stichting
mathematisch
centrum

MC

AFDELING INFORMATICA IW 138/80 JUNI
(DEPARTMENT OF COMPUTER SCIENCE)

H.J. SINT

A SURVEY OF HIGH LEVEL MICROPROGRAMMING LANGUAGES

Preprint

kruislaan 413 1098 SJ amsterdam

Printed at the Mathematical Centre, 413 Kruislaan, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

1980 Mathematics subject classification: 68B99

ACM-Computing Reviews-categories: 1.3, 4.22

A survey of high level microprogramming languages*)

by

Marleen Sint

ABSTRACT

This paper surveys the current state of design and implementation of high level microprogramming languages. First, a number of important design issues is formulated. Next, four microprogramming languages are considered in detail, to see how each of them has approached these issues. Brief remarks are made about six other languages. Finally, some concluding remarks are made.

KEY WORDS & PHRASES: microprogramming languages, microcode compilers

*) This paper is not for review; it has been submitted for publication elsewhere.

1. INTRODUCTION

When a contemporary software designer, used to high level languages and structured programming, decides to exploit the microprogrammability of her new machine, she will soon imagine herself back in the mid fifties. At best, support provided by the manufacturer consists of a good manual, an assembler and a loader. In the worst case she has to manage with the hardware diagrams of the machine, and the listings of the microprograms for the basic (macro)instruction set (*). Moreover, programming in microassembly language turns out to be even more difficult than programming in conventional assembly language, especially when the machine happens to have a horizontal microarchitecture. There have been various attempts to design and implement higher level languages for microprogramming, but none of these has resulted in the production of a generally available compiler. This paper surveys high level microprogramming languages, emphasizing the problems which have yet to be overcome in order to change the situation just sketched.

It may seem surprising that the development of microprogramming languages is lagging so far behind that of macroprogramming languages, and that compiler construction, which for the macrolevel has become a mere routine, is still causing so many problems when code for the microlevel has to be generated. There are at least three factors which complicate compilation to microcode.

First of all, the structure of (horizontal) microcode is much more complicated than that of conventional machine code. Microprograms exercise almost direct control over the hardware. The parallelism inherent in the latter, which is largely invisible at the conventional machine level, is still visible at the microlevel. A horizontal microinstruction is composed of several microoperations. In principle, these are executed in parallel while consecutive microinstructions are executed sequentially. In general, timing is far from straightforward. The microoperations which together make up one microinstruction are not necessarily all initiated simultaneously, nor do they always take exactly one microcycle to complete. This may lead to possible overlap between the execution of consecutive microinstructions. (Most of the parallelism is hidden from the microprogrammer when a vertical encoding scheme is employed, but this usually implies a loss of flexibility and speed [5]).

Secondly, microcode has to meet much higher efficiency requirements than macrocode. Traditionally, microprogramming has been used for the realization of macroarchitectures. Since the efficiency of each computer system ultimately depends on the efficiency of the microprograms implementing the instruction set of that computer, virtually each effort

(*) The term "macro" in words like "macroprogram", "macroassembler", "macrolanguages" etc. will refer to the conventional machine level as opposed to the microlevel.

spent on speeding them up will pay off in the end. Though less extreme, efficiency requirements for user microprograms are still strict. The only reason to write microprograms is to gain speed; if a compiler is unable to produce sufficiently efficient microcode it is of no use at all.

There is yet a third factor which complicates microcode generation: Most machines sold as microprogrammable are so only to a limited extent. The microarchitecture is usually tailored towards the efficient implementation of the standard (macro)instruction set of the machine. But efficiency and generality seldom go together. Microprogramming such machines is therefore often like writing a text editor in a language designed especially for matrix manipulation: the beautiful features that are available are of no use, and the ones needed are not provided. Only a few machines, of which the different models of the Burroughs B1700/1800 series are the best known examples, provide real hardware support for user microprogramming, but the architecture of these machines is vertical.

The next section contains an overview of existing microprogramming languages, preceded by a list of important design issues. This paper is concerned with user microprogramming, which implies that microarchitecture and control word format are fixed. The overview is therefore restricted to languages which can be implemented on existing hardware, as opposed to specification languages to assist in the design of microprogrammed hardware. Furthermore, it only considers the most general class of languages: those which can be implemented on machines with a horizontal architecture.

A final section contains some concluding remarks.

2. HIGH LEVEL MICROPROGRAMMING LANGUAGES

2.1. Design issues for microprogramming languages.

From a language design point of view it is not at all obvious why the design goals for high level microprogramming languages should be much different from the usual design goals for macroprogramming languages. They should be well structured, they should have sufficient expressive power to be useful within their specific area of application, and they should be as machine independent as possible. It is not even obvious that a special microprogramming language is needed at all - most algorithms executed on the microlevel can be expressed in existing languages. A subset of PASCAL or ALGOL 68, with all constructs requiring elaborate runtime support removed, would probably do very well.

For the reasons mentioned in the introduction, construction of a satisfactory compiler for such a language is not yet within reach. Evaluating existing languages only in terms of expressive power and machine independence is therefore unrealistic; the implications of certain design choices for the optimization and code generation phases of a compiler should be taken into account as well.

