A basic clock cycle may consist of one or more *minor clock cycles*, and this relates to the *monophase/polyphase* characteristic of the machine. In a monophase implementation, all MOs within an MI execute in one minor cycle, whereas in a polyphase implementation, MOs could span multiple minor cycles within the basic clock cycle. Another basis for classification of microprogrammable machines measures the degree of concurrency within an MI. *Horizontal* and *vertical* machines form the end points of the spectrum, with horizontal machines having the most concurrency in MI execution. Machines that fall somewhere in between are sometimes termed *diagonal* machines.

An MO m_2 is *data dependent on* m_1 if m_1 executes before m_2 and m_1 writes to some resource read by m_2. Similarly, an MO m_1 is *data antidependent* on m_2 if the m_1 destroys data required by m_2 [6, 43].

2. MICROINSTRUCTION SET DESCRIPTION

A crucial step in automating the generation of code generators is the definition of a target machine description model.[2] The choice of the model determines both the scope of retargetability and the potential for efficiently and effectively generating the machine-dependent component of the code generator. Since these are competing goals, trade-offs must be made.

When the target machine model is of the von Neumann variety, each machine can be viewed as an instruction set processor that cyclically fetches instructions from a primary memory and then executes them to modify the processor state [8]. For such a domain, the machine description can be organized into five components: (i) a set of *storage bases* that represent the storage elements (state) of the machine (e.g., registers and primary memory), (ii) a set of *operand addressing modes* that describe how information in the storage bases can be accessed, (iii) *machine operation semantics* that describe the machine operations in terms of how they modify the machine state, (iv) *data types* of the operands of the machine operations, and (v) *instruction field format* information that describes how operations can be encoded into machine instructions.

[2] The term *target machine* denotes that machine which ultimately executes the microprograms being generated.

A microinstruction set processor is assumed, for our purposes, to cyclically fetch its instructions from a read-only *control store memory* and execute them to modify the machine state. It differs from an instruction set processor in at least two important ways: (i) a microinstruction set processor generally uses very simple addressing, and (ii) concurrency of microoperations execution is allowed, implying the need for the model to include timing and field encoding conflict information [43]. Some of the different microinstruction set description models proposed for use in automating microcode generation are given in [17], [27], [41], and [43].

The model we present is somewhat different from those previously proposed in that it is *data flow* oriented. That is, the machine is described, in part, in terms of how information flows through the storage base elements and operational units. The motivation for such an approach is to improve the efficiency of the microcode generation algorithm, which is based on the data-flow concept.

We can distinguish two major phases of the microcode synthesis process: the generation of a partial data dependency ordered collection of MOs that realizes the function associated with the intermediate language representation of a microprogram segment, and the compaction of the MOs into target machine MIs subject to the data dependency constraints in the partial order and the resource constraints manifested in the MI field encodings. Thus, our machine model has two major components: a *flow graph* to facilitate microcode generation and a *microgrammar* that reveals valid MI templates to facilitate compaction. More detailed discussions of the flow-graph model and microgrammar model can be found in [29] and [32], respectively, or in [27].

2.1 Flow-Graph Descriptions of Machines

The flow-graph component has two major subcomponents: machine resources (i.e., storage bases and operational units) and the data paths that allow information to flow between machine resources. Thus, our model is essentially a graph whose nodes represent machine resources and whose edges represent the potential for data flow between nodes. Since only the generation of local code is being considered, control sequencing information is not included.

2.1.1 *Machine Resources.* Each node in a flow graph represents a machine resource. Machine resources have both structural and behavioral characteristics. Structurally, a storage base element is a sequence of m words, each of length n. Similarly, a machine resource is behaviorally characterized in terms of the result it produces.

The behavioral characteristics of a machine resource correspond to the conditions under which the resource is permitted to be modified (*visibility*), the MOs that modify the state of the resource (*functionality*), and the lifetimes of states realized as a result of executing these MOs (*retention*). There are two degrees of visibility: image and target. *Image* variables are those resources accessible to the machine language programmer. The values of image variables cannot be modified by the synthesis system unless the microprogram specification calls for such a modification. *Target* variables may be used as temporaries. The distinction between image and target variables is particularly useful in microarchitecture emulation.

Functionality is given by the list of MOs that assign new values to (modify the state of) a resource. There are two components in each MO: the functional transformation effected by the MO and the MI encoding that enables it. The value assigned to the resource could be an expression that includes constants and the values of other resources adjacent to this resource in the flow graph. The encoding information aids in the detection of side effects.

It is sometimes the case that two independent or polyphase-ordered MOs are assigned the same MI encoding. In the code generation sense, we view one as a *side effect* of the other. Thus the selection of one of these MOs for execution necessitates the execution of all its side effects. For example, the execution of an arithmetic logic unit (ALU) operation causes the setting of the condition bits as side effects. A side effect may be necessary if it performs an action that is required by the specification of the computation for which code is being generated. An unnecessary side effect can be ignored if it affects only target variables. An unnecessary side effect, which affects image variables, is potentially harmful and could produce unexpected results if selected for execution.

The other important property of machine resources pertains to the lifetime of a value assigned to a resource. The *retention characteristic* is either *permanent*, when the resource retains its value until explicitly modified, or *transient*, when the value assigned to it is only stable for a fixed period of time. The timing information associated with each resource includes the earliest and latest time that the value of the resource is stable after an assignment to the resource is made. We assume that these times are a function of the resource and not of the MO that makes the assignment. This is a simplification, and the case in which these times vary can be modeled by making conservative estimates of the earliest and latest times.

Accordingly, each node in the flow-graph model has five attributes: (i) a unique identifier, (ii) capacity information in terms of bit length and the number of words, (iii) visibility, which is either image or target, (iv) a retention characteristic, which is either permanent or transient, and the associated timing information, and (v) functionality, which is a functional description of each of the MOs that can modify the state of the resource.

2.1.2 *Data Flow.* The flow of data between machine resources is inferred from the functional component of the nodes, in which the MOs that modify the state of the resource are described. Thus, a given resource is the implicit destination of each MO associated with it. Similarly, those resources that may pass data to the resource are elements of the expressions computed by the MOs.

