Design considerations for microprogramming languages*

by GREGORY R. LLOYD and ANDRIES VAN DAM

Brown University
Providence, Rhode Island

INTRODUCTION

Historically, microprograms have been developed using tools which are appropriate to logic designers (block diagrams, register transfer languages), or systems programmers (microcode assemblers). With the growth of user microprogramming, and the increased demands placed upon computer manufacturers for firmware support, improved tools and techniques have been suggested. In particular, microprogram compilers, i.e., compilers which translate high level source statements into sequences of microprogram control words, have been proposed and implemented. The larger issue to be faced is the nebulous task of supporting the needs of a community which includes:

- Manufacturers developing one particular computer (target machine) using microprogramming as a design and implementation technique.
- Manufacturers developing a computer system, integrating hardware, operating systems, and programming language support, utilizing microprogramming to tie system functions together.
- Users developing their own target machines tailored to applications such as computer graphics, signal processing, or interpretation of higher level languages.
- Users or manufacturers who utilize hardware which is microprogrammable to directly solve one or more specific problems, without formally defining a target machine (i.e., simply programming the hardware, or host machine).

All groups may be able to employ the same particular host machine (a microprogrammable computer), but their needs are shaped by different viewpoints. The remainder of this paper attempts to focus on the basic issue of problem solving methodologies, particularly, what can be done to make the task of the microprogrammer less difficult, regardless of the application.

Probably the major consideration in the design of a microprogramming language is a pragmatic issue—efficiency of generated code. The second section briefly describes some of the problems involved in compiler design. Language design (third section) is constrained by practical limitations on the complexity of the compilation process. Within these limits, determined by machine architecture, it may be possible to design a useful higher level microprogramming language. A specific example of a tailored microprogramming language is presented in the last section.

MICROPROGRAM COMPILERS

General issues

There are at least two distinct approaches to providing higher level support for a user microprogrammable computer. One is definition of a hospitable target machine over the host machine's basic hardware (and possibly a systems language which generates code for the target machine). It is possible to define high level primitives in the target machine's repertoire which closely match the facilities provided in a higher level language, including flow of control, expression evaluation (for complex mathematical operations), data manipulation, etc. A number of papers presented at the SIGPLAN/SIGMICRO Interface Meeting (May, 1973) treat the topic of instruction set design in some detail. A typical target machine could include general arithmetic, logical, and executive primitives in an efficient manner (particularly given some hardware support to speed instruction decoding).

The other approach is to make the microprogramming environment itself reasonably hospitable by providing higher level support for the direct generation of microcode. This also has received some attention in the current literature. Defining the phrase “high level language” is one immediate problem. Within this paper, a high level language is defined as one which has at least the following set of features:

(1) Symbolic user variables (allocated by the compiler),
(2) Ability to evaluate arbitrary arithmetic or logical expressions,
(3) Flow of control statements beyond simple (conditional and unconditional) GOTO, SKIP, Branch and Link.

It should be stated that the target machine approach and the microprogram compiler approach are not mutually exclusive: language and target machine design can proceed.

* This research was supported by U.S. Naval Research Laboratory Contract N00014-67-A-0191-0027.
simultaneously. For example, the Microdata 32/8 computer and the MPL language were designed in parallel, reflecting language constructs directly into target machine operations, and vice versa.12 The end result is a PL/I subset which is the target machine language in much the same sense that ALGOL is the machine language of the B5500. In another sense, a higher level language for the host machine can be considered an abstraction of the host hardware, shielding the programmer from details (such as loading a shift-amount register), but constraining him to those functions which are implemented at a hardware level.

Host machine microprogram compilers

The general advantages of programming in a higher level language apply when the compiler happens to generate microcode—namely, increased programmer productivity, self-documentation, improved maintainability, ease of training, etc. In addition, an application running in direct microcode could gain advantages in efficiency and security over a similar software based system. A language can be designed on a number of levels ranging from problem oriented languages through "general purpose" languages to machine dependent languages. Depending upon the design level of the language, the programmer can be fully isolated from the hardware or can be presented with a set of facilities which allow him to interact at a register level (a "CODE . . . ENDCODE" statement, which allows insertion of assembler or machine language instructions in-line13). The concept of language extensibility can be used to tailor a base language to a specific set of applications by, for example, enriching the run time environment with operating system primitives when appropriate.