A list of important design issues will now be formulated. It will be used as a guideline for the evaluation of existing languages. The first issue provides some overall perspective. The next four issues are of particular importance for microprogramming languages and should be considered with the pragmatical background just sketched in mind. The last three issues are more general and are applicable to any programming language.

2.1.1. What are the design goals for the language?

The use of a high level microprogramming language may serve two different purposes:

(1) To relieve the programmer from dealing with irrelevant, low-level details of a specific microarchitecture.

(2) To reduce the chance of errors in the microprogram.

Though design goals of most microprogramming languages state some combination of these two, the emphasis varies. (Extreme examples are on the one hand YALLL [16], which is almost exclusively concerned with the first purpose, and on the other hand S* [4], which is almost exclusively concerned with the second.)

To achieve the first purpose, the language may include features like sequential program specification, the use of symbolic variables instead of register names, and the specification of computations in terms of a fixed set of primitives instead of the set of microoperations available on a specific machine. Machine independence means that the programmer is relieved from all details of a specific architecture, with the consequence that one and the same program can be compiled and executed on different machines. In practice, machine independence in this sense is only seldom a design objective. Some authors [1,18,23] use the term in a quite different sense: they call their microprogramming language machine independent because it allows specification of programs for different machines, but programs written in these languages can only be executed on the machine for which they were designed and are therefore machine dependent. In the context of high level languages, this is a somewhat unusual meaning of the term "machine independent".

Language features included to achieve the second purpose, i.e. to facilitate writing correct microprograms, include a good control structure and good data-structuring facilities. A language may be designed in such a way that there is a strong correlation between "syntactically correct" and "meaningful" programs, which is rarely the case with (micro)assembly languages. A language may even enforce some form of program correctness proof.

Verification of microprograms has received more attention than verification of macroprograms (see for example, the remarks made in [6]). There are several reasons for this. Microprograms are traditionally placed in read-only-memory, which means that bugs are hard to correct. Microprograms are at the lowest level in the system hierarchy; they form the basis for the remainder of the system and hence their reliability is crucial. Finally, microprograms are small and simple in comparison with macroprograms. The first two facts make verification attractive; the last one makes it feasible as well.

2.1.2. What kind of primitive operations are provided?

A compiler for a particular machine has to map the primitives provided in the language to the set of microoperations available on that machine. Because the generated object code should be highly efficient, it is in general not acceptable if a compiler overlooks possibilities to apply certain microoperations. For example, a macroprogram must preferably be kept in a hardware register with auto-increment capability. Only if such a special register is not available in the microarchitecture, it is acceptable to use the main ALU to increment it.

Utilization of machine primitives is most easily guaranteed, when the language primitives are of at least the same complexity as the microoperations to which they must be compiled. For a machine independent language, this implies that the set of primitives should be chosen such, that it covers most microoperations on most machines. There is, however, such a variety of hardware features available on different machines that this approach will render the language bulky, to say the least. Moreover, such a large set of primitives is bound to be redundant (in the sense that different, but closely related primitives will be included), which in itself will make code-generation difficult. This is best illustrated by an example:

On the Interdata 3200 the programmer can switch to a different block of 32 registers, by setting 3 bits in the program status word (there are eight such blocks). This is very useful for the implementation of a macro procedure call instruction; such a block can be made to contain the current activation record or at least part of it. This feature could be incorporated in a high level language by means of a "new-block" operation, with one (integer) operand specifying the selected block. There is a definite overlap between such a "new block" facility and a "push stack" operation, which is also hardware supported on several machines. Even if both primitives are included, a program may contain uses of the "push"-operation which, on an Interdata, should be translated using its "new-block" facility. As this is very hard to detect, there is still no guarantee that both primitives will be efficiently translated.

From a language design point of view it is much more attractive to include only a small set of primitives along with the possibility to declare new operations in terms of already existing ones. If this approach is taken, there will exist microoperations on some machines which are more complex than the language primitives. In order to utilize these operations, the compiler must be able to recognize that _a sequence_ of source statements can be translated into _one_ microoperation. 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.

Both approaches therefore cause the implementer problems. The easiest way out is to allow in a program exactly the primitive operations available on the target machine, but this implies that machine independence is sacrificed.

2.1.3. _To what extent are variables viewed as machine registers?_

When the language allows the use of symbolic variables in the same manner as conventional high level languages, the compiler must allocate registers for them. There are two factors which complicate this task:

- The number of registers exclusively accessible to the microprogram is limited. It may vary from 16 (e.g. on the DEC VAX-11) to 256 (e.g on the Control Data 480). Temporarily storing variables in a reserved area of main memory will sometimes be unavoidable, but should be done in such a way that the number of fetches and stores is minimized.

- The microregister set is generally not homogeneous. Allocating a variable to a certain register at a certain program point, also determines which subset of microoperations can be applied to that variable at that point. Constraints imposed can be bizarre, for example, the fact that a certain microregister is occupied may disable a part of the microinstruction set. In order to allocate registers without hampering efficient code generation, the compiler needs some insight in the use (for example, access frequency) of variables.
  Especially this factor renders register allocation much more complex at the microlevel than at the macrolevel.