As such, a sequence of data dependent MOs represents a path in the flow graph. Finding a sequence of MOs that transforms a given initial state to a desired final state can be achieved by finding a path through the flow graph. The destination resource orientation of the flow graph is conducive to the *backward chaining* method, used in the microcode derivation algorithm, in which we move from a desired final state toward an assumed initial state [45].

2.1.3 *The PUMA Microarchitecture.* To illustrate the representation, we use the PUMA machine [19] that was also selected for illustrative purposes by [14] and [43]. The PUMA microarchitecture was designed to emulate the CDC 6600,

and it reflects this bias in the word sizes of its registers. The CDC 6600 has eight 60-bit registers and sixteen 18-bit registers. Since the PUMA was constructed out of 4-bit slices, it has sixteen 60-bit registers and sixteen 20-bit registers. The data paths are 60 bits wide, and 20-bit data are right justified with zeros in the first 40 bits. The output of the registers is fed into an *unpack* unit that unpacks floating point data into a mantissa and an exponent. The output of the unpack unit feeds into the *buffer* and the exponent arithmetic section.

The arithmetic and logic unit (*alu*) has the buffer and the accumulator (*ac*) as inputs. The output of the alu is wire-ORed with data from the instruction unit and the exponent unit and, concatenated with the output of the *mq* register, it forms the input to the 120-bit *shifter* which is capable of 4, 16, and 60-bit right shifts. The output of the shifter feeds the ac and mq registers, each of which are 60 bits wide. Data from the ac register can be fed to the alu, to the *pack* unit (which assembles floating point data from the ac and from the exponent arithmetic unit), to the main memory unit, and to the instruction unit. Data from memory also passes through the ac register. This is only a brief description of the PUMA. For more details see [19]. A pictorial sketch of the PUMA data paths is given in Figure 2.

Besides the fact that it is a real, horizontal machine, the PUMA has other properties that recommend its use as a test target machine. The different word sizes in the register banks and the data paths are fairly common in real machines. The presence of side effects and transient machine resources also add to the complexity. There were also several compromises made when the PUMA was designed. For example, data can either be written into a register or read from a register during an MI, but not both, even if they are in different register banks. There are also some features that are easily modeled but not yet fully supported (e.g., floating point arithmetic).

2.2 Microgrammar Compaction Model

The local microcode compaction problem can be briefly described as follows: Given a set of MOs with data dependency, antidependency, and timing constraints, find a minimal-length microprogram that is functionally equivalent to this set of MOs. This problem has been shown to be NP-complete [11], and various heuristic solutions have been formulated that sacrifice optimality for near-optimal solutions.

Compaction algorithms can be machine independent if they are provided with machine-dependent information. Typically they need to know about MO conflicts and about the timing of MOs within MIs. MO conflicts can arise from contradictory assignments to MI fields or from resource usage conflicts. Conflicting MOs cannot be executed in the same MI.

The grammatical model assumes that MIs are composed of *series/parallel* sequences of MOs [42]. The metasymbol *next* denotes serial execution, whereas the metasymbol ; denotes parallel execution. MO conflicts can be inferred from the encoding information associated with each MO, and timing relationships can be derived from the series/parallel metasymbols and the position that an MO occupies in a series/parallel sequence [29]. The Backus–Naur form (BNF) notaton for series/parallel grammars is given in Figure 3.

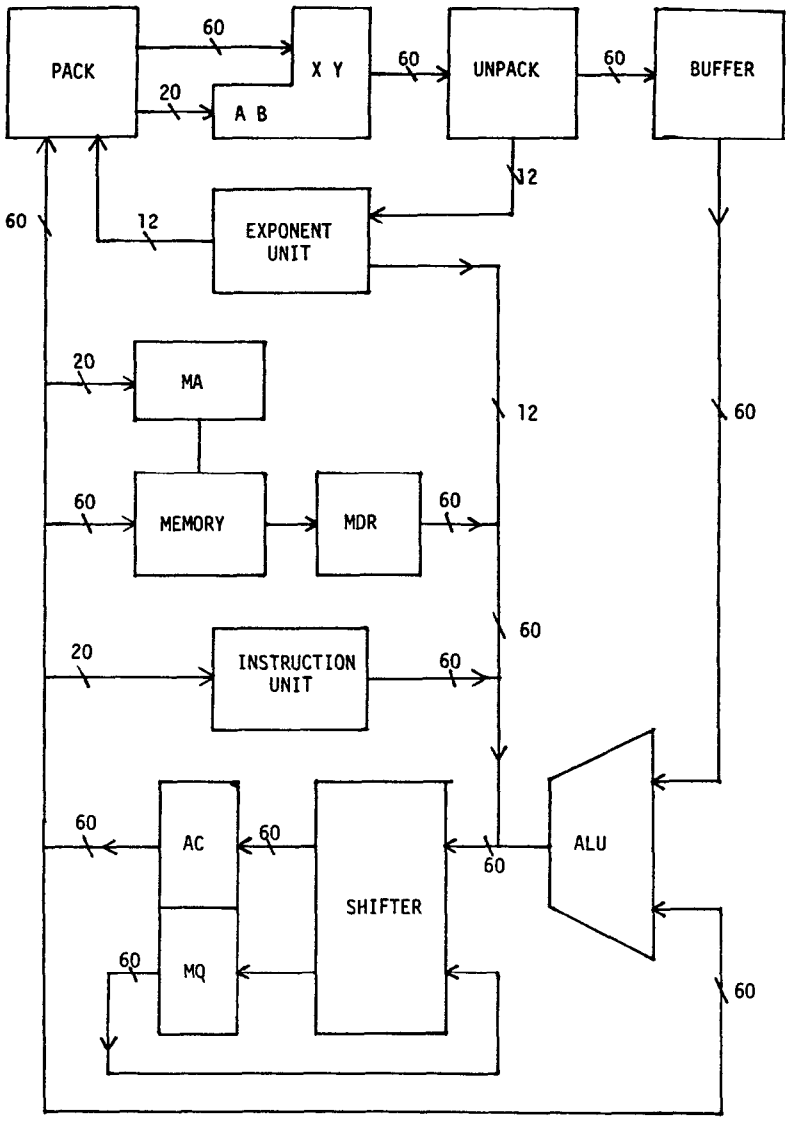


Fig. 2. The PUMA microarchitecture.