The major disadvantage of microprogram compilers is the conjectural nature of statements on efficiency of compiler produced code. Microprogramming has traditionally been viewed as a fairly esoteric discipline, requiring complete knowledge of the host architecture and a willingness to spend considerable time tuning programs to (simultaneously) conserve control storage and achieve maximum performance. If the quality of compiler produced code is significantly worse than that produced by hand, the ergonomic advantages of coding in a higher level language may be outweighed. Hopefully, the efficiency requirement can be recognized as a major influence on language design, and feature selection can be guided by (1) the functions which can be performed efficiently in hardware and (2) recognition of constructs which can be compiled to efficient code. Optimization of code produced for some classes of microprogrammable machines is inherently difficult, and, except for work by Ramamoorthy and Tirrell, relatively unexplored in the literature.5,8,14

The following section treats some of the problems in compiler design which relate to the decision to generate microcode directly from a higher level language.

Problems in compilation to microcode

Microinstruction formats

Three possible microinstruction formats are illustrated in Figure 1. The first (labelled V) is typical of the vertically encoded format. One microinstruction (MI) consists of a control field F, and a data part D (which might specify register number(s), literal data, a branch address, etc.). The Nv bits which comprise the operator part COP) are fed through a decoding network, controlling up to 2Nv different lines, Ci. Each control line determines a micro-operation (MO) which might, for example, perform an arithmetic operation, transfer the contents of one internal register to another, initiate a memory read, etc. For format V, one micro-operation is specified in one microinstruction.

The format labelled H1 is an example of a horizontal direct control format. Each bit in the operator part controls one of NH1 lines. In general, not all of the OP bit patterns are legal, since some micro-operations cannot be executed in the same machine cycle (because of conflicting register or function utilization). The NH1 bits which comprise the operator part (OP) are fed through a decoding network, controlling up to 2NH1 different lines, ci. Each control line determines a micro-operation (MO) which might, for example, perform an arithmetic operation, transfer the contents of one internal register to another, initiate a memory read, etc. For format V, one micro-operation is specified in one microinstruction.

The format labelled H2 is an example of a horizontal direct control format. Each bit in the operator part controls one of NH2 lines. In general, not all of the OP bit patterns are legal, since some micro-operations cannot be executed in the same machine cycle (because of conflicting register or function utilization). The number of micro-operations which could be specified per microinstruction is NH2, the number of control lines.

The third format (H2) represents a horizontal field-encoded microinstruction. The OP is split into several fields, each of which is decoded to control one of a set of (related) control lines. Again, not all values of the OP may be legal. It is also possible for one field (say, F') to determine the layout and

Figure 1—Microinstruction formats

From the collection of the Computer History Museum (www.computerhistory.org)
The interpretation of the remaining fields. The number of microoperations specified per microinstruction is K, the number of encoded fields.

In practice, horizontal machines may employ a mixture of direct and encoded control fields, and "vertical" machines may have more than a single encoded field. For the sake of discussion, each field for H2, each control bit in H3, and the single field in V will be said to specify the execution of one micro-operation. The set of operations in a horizontal machine will be described as independent if, during the execution of each machine cycle, any bit pattern of the OP designates the execution of NH (for H1), or K (for H2) parallel micro-operations (the interpretation of the data part is also assumed to be independent of any micro-operations specified). Again, the concept of complete independence of micro-operations is unrealistic, but the degree to which the independence is constrained is an important concept in the following section.

Compilation models

In the present context, compilation is the process of translating statements expressed in a source language L into a sequence of microinstructions. One common compiler design divides this mapping into two separate translation phases. In the first, syntactic recognition and semantic analysis produce an intermediate form of the source, which will be referred to as a sequence of language primitives (lp's). Typical primitives include arithmetic and logical operations, register loads and stores, condition testing and branching, etc. In the second mapping, lp's are transformed into a sequence of microinstructions which may be executed directly.

In symbolic form (quadruples, triples, duos), lp's might resemble assembly language statements for a sophisticated target machine, with instruction addresses replaced by internal linkages and operand addresses replaced by symbol table pointers. The mapping from lp's to microinstructions could be one-to-one for simple operations, or one-to-many, for operations such as procedure calls or arithmetic operations, where a 16-bit adder might be used for 32-bit arithmetic.

Figure 2 illustrates three models for compilation, indicating the mappings described so far in the boxes labelled "translation" and "generation". The first (V) is the compilation process for a pure vertical machine. The generation of microinstructions is a process of selecting a sequence of (legal) OP bit patterns and data parts (the same as the code generation phase of a normal compiler). The ordering of the microinstructions may be modified, subject to data dependent constraints (input/output and encoded fields).