In many microprogramming languages the allocation problem is completely avoided by requiring the programmer to bind all variables used to the physical registers of the target machine. The association between variable and register may range from very simple (each variable denotes one specific register) to fairly complicated (a variable can denote a field within a register, or the concatenation of several registers). Such a restriction again introduces machine dependencies, and decreases programming convenience as the programmer has to keep track of which value resides in which register, and has to be aware of datapaths between registers.

2.1.4. Is parallelism implicit or explicit?

When the language allows sequential specification of the source program, the compiler should decide which source statements can be executed in parallel in order to be able to compose the horizontal microinstructions. Two forms of dependence must be taken into account:

- Data dependence.
  When a statement S1 creates a value used by a statement S2, or, alternatively, when S2 destroys a value needed by S1, S1 must be executed before S2.

- Resource dependence.
  Statements S1 and S2 cannot be executed in parallel if their resource usage may lead to conflicts, for example, if they both use the (same) ALU or both write into the same register.

Several algorithms have been developed to compose a minimal or, using heuristic methods, a near minimal sequence of microinstructions from a sequence of microoperations (without branches), see for example [18,22,3,21]. The algorithm presented in [21] deserves special attention; it employs a very general model of microinstructions. Such algorithms can be used not only for microinstruction composition during compilation of a sequential source program, but also for the optimization of hand-written microprograms.

Register allocation and microinstruction composition are interdependent. In order not to block possibilities to execute operations in parallel, a register allocation phase should introduce as little resource dependencies as possible between statements which are not data dependent. This is an additional complication in the implementation of languages which allow both sequential program specification and the use of symbolic variables.

A language can incorporate explicit parallelism in two different ways:

- The programmer must denote which statements are not data dependent, i.e. could be executed in parallel if an unlimited number of resources were available.

- The programmer should take both data dependence and resource dependence into account.

The first alternative is a compromise between programming and implementation convenience. It does not, like the other one, exclude the use of symbolic variables, while it relieves the compiler from a non-trivial analysis.

2.1.5. What is done about interrupts and microtraps?

Microprograms control the machine to a much greater extent than macroprograms. On microprogrammable machines, this is a potential source of trouble. User microprograms will often have to be developed and executed in a multi-programming environment, and will often coexist with a set of unalterable, manufacturer supplied microprograms which interpret the basic instruction set. The possibly disastrous effect of user microprogramming on system reliability is well known - in general nothing will keep a microprogram from blowing up the operating system. The necessity to service interrupts once in a while, and the possible existence of microtraps which are not completely transparent to the microprogrammer (see below for an example), are complications which stem from this same source.

If the execution of a microprogram may take long compared to the cycle time of the machine (think for example of a fast fourier transform) it must periodically check whether any interrupts are pending, and if so, transfer control to an interrupt handler in order to allow them to be serviced. There may exist a fixed, standard macroinstruction to return from an interrupt, in which case special provisions may be required in the user microprogram to ensure that it will indeed get back control. It should moreover be able to resume its execution at the exact point where it left off.

Microtraps (like, on some machines, the occurrence of a pagefault) cause even more trouble. Consider the following trivial microprogram specification:

```
program incread(n)
begin reg[n] := reg[n]+1; mbr := readmem(reg[n]) end
```

It increments the contents of reg[n] which are subsequently used as an address in main memory. The memory fetch may lead to a pagefault. The microprogram will be restarted from the beginning after the pagefault has been taken care of. If reg[n] corresponds to a register which is also part of the macroarchitecture and is therefore saved and restored, it will be erroneously incremented a second time.

These problems are too complicated and require a too detailed analysis to justify a full treatment in this survey paper. It should be clear however that a language designer will have to decide whether their solution is left to the compiler or to the programmer. If the programmer is allowed to disregard them completely, the compiler must be able to determine suitable program points at which to test for interrupts. In addition, it must insert special code at these points to ensure that the program is correctly restarted upon return. If the machine has microtraps, the compiler must locate all program points where they can occur and determine whether a trap at such a point will lead to undesirable side-effects.

Handling of interrupts and traps was one of the first problems suggested to me in the context of compilation of high level microprogramming languages, but no attention whatever is paid to it in the papers I surveyed. Therefore, although it deserves being on the list of design issues, it will not be further mentioned in the next section.

2.1.6. What kind of control structures are provided?

Subroutines and expressions deserve special attention. Allowing formal parameters and local variables introduces a certain amount of space and time overhead for each procedure call, which may not always be acceptable. Allowing arbitrarily complex expressions leads to the introduction of temporary variables during compilation, which complicates register allocation.

2.1.7. What kind of data types and data structures are provided?

As a consequence of the fact that microprograms are primarily concerned with (fixed length) bitstrings, most microprogramming languages have only one datatype. Datastructuring facilities greatly enhance program readability. Inclusion in a language of e.g. arrays and PASCAL-like records does not necessarily cause much trouble to