Example 2.1 Consider the following set of productions:

$$
\begin{aligned}
\langle \text{MI\_set} \rangle ::= {} & \langle A_1 \rangle \ \textit{next} \ \langle B \rangle \ \textit{next} \ \langle C_1 \rangle \mid \\
& \langle A_2 \rangle \ \textit{next} \ \langle B \rangle \ \textit{next} \ \langle C_2 \rangle \\
\langle A_1 \rangle ::= {} & a_1 \mid a_2 \mid a_3 \\
\langle A_2 \rangle ::= {} & a_4 \mid a_5 \mid a_6 \\
\langle B \rangle ::= {} & \langle D \rangle; \langle E \rangle \\
\langle C_1 \rangle ::= {} & c_1 \mid c_2 \mid c_3 \\
\langle C_2 \rangle ::= {} & c_4 \mid c_5 \mid c_6 \\
\langle D \rangle ::= {} & d_1 \mid d_2 \mid d_3 \\
\langle E \rangle ::= {} & e_1 \mid e_2
\end{aligned}
$$

⟨system_description⟩ ::= ⟨step⟩ | ⟨system_description⟩ **next** ⟨step⟩
⟨step⟩ ::= ⟨action⟩ | ⟨step⟩; ⟨action⟩
⟨action⟩ ::= ⟨elementary_action⟩ | (⟨system_description⟩)

Fig. 3. Series/parallel grammar.

The a_i, b_i, c_i, d_i, and e_i are MOs of a hypothetical micromachine. The terminal strings derived by the ⟨MI_set⟩ are series/parallel sequences of MOs that represent potential MIs. Each MO has an encoding component associated with it, as mentioned in Section 2.1.1. If MOs a_1 and b_1 have encodings $f = x_1$ and $f = x_2$, respectively, for some field f where $x_1 \neq x_2$, then a_1 and b_1 conflict in the field encoding sense. If MOs d_1 and e_1 have functional transformation components $r \leftarrow \exp_1$ and $r \leftarrow \exp_2$, respectively, for some machine resource r where $\exp_1 \neq \exp_2$, then they cannot be executed in the same minor cycle of an MI. Here d_1 and e_1 conflict in the machine resource usage sense.

Field encoding conflicts and machine resource usage conflicts can be determined by examining the encoding and functional transformation components, respectively, of the corresponding MOs. Timing conflict determination requires examination of the microgrammar. If a microprogram requires MO a_1 to be executed after MO c_1, then they conflict in the timing sense because, in an MI, the a_i MOs must always execute before the c_i MOs. MOs a_1 and c_4 cannot execute in the same MI even if they do not conflict in any of the senses described above. This is because there is no MO sequence that can be generated by the ⟨MI_set⟩ that contains both a_1 and c_4.

The flow-graph model and the microgrammar model can, in conjunction, handle most of the features of real machines. However, some of the basic assumptions used in these methods prevent the modeling of some aspects of microprogrammed machines. The requirement that the timing of MOs be representable in a series/parallel framework precludes the incorporation of asynchronous MOs (found, for example, in memory-accessing operations). Conditional MOs cannot be easily modeled using the flow-graph technique, especially when the operation is conditional on the value of a machine resource. Writable control stores permit the modification of microprograms, and this is a very difficult problem for program analysis tools in general.

3. RETARGETABLE LOCAL MICROCODE SELECTION

Microcode selection is the process of mapping intermediate representations of what function a code segment is to perform into a collection of target machine MOs that implement that function. We view microcode selection as the part of microcode generation in which machine-specific microcode sequences are substituted for machine-independent representations of microcode function. We assume that the intermediate representations correspond to any part (or all) of a basic block and that any storage resources referenced in the intermediate representation designate specific machine resources.[3]

[3] Thus, some register allocation is done prior to code selection. Additional "on-the-fly" register allocation is done during code selection when temporary registers are needed. The actual method used in handling register assignment is to assign *symbolic references* rather than specific register bank indices to delay the final register assignment until local compaction has been attempted. See [4] for further details.

The target machine MOs produced as output from the microcode selector have been *parallelized* and organized as a data dependency graph with *min/max delay* constraints labeling edges in the graph. A min (max) time on an edge from MO m_1 to MO m_2 specifies that a minimum (maximum) number of cycles can elapse between the execution of m_1 and m_2. Global optimization, local compaction, and final register assignment are subsequently performed on the generated data dependency graphs before microcode generation is completed. The time-constrained data dependency graphs and their manipulation are described in [34].

The approach we take to microcode selection is based on the observation that *every data-dependent sequence of target machine MOs corresponds to a path in the target machine flow graph.* As such, we can generate microcode by attempting to discover a set of flow-graph paths that collectively transfer and transform the source data of the basic block to the designated destination storage bases of the block. An important property of such an approach is that it cleanly separates the machine-independent microcode selection algorithm from the target machine.

In this section we discuss the intermediate representation of local microcode function and the retargetable local microcode selection algorithm that translates intermediate representations into target machine microcode.

3.1 Intermediate Representation of Local Microcode

There have been several different intermediate representations for basic blocks discussed in the literature. Cattell uses a tree representation (TCOL) [8], whereas Sheraga and Gieser use sequences of triples and seven-tuples [40]. Our representation is, basically, the set of *exit expressions* computed in the block.

In program flow analysis, there are two important classes of variables: those that are read before being defined in a basic block (*use* variables), and those variables that are live upon exiting the block as a consequence of being defined in the block (*def* variables). There are efficient algorithms for computing both the use and def variables [20]. The function of a basic block can be represented by associating with each def variable the value of the expression assigned to it during any execution of the basic block. Such exit expressions are independent of any actual block execution when expressed symbolically in terms of the initial values of the use variables and are derived using symbolic execution [38].