The second and third diagrams in Figure 2 represent compilation models for horizontal machines. The HI diagram represents the compilation process for a machine with totally independent micro-operations. The generation phase produces a stream of micro-operations. Since only data dependencies exist, the micro-operations may be reordered and optimized, much as the microinstructions for ease V. The composition phase combines MO's into microinstructions, utilizing data dependency information from the translation and generation phases. The complexity of the composition step is determined by the number of micro-operations within each horizontal control word.

As MO constraints are added, the situation approaches that illustrated by HD, where the generation and composition phases are inextricably bound, i.e., the rules governing the composition of MO's are so complex and interrelated that it is no longer useful to consider the generation of a stream of independent MO's. The emphasis shifts back to determining a sequence of legal OP bit patterns which perform some composite action. This analysis may be performed statically, considering the OP as one large operation code field, or dynamically, by searching for the required bit pattern on an MI-by-MI basis. In either case, the probability of determining a sequence of microinstructions which utilize all available resources most efficiently is low (or the cost is correspondingly high).

In any realistic case, MO constraints for horizontal machines place the compilation complexity somewhere between HI and HD. Two distinct phases may still exist, but the complexity of the composition function is increased to include resource, as well as data scheduling. It is useful to categorize the difficulty of microcode generation for a specific machine in terms of:

1. Combinatorial complexity proportional to the number of MO's within each word.
2. MO dependency in terms of number of shared resources which must be managed.
3. Timing dependency, i.e., the scheduling of operations which require more than one machine cycle to complete (for example, a core reference).

Any of these factors complicate the code generation process, and their combined effect is synergistic. In these simple terms, an ideal horizontal organization for the efficient compilation of microcode would utilize field encoded control (reducing the number of MO's per word), each field managing a disjoint resource set, with no timing dependencies. To some extent, this type of design is exemplified by the MCU and the AMP. A different approach to providing a horizontal host machine is the QM-1, where a horizontal (360 bit) nanocontrol word specifies the micro-operations to be executed within one major cycle, and an 18-bit field in the microinstruction selects a particular nanoword.

MICROPROGRAMMING LANGUAGE DESIGN

Discussion

The previous section concerned itself mainly with compilation from an unspecified language through to microcode for a
variety of machines. Within this section, the inverse problem is considered, i.e., given a host machine, is there a higher level language which may be defined to ease the microprogrammer's task, without exceeding some "reasonable" efficiency bound?

Microprogramming language levels

Categorizing the levels of microprogramming language support is a relatively difficult task, due in part to some of the peculiarities of horizontal machines, as outlined in the previous section. One hierarchical description would include:

1. Symbolic microinstruction assemblers—Straightforward field sensitive specification of control word contents.

2. Symbolic micro-operation assemblers—Each statement of the language corresponds to one microinstruction. Within a statement, micro-operations may be expressed symbolically (for example, in register transfer notation), AMIL.

3. Symbolic micro-operation languages—A sequence of MO statements is composed by the compiler into microinstructions.

4. Restricted language with micro-operation statements—A mixture of MO statements and language constructs which map (one-to-many) onto MO's. Typically, higher level constructs include flow of control (IF ... THEN ... ELSE, conditional DO's, etc.). Composition of MO's is performed by the compiler. The level of support is analogous to that provided by PL360, in that a detailed knowledge of the host machine's architecture is required. Examples are in References 2, 3 and 9.

5. Machine dependent languages—Higher level languages with some machine dependent features, but general symbolic facilities such as expressions, data aggregates (arrays, structures). Explicit specification of MO's and host machine registers would not be necessary.

6. Machine independent language—A higher level language with features specified independently of any particular machine architecture. For example, ALGOL or an ALGOL subset.

Levels 1 and 2 provide a representation of microinstruction sequences. If any composition of MO's is required, the programmer must perform this mapping himself. Thus, detailed knowledge of all MO and timing dependencies is required. Macro extensions of these languages could provide some higher level support, but the complexity of the composition phase may practically preclude any non-local optimization of microinstruction sequences.

Levels 3 and 4 provide a composition function within the language compiler. For horizontal machines, this removes a large burden from the microprogrammer, who may concentrate on the specification of operation sequences without explicitly recognizing many of the low level dependencies which exist within the hardware. A knowledge of host machine facilities (registers, local stores) and operations is necessary, but the flavor of these languages is similar to that of normal assembly language programming. Macro extensions from these levels could be extremely useful in building type 5 support.

Levels 5 and 6 may be similar in syntactic form. The distinction between the two is analogous to the differences in design between a general purpose programming language such as PL/I and a system programming language such as LSD. For level 5, some commonality in language features with a general purpose language is desirable, but constructs which would require a complex run time environment, or volumes of in-line code, are to be avoided (implicit data conversions, varying string operations, etc.). The process of feature selection as it relates to the design of a level 5 language for microprogramming is discussed in the following section.

Tailored languages

A language whose features are explicitly designed to coincide (to a large extent) with the hardware capabilities of its object machine will be referred to as a "tailored language". In the microprogramming context, this corresponds to defining a set of language primitives which map directly into micro-operations, on almost a one-for-one basis, and building a more expressive source language on this base.

For example, the set of arithmetic and logical operations supported in hardware define lp's for these binary and unary operations. At the source language level, these simple operations may be combined to form expressions. Main memory references might be defined as lp's which would be expanded to memory address register loads, and memory read/write micro-operations. This would allow the use of symbolic main store variables in the microprogramming language to be supported at the source level. Built-in functions could be included to reflect specific hardware capabilities (bit string selection, hardware instruction decoding primitives, etc.).

The extent to which this approach may be successful is determined largely by the architecture of the host machine. A complex microinstruction format (HD in the compiler model) would define a complex set of lp's, which would make translation from the source language difficult (or conversely, would define a strangely convoluted source language).

PUMPKIN—MCU SYSTEMS LANGUAGE

A tailored microprogramming language, PUMPKIN, has been designed for the Microprogrammed Control Unit (MCU) of the AN/UYK-17, currently under development by the U.S. Naval Research Laboratory. After briefly describing some of the hardware characteristics of the MCU, some features of the PUMPKIN base language are discussed.

* Language for Systems Development, a systems programmer's dialect of PL/I.

From the collection of the Computer History Museum (www.computerhistory.org)
Descriptions of the two additional levels of the PUMPKIN language (base + operating system primitives, extensible base), as well as a proposed base language manual are contained in Reference 21. To date, only the language definition exists. Future research will investigate some of the optimization and code generation techniques which would be used to implement a PUMPKIN compiler.

**MCU description**

The MCU is a 16-bit processor which functions primarily as the system controller of the AN/UYK-17 Signal Processing Element. Components of the MCU include:

- Two 16-word local stores;
- Two memory data channels which (separately) address words in 32-bit buffer stores;
- A field select unit (FSU), which may select any subfield (up to 16 bits) of a 32-bit word read from buffer store;
- An arithmetic and logical unit (ALU) with 16-bit operations such as addition, subtraction, logical and, or, exclusive or, complement, equivalence, plus a shift capability;
- 16 hardwired interrupt levels;
- Communication with and control of other system components (including other MCU’s) across a shared bus (Z-bus).

The MCU is a horizontal microprogrammed machine with a 64 bit wide control word composed of 16 encoded fields which determine source and destination address for parallel register transfers, ALU operations, parallel buffer store reads and writes, conditional microinstruction sequencing, etc.

The firmware development support for the MCU is a symbolic micro-operation assembler, AMIL, which includes a syntax macro capability. Programming the MCU requires careful consideration of operations which may proceed in parallel within one 150 nanosecond cycle.

**PUMPKIN base language features**

Excursions from the tailored concept (e.g., for the MCU, allowing stack operations) should be carefully examined in terms of (1) utility, (2) code generation possibilities, and (3) required run time overhead.

Definition of the primitives for the MCU is relatively straightforward. Arithmetic and logical operations (add, subtract, and, or, exclusive or, equivalence, complement, left shift, right shift, shift left circular, increment and decrement by one) on 16-bit operands, and substrings of 32-bit operands read from buffer store (maximum length 16 bits) are the primitive operations. Iterative loop control, interrupt generation/servicing, (one level) subroutine calls, and Z-bus communications all have hardware support. Extensions to these primitives include:

- Symbolic variables in buffer store or local store (submerging buffer address register operations and read/write code points in compiled code).
- Evaluation of arithmetic and logical expressions involving primitive operations.
- Unrestricted subroutine calls (number of levels).
- Flow of control primitives which involve generation of condition testing and branching code (IF ... THEN ... ELSE, DO WHILE, DO UNTIL), or address arithmetic (CASE).
- Data aggregates involving simple address calculations (one dimensional arrays, structures of bounded length—32 bits maximum).
- Based storage without dynamic storage allocation, i.e., allowing pointer qualifiers as in LSD, but restricting the use of based variables (including structures and arrays) to storage templates. Allocation of storage would be the explicit responsibility of the programmer.
- Automatic storage following a simple stack frame model to facilitate the writing of reentrant code (including interrupt service routines).
- Substring qualification on the left of assignments (i.e., setting bit substrings of words) which involves masking code, since the FSU selects bit substrings on input operations only.
- Low level interfaces with AMIL, i.e. hardware facility reservation and use, a CODE ... ENDCODE statement.