The function of a basic block can then be represented as a pair (I, F), where I associates with each use variable a symbolic constant that represents the value of that variable upon entry to the block, and F associates with each def variable a symbolic representation of the exit expression assigned to that variable in terms of the symbolic constants associated with the use variables. We call any mapping between a program variable and a symbolic expression a *symbolic state mapping* (SSM), any tuple of SSMs for distinct program variables a *symbolic state vector* (SSV), I an *initial symbolic state vector* (ISSV), F a *final symbolic state vector* (FSSV), and $sa(I, F)$ *a symbolic assertion* that specifies the function of a basic block.

For example, suppose the execution of basic block B assigns X the sum of the initial value of X and 1 and assigns Y the product of the initial values of W and X. Then X and W are the use variables of B, X and Y are the def variables of B,

and a symbolic assertion representation of B is

$$\langle W = a_1,\ X = a_2 \rangle \rightarrow \langle X = a_2 + 1,\ Y = a_1 * a_2 \rangle$$

3.2 Microcode Selection Algorithm

In general, there may be many def variables in the goal SSV. Further, each of the expressions associated with those variables may be computed independently. This will not generally yield optimal code, but this is to be expected since the optimization problem is NP-hard [7].[4] However, the significant advantage of such an approach lies in the potentially substantial efficiency gains in the execution of the algorithm.

The flow-graph method recursively constructs n microprograms independently for n exit expressions in a final SSV. There are two major processes in the procedure: the path determination process that selects a target machine path that moves information (backward) so as to derive microcode for an exit expression, and the data-dependency graph coupling process that unifies the separate but related data dependency graphs by adding data antidependency arcs. The first process is called *path arbitration*, and the second process is called *data dependency graph coupling.*

3.2.1 *Path Arbitration.* Let Q be an initial SSV and P a final SSV; $P = \langle s_1, \ldots, s_n \rangle$. We attempt to resolve the specification by independently deriving microcode for each of $Q \rightarrow \langle s_1 \rangle, \ldots, Q \rightarrow \langle s_n \rangle$, and coupling the resulting n data dependency graphs in a way that exposes potential parallelism and preserves the functionality of each graph.

Given $Q \rightarrow \langle X = \gamma \rangle$, we attempt to move the information associated with γ backwards from X to either a storage base or operational unit, depending on γ. Since γ is a symbolic expression, there are only three possible forms it can take: a *hard constant*, a *symbolic constant*, or a *functional expression.*

Hard constants can be generated using only a few different methods. Symbolic constants must be initial values of use variables in the initial SSV. The task is now to find a (reverse) path from X to use variables such that the MOs representing the paths perform the computation necessary to derive $Q \rightarrow \langle X = \gamma \rangle$.

If γ consists only of a single constant, then the synthesis system seeks a path to the origin (source) of this constant such that the data are not transformed. For example, if the symbolic assertion is

$$\langle X = \gamma \rangle \rightarrow \langle Y = \gamma \rangle$$

then X is the source of γ, and we need a sequence of MOs that transfer data from X to Y.

If the expression is more complicated, we consider its leftmost prefix operator. If the target machine contains an operational unit corresponding to that operator,

[4] We note, however, that the preprocessing of SSVs based on recognized common subexpressions will generally improve the efficiency of the generated microcode considerably. This preprocessing is currently implemented in the CSU retargetable microcode compiler.

the system tries to find a path to one such unit.[5] If not, it attempts to algebraically rewrite the expression in terms of those operators available on the machine.

For example, if the symbolic assertion is of the form

$$\text{ISSV} \rightarrow \langle X = f(\gamma_1, \ldots, \gamma_n) \rangle$$

then the algorithm searches for a target machine operational unit that performs the operation f on n operands. If no such operational unit exists, then the algorithm tries to transform the expression $f(\gamma_1, \ldots, \gamma_n)$ into some semantically-equivalent expression $g(\alpha_1, \ldots, \alpha_m)$ such that the target machine contains an operational unit for g. Note that when the prefix operator does have a corresponding operational unit, the choice of which operational unit to select as the source of the path must be made. The system picks the one that generates the shortest overall path, which is equivalent to picking the MO sequence having the smallest number of MOs.

The algorithm attempts the transformation one MO at a time, starting with the goal SSV and working backwards. If the MO we choose is of the form $A \leftarrow B$ and the goal specification is ISSV $\rightarrow \langle A = \gamma_1 \rangle$, then the new goal becomes ISSV $\rightarrow \langle B = \gamma_1 \rangle$. If the goal is of the form ISSV $\rightarrow \langle A = f(\gamma_1, \gamma_2) \rangle$, and the MO we choose is $A \leftarrow f(B_1, B_2)$, then the new goal is ISSV $\rightarrow \langle B_1 = \gamma_1, B_2 = \gamma_2 \rangle$. The determination of the new goals has a formal justification on the basis of the *weakest precondition* operator [12]. Since formal treatment of this is not consistent with the central theme of this paper, we refer the interested reader to [30] or [27].

Algebraic properties can often lead to a variety of choices in the determination of a new goal specification. For example, if f is a commutative operator, then we have two choices for the new goal, $\langle B_1 = \gamma_1, B_2 = \gamma_2 \rangle$ or $\langle B_1 = \gamma_2, B_2 = \gamma_1 \rangle$. Such choices can be represented with an AND-OR search tree [37, 43, 45]. When all expressions have been reduced to simple values of use variables, the process terminates.

We now demonstrate the method with several examples based on the PUMA machine model, in which the image variables are the 8 X, the 8 A, and the 8 B registers, and the rest of the variables, including the 8 Y registers, are target variables. The X and Y registers are 60 bits wide, and the A and B registers are 20 bits wide. All generated code shown was derived by a prototype implementation developed by the authors. Note that the examples have been kept simple for the sake of clarity, and much of the detail of the operation of the implementation has been left out to improve readability.

Example 3.1 Consider the symbolic assertion

$$\langle X(0) = \alpha \rangle \rightarrow \langle A(0) = \alpha[19, \ldots, 0] \rangle$$

where $Z(n)$ denotes the nth word in the Z array (which may be a register bank) and $Z[a, \ldots, b]$ refers to bits a to b, inclusive, of Z, where Z may be a symbolic expression or a variable. The expression $\alpha[19, \ldots, 0]$ is a fragment of a symbolic constant whose source is $X(0)$. Looking backwards from $A(0)$, the shortest path to $X(0)$ is through the pack unit, the ac register, the shifter, the alu, the buffer,

[5] Note that our implementation precomputes all paths exactly once for each target machine.

```
unpack ← X(0)
buffer ← unpack
alu ← buffer
shifter[119, ..., 60] ← alu
ac ← shifter[119, ..., 0]
pack ← ac
A(0) ← pack[19, ..., 0]
```

Fig. 4. MO sequence derived in Example 3.1.

and the unpack unit in that order. (Shortest paths between nodes can be precomputed using any of the methods applicable to graphs [2].) Using the weakest precondition semantics and moving $\alpha[19, \ldots, 0]$ back to the pack unit, we get the new goal $\langle \text{pack}[19, \ldots, 0] = \alpha[19, \ldots, 0] \rangle$, using the MO $A(0) \leftarrow \text{pack}[19, \ldots, 0]$.

Since the path from $X(0)$ to the pack unit is 60 bits wide, the most likely candidate for the contents of pack[59, ..., 20] is $\alpha[59, \ldots, 20]$. Hence this goal is equivalent to $\langle \text{pack} = \alpha \rangle$. Moving α backwards to the ac register produces the goal $\langle \text{ac} = \alpha \rangle$. Continuing in this manner, we get the MO sequence shown in Figure 4 in the reverse order of selection.

In performing the MO shifter[119, ..., 60] ← alu, we have as a side effect the MO shifter[59, ..., 0] ← mq. However, as mentioned earlier, the shifter is a target variable, and side effects to target variables can be ignored unless they are required by the microprogram specification. Henceforth, we will simply ignore such side effects in the derivations presented.

The MOs presented above were linearly ordered. In general, the MO set derived is usually in the form of a partial order with directed edges between MOs in its graphical representation. A *directed edge* (m_1, m_2) denotes that MO m_1 must execute before MO m_2, because m_2 is data dependent or data antidependent on m_1.

The directed edges also have two attributes that represent the timing relationships between MOs. The first attribute of the directed edge is the minimum number of clock cycles that must elapse between the executions of the dependency-related MOs. The second attribute is the maximum such number; *inf* indicates that the second MO can execute any time after the first, subject to the minimum time constraint. Transient variables lose their values after a certain fixed number of cycles. Thus the maximum number of cycles that can elapse between a definition of such a variable and its use is finite. The data dependency graph representation of the MO sequence generated in Example 3.1 is given in Figure 5.

Example 3.2. As another example of a data transfer, we look at a memory read operation. The assertion for reading from memory into register $X(1)$ is

$$\langle mem(\alpha) = \gamma, ma = \alpha \rangle \rightarrow \langle X(1) = \gamma \rangle$$

where *ma* is the memory address register. There is no way to transfer data from memory to the ac register directly. However, the data from the memory data register (*mdr*) are wire-ORed with the output of the ALU. This requires transforming $\langle X(1) = \gamma \rangle$ to $\langle X(1) = \text{or}(0, \gamma) \rangle$, using one of a set of equivalence preserving axioms discussed at greater length later in this section. Passing this

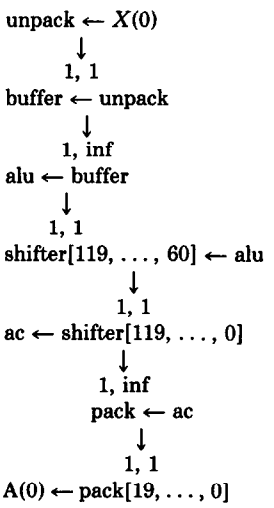


Fig. 5. Data dependency graph derived in Example 3.1.

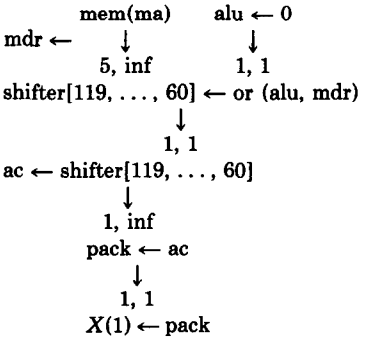


Fig. 6. Data dependency graph derived in Example 3.2.

goal backward through the ac and the shifter yields ⟨shifter[119, . . . , 60] = or(0, γ)⟩. Using the MO shifter[119, . . . , 60] ← or(alu, mdr) we get ⟨alu = 0, mdr = γ⟩. The complete data dependency graph for this example is given in Figure 6.

Example 3.3 As a final example consider the case in which we require some data divided by 64, which is equivalent to shifting the data right by 6 bits (assuming one's complement arithmetic). The symbolic assertion for this computation is

$$\langle X(0) = \alpha \rangle \rightarrow \langle X(0) = sxt(\alpha[59, \ldots, 6]) \rangle$$

where $sxt(Z)$ refers to sign extension. The shifter can only shift data by 4, 16, or 60 bits. The ac and the mq can function together as a 120-bit shifter by shifting data by one bit to the left or right. The synthesis system can classify bitwise operations into different categories, one of which is arithmetic shifts. Any arithmetic shift that the machine can perform in one or more operations can be broken down into functional compositions of arithmetic shifts. Hence, the machine has only to look at the shifter and the ac-mq shifts. Thus a 6-bit shift can

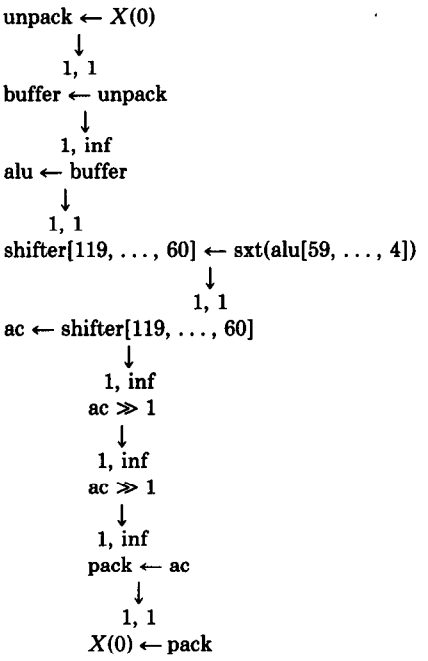


Fig. 7. Data dependency graph produced in Example 3.3.

be accomplished by shifting to the right by 4 bits in the shifter and then twice in the ac and mq registers. The MO sequence for this example is given in Figure 7. Here $X \gg n$ refers to a shift right of X by n bits.