The syntactic form of PUMPKIN is similar to that of LSD, with some limitations (no character strings, data elements of 16 bits or less), and several built in functions corresponding directly to conditions (carry out, adder overflow) which may be tested within the MCU (see Appendix for an example of a simple PUMPKIN program). Constructs within the language are designed so that the translation phase can produce a simple sequence of micro-operations which implement the specific function. The language itself was designed by considering features which are supported by hardware resources. For example, the field select unit allows definition of bit substring operations and bit string components of structures with low overhead in terms of compiled code.

Other features (such as multiply/divide operators) would not be provided in the base language, but could appear: (1) within a well-defined superset of the base language (another language level), (2) utilizing an extensible version of the base language to define new data types and operators, (3) as a set of micro-subroutines callable from base language programs.

**REFERENCES**

12. MICRODATA 82/S Computer Programming Language MPL, Microdata Corporation, 73.
16. AN/UYK-7 Signal Processing Element Architecture (preliminary), Naval Research Laboratory, June 73.

APPENDIX

The following is an example of a simple PUMPKIN program. It is presented in the interests of providing some idea of the form of the language. For a formal description of the syntax, see Reference 21.

PROC PARSERX (RX);
DCL 1 RX DWORD, "32 BIT RX INSTRUCTION"
  2 OPCODE BIT(8), "OPERATION CODE"
  2 REG1 BIT(4), "SOURCE/TARGET REGISTER"
  2 STORAGE_ADDRESS BIT(20),
      3 INDEX BIT(4), "INDEX REG"
      2 BASE BIT(4), "BASE REG"
      2 DISPLACEMENT BIT(12).
DCL PARSED_OPCODE WORD IN LSB (8); "LOCAL STORE B, LOC 8"
DCL REGNO WORD IN LSB(9);
DCL EFFEC_ADDRH WORD IN LSB(10);
DCL EFFEC_ADDRL WORD IN LSB(11);
"SIMULATED GENERAL PURPOSE REGISTERS"
DCL 1 GPR (0:15) DWORD IN BSM1(100), "BUFFER STORE"
  2 HIGH WORD,
  2 LOW WORD;
"THIS ROUTINE IS PASSED A /360 RX TYPE INSTRUCTION AS A PARAMETER, STORES THE OPCODE AND REGISTER OPERAND IN LOCAL STORE, AND CALCULATES THE EFFECTIVE ADDRESS OF THE STORAGE OPERAND (BASE REGISTER + INDEX REGISTER + 12 BIT DISPLACEMENT). IT THEN CALLS EXECHRX TO EXECUTE THE INSTRUCTION."
PARSED_OPCODE ← OPCODE; "SET OPCODE IN LSB"
REGNO ← REG1; "SET REG # IN LSB"
EFFEC_ADDRH ← 0; "ZERO HIGH 16 BITS OF EA"
"CARRYOUT RETURNS THE VALUE OF THE ADDER CARRY WHEN EVALUATING THE EXPRESSION USED AS AN ARGUMENT. AS A SIDE EFFECT, THE 16 BIT RESULT MAY BE ASSIGNED WITHIN THE FUNCTION."
EFFEC_ADDRH ← CARRYOUT(EFFEC_ADDRL ← DISPLACEMENT+LOW(BASE)) + HIGH(BASE);
IF INDEX ≠ 0 THEN
  IF CARRYOUT(EFFEC_ADDRL ← EFFEC_ADDRL + LOW(INDEX))
    THEN EFFEC_ADDRH ← EFFEC_ADDRH + 1;
"ZERO HIGH BYTE OF SUM (24 BIT ADDRESSING)"
EFFEC_ADDRH ← (EFFEC_ADDRH + HIGH(INDEX)) & X'00FF';
CALL EXECRX; "EXECUTE THE RX INSTRUCTION"
END PARSERX;

"C. BERGMAN, AUG 73"