As is obvious from the above example, the synthesis system has some understanding of bitwise operations. This has been one of the major weaknesses of other microprogram development systems [10, 43]. To realize this capability, the system must be able to convert among different ways of representing data. It does not have a complete set of axioms but instead uses a set of heuristics to guide the derivation. This kind of reasoning is also useful in constant generation in cases in which the literal field of the MI word is a bottleneck [43].

The solution search time can be drastically improved by storing computed data dependency graphs in a table for subsequent direct look-up. This makes the derivation of efficient code through the use of coupling methods between different phases such as resource allocation, microcode selection, and microcode compaction more computationally tractable. The mechanism used in the CSU retargetable microcode synthesis system for machine table construction and direct code sequence look-up is described in [13] and [28].

3.2.2 *Data Dependency Graph Coupling.* Given a collection of input data dependency graphs that compute the exit expressions that comprise a basic block, we need to compute a single data dependency graph for the block. The single data dependency graph includes all the MOs and data dependencies between the MOs from the input data dependency graphs. However, additional data antidependencies must be introduced to synchronize usage of common resources in different input data dependency graphs.

Each resource used in an input data dependency graph has a *live track* associated with it. A live track is a subgraph rooted by a *born node* and having *die nodes* as leaves. The born node is an MO in which the resource assumes a value. A die node is an MO that reads that value and is not followed by another MO that reads the same value. (For example, the sequential assignments $X \leftarrow A$; $Y \leftarrow X$; $Z \leftarrow X$ might define a live track for X. X is born in the MO $X \leftarrow A$ and dies at the MO $Z \leftarrow X$. X does not die at $Y \leftarrow X$, since the same value of X is read by the following MO $Z \leftarrow X$.)

The data-dependency graph coupling problem is a problem of introducing data antidependencies between nodes of the input data dependency graphs so that no live tracks for a particular resource overlap. In general, solutions to the problem are not unique. Further, the solution found generally has a significant influence on the effects of subsequent compaction and register assignment. The specifics of the coupling algorithm form the basis for a paper in their own right. (We are currently writing such a paper and refer the interested reader to the interim report [34].) To illustrate the nonuniqueness, Figure 8 offers distinct solutions for the coupling of the data dependency graphs derived in Examples 3.1 and 3.2.

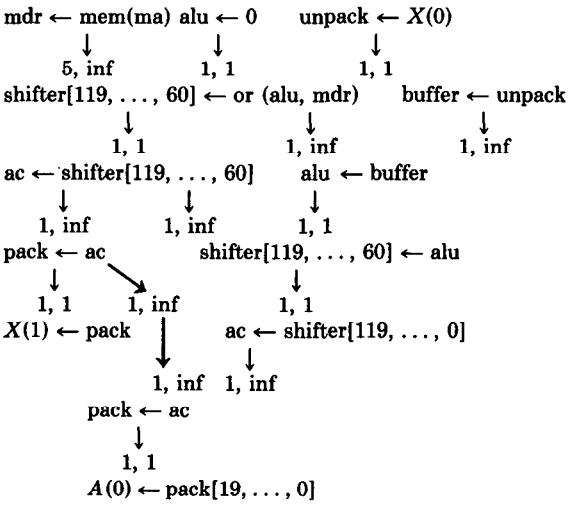
3.3 Microcode Compaction Algorithm

Given a data dependency graph representing local target microcode, the MOs represented by the nodes must be assigned to MIs subject to the dependency and timing constraints in the graph and the encoding constraints on MIs of the target machine. The problem is referred to as *microcode compaction.* Generally, one seeks an assignment of MOs to MIs that minimizes expected execution time. However, heuristics are required, since the general form of the problem is NP-hard [11]. Many such heuristics have been reported in the literature, with a good summary given in [22].

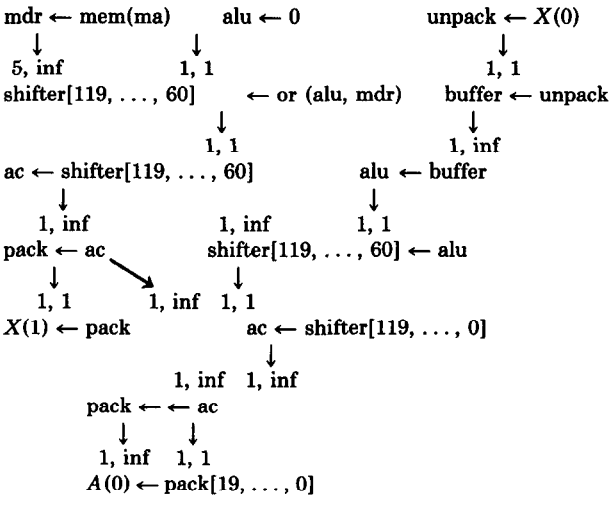
A major subproblem of microcode compaction is the problem of determining whether an MO may reside in a particular MI. An MO is *data ready* with respect to placement in a particular MI if its placement in that MI does not violate any data dependencies or data antidependencies. An MO is *timing ready* with respect to placement in a particular MI if it is data ready with respect to that MI and its placement in the MI does not violate any timing constraints. A timing-ready MO with respect to a given MI can only be placed in that MI if it does not conflict with the encoding of any other MO assigned to that MI.

The microgrammar compaction model introduced in Section 2.2 is used to guide the compaction of timing-ready MOs into MIs. It is easily used with any of the compaction heuristics for selecting timing ready MOs [15, 22] to yield a machine-independent compactor. The grammatical packing of an MI is similar to a recursive-descent parse in which parse tree leaf nodes represent MOs. The MOs are placed in (reverse) polyphase order in the MI, as determined by the series/parallel meta-symbols embedded in the microgrammar for the target machine. Thus, with a given heuristic for ordering the placement of timing-ready MOs, the grammatical packer resolves the placement of MOs in MIs in a machine-independent fashion, using the microgrammar to resolve the machine-dependent MI timing and encoding structure. The result is a retargetable local compactor.

The result of packing the PUMA machine data dependency graph of Figure 8a is shown in Figure 9. The compaction algorithm utilizes a list-scheduling heuristic



(a)



(b)

Fig. 8. Results of coupling data dependency graphs from Examples 3.1 and 3.2.

that is greedy with MO height in the data dependency graph to select timing-ready MOs and the grammatical method of packing MIs.

4. CONCLUSIONS

We have presented a synthesis system for the local generation of microcode. This system can be incorporated into the back end of a retargetable microcode compiler, such as the Colorado State University retargetable microcode compiler or any of the table-driven compilers proposed by [5], [24], or [40]. Input to the system is in the form of symbolic assertions that express the functionality of

MI #	Minor Cycle #	Microoperation
1	1	mdr ← mem(ma); unpack ← $X(0)$
	2	buffer ← unpack
2	1	alu ← 0
	2	shifter[119, . . . , 60] ← or (alu, mdr)
	3	ac ← shifter[119, . . . , 60]
	4	pack ← ac
	5	$X(1)$ ← pack
3	1	alu ← buffer
	2	shifter[119, . . . , 60] ← alu
	3	ac ← shifter[119, . . . , 0]
	4	pack ← ac
	5	$A(0)$ ← pack[19, . . . , 0]

Fig. 9. Compacted MOs for Figure 8.

basic blocks without imposing any sequencing structure. The symbolic assertions can be efficiently constructed from conventional intermediate program representations produced by the front end of a high-level language compiler [27]. The microcode selector transforms an assertion into a data dependency graph using a data-dependency graph coupling procedure to synchronize resource usage. The resulting data dependency graph is compacted using any of a variety of list-scheduling heuristics and a grammatical MI packer.

The flow graph model has been shown to be sufficiently general to handle an actual, representative horizontal machine. This model is oriented toward code generation using data flow principles. Its main features include the capability to model different word widths in the same machine, complex timing relationships, and volatile machine resources. The microgrammar model is used during compaction to determine microoperation conflicts and to determine the relative positioning of microoperations within microinstructions. At present, these machine models cannot handle some features of real machines such as asynchronous timing, writable control stores, and subroutines.

The main feature of the local microcode synthesis system is its data flow orientation. Among its major innovations are machine-independent methods of coupling independently-derived data dependency graphs without imposing strict sequentiality, the ability to handle side effects, the ability to reason about machines with different word widths, and a limited understanding of bitwise operations such as rotates, shifts, and bit extractions.

Although the system did generate the microcode in the examples described above, it does have some limitations. It does not have a complete understanding of bitwise operations and in this area it must rely on heuristics. The application of equivalence axioms can lead to much larger search spaces and more intelligence is required in selecting the relevant axioms. The system described forms the basis for a retargetable microcode compiler for a C-like language that was developed at Colorado State University. The compiler has been targeted to several hypothetical machines, the PUMA machine, and an AMD29500-based signal processing architecture [1].

We are currently targeting the compiler to the highly horizontal microarchitecture of the next-generation Evans and Sutherland graphics processor

and developing retargetable methods of phase-coupled global optimization and delayed register assignment.

ACKNOWLEDGMENTS

The authors gratefully acknowledge the financial support provided by the National Science Foundation and the Evans and Sutherland Corporation and the use of the DEC VAX-11/780, provided by the Center for Computer-Assisted Engineering at Colorado State University, for our implementation and experiments. Thanks also to Steve and Vicki Allan for their critical comments on an earlier draft of this paper and to Mike Duda for helping with revisions.

REFERENCES

1. ADVANCED MICRO DEVICES COMPANY, AM29500 Application Note, Santa Clara, Calif. 1985.
2. AHO, A. V., HOPCROFT, J. E., AND ULLMAN, J. D. *The Design and Analysis of Computer Algorithms.* Addison-Wesley, Reading, Mass., 1974.
3. AHO, A. V., AND ULLMAN, J. D. *Principles of Compiler Design.* Addison-Wesley, Reading, Mass., 1977.
4. ALLAN, V. H., AND MUELLER, R. A. Retargetable code generator register assignment and heuristics (version 3.1). Firmware Engineering and Micro-Architecture Design Lab. Doc. MAD-86-18, Colorado State Univ., Fort Collins, Colo., 1986.
5. BABA, T., AND HAGIWARA, H. The MPG system: A machine-independent efficient microprogram generator. *IEEE Trans. Comput. C-30,* 6 (1981).
6. BANERJEE, U., SHEN, S., KUCK, D. J., AND TOWLE, R. A. Time and parallel processor bounds for Fortran-like loops. *IEEE Trans. Comput. C-28,* 9 (Sept. 1979).
7. BRUNO, J., AND SETHI, R. Code generation for a one-register machine. *J. ACM, 23,* 3 (July 1976), 502–510.
8. CATTELL, R. G. G. Formalization and automatic derivation of code generators. Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., April 1978.
9. CATTELL, R. G. G. "Automatic derivation of code generators from machine descriptions. *ACM Trans. Program. Lang. Syst. 2,* 2 (Apr. 1980), 173–190.
10. DASGUPTA, S. Principles of firmware verification. Submitted for publication.
11. DEWITT, D. J. A machine-independent approach to the production of optimal horizontal microcode. Ph.D. dissertation, Univ. of Michigan, Ann Arbor, Mich., 1976.
12. DIJKSTRA, E. W. Guarded commands, nondeterminacy, and the formal derivation of programs. *Commun. ACM, 18,* 8 1975, 453–457.
13. DUDA, M. R., AND MUELLER, R. A. Retargetable code generator micro-architecture specification and representation (version 3.1). Firmware Engineering and Micro-Architecture Design Lab. Doc. MAD-86-13, Colorado State Univ., Fort Collins, Colo., 1986.
14. FISHER, J. A. The optimization of horizontal microcode within and beyond basic blocks: An application of processor scheduling. Ph.D. dissertation, New York Univ., New York, N.Y. (Oct. 1979).
15. FISHER, J. A. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput. C-30,* 7 (July 1981).
16. FRASER, C. W. Automatic generation of code generators. Ph.D. dissertation, Computer Science Dept., Yale Univ., New Haven, Conn., 1977.
17. GIESER, J. L. On horizontally microprogrammed microarchitecture description techniques. *IEEE Trans. Softw. Eng. SE-8,* 5 (Sept. 1982).
18. GLANVILLE, R. S. A machine-independent algorithm for code generation and its use in retargetable compilers. Ph.D. dissertation, Electrical Engineering and Computer Science, Univ. of California, Berkeley (Dec. 1977).
19. GRISHMAN, R. The structure of the PUMA computer system. U.S. Department of Energy Report, Courant Mathematics and Computing Lab., New York Univ., New York, 1978.
20. HECHT, M. S. *Flow Analysis of Computer Programs.* North-Holland, New York, 1977.

21. JOHNSON, S. C. A portable compiler: Theory and practice. In *Proceedings of the Fifth Annual ACM Symposium on Principles of Programming Languages*, (Tucson, Ariz., Jan. 23–25). ACM, New York, 1978, 97–104.
22. LANDSKOV, D., DAVIDSON, S., SHRIVER, B. D., AND MALLETT, P. W. Local microcode compaction techniques. *Comput. Surv. 12*, 3 (Sept. 1980), 261–294.
23. LEVERETT, B. W. Register allocation in optimizing compilers. Ph.D. dissertation, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1981.
24. MA, P-Y. R., AND LEWIS, T. On the design of a microcode compiler for a machine-independent high-level language. *IEEE Trans. Softw. Eng. SE-7*, 3 1981.
25. MARWEDEL, P. A retargetable compiler for a high-level microprogramming language. *Micro-17* (New Orleans, La., Oct. 30–Nov. 1, 1984), ACM, New York, 1984.
26. MUELLER, R. A. Automated microprogram synthesis. Ph.D. dissertation, Univ. of Colorado, 1980.
27. MUELLER, R. A. *Automated Microcode Synthesis.* UMI Research Press, Ann Arbor, Mich., 1984.
28. MUELLER, R. A. Retargetable code generator table-driven code selector–Prolog Implementation (version 3.1). Firmware Engineering and Micro-Architecture Design Lab. Doc. MAD-86-7, Colorado State Univ., Fort Collins, Colo., 1986.
29. MUELLER, R. A., AND VARGHESE, J. Grammatical models of micro-instruction set processors. Tech. Rep. CS-81-10, Dept. of Computer Science, Colorado State Univ., Fort Collins, Colo., 1981.
30. MUELLER, R. A., AND VARGHESE, J. Formal semantics for the automated derivation of microcode. In *ACM IEEE 19th Design Automation Conference, Proceedings* (Las Vegas, Nev., June 14–16). IEEE, New York, 1982, 815–824.
31. MUELLER, R. A., AND VARGHESE, J. Applying algebraic simulation to machine-independent microcode synthesis. *Euromicro J. Microprocess. Microprogramm. 11*, 2 (Feb. 1983).
32. MUELLER, R. A., AND VARGHESE, J. Flow graph machine models in microcode synthesis. In *Proceedings of the 16th Annual Microprogramming Workshop* (Downington, Pa., Oct. 11–14) New York, 1983, 159–167.
33. MUELLER, R. A., AND VARGHESE, J. Knowledge-based code selection methods in retargetable microcode synthesis. *IEEE Design and Test, 2*, 3 (Aug. 1985).
34. MUELLER, R. A., AND SWEANY, P. Retargetable code generator series-parallel coupler/decoupler for DAGs (version 3.1). Firmware Engineering and Micro-Architecture Design Lab. Doc. MAD-86-10, Colorado State Univ., Fort Collins, Colo., 1986.
35. MUELLER, R. A., ALLAN, V. H., AND VARGHESE, J. The complexity of horizontal word encoding in microprogrammed machines. *IEEE Trans. Comput. C-33*, 10 (Oct. 1984).
36. MUELLER, R. A., ALLAN, V. H., AND VARGHESE, J. The complexity of horizontal word encoding in microprogrammed machines. *IEEE Trans. Comput. C-33*, 10 (Oct. 1984).
37. NILSSON, N. J. *Principles of Artificial Intelligence.* Tioga Press, Palo Alto, Calif. 1980.
38. REIF, J. H., AND TARJAN, R. E. Symbolic program analysis in almost-linear time. *SIAM J. Comput. 11*, 1 (Feb. 1981).
39. RIPKEN, K. Formale beschreibung von maschinen, implementierungen und optimierender maschinencodeerzung aus attributierten programmgraphe. Ph.D. dissertation, Technische Univ. München, Munich, 1977.
40. SHERAGA, R. J., AND GIESER, J. L. Automatic microcode generation for horizontally microprogrammed processors. In *Proceedings of the 14th Annual Workshop on Microprogramming* (Chatham, Mass., Oct. 12–15). ACM, New York, 1981, 154–168.
41. SINT, M. MIDL—A microinstruction description language, In *Proceedings of the 14th Annual Workshop on Microprogramming* (Chatham, Mass. Oct. 12–15). ACM, New York, 1981, 95–106.
42. SMITH, B. J., AND JORDAN, H. F. Implications of series-parallel sequencing rules. *Computing 19* (Jan. 1978), New York.
43. VEGDAHL, S. R. Local code generation and compaction in optimizing microcode compilers. Ph.D. dissertation, Carnegie-Mellon Univ., Pittsburgh, Pa., 1982.
44. WILKES, M. V. The best way to design an automatic calculating machine. Report of the Manchester University Computer Inaugural Conference, Electrical Engineering Dept., Manchester Univ., England, July 1951.
45. WINSTON, P. H. *Artificial Intelligence.* Addison-Wesley, Reading, Mass., 1984.

Received April 1984; revised June 1986; accepted August 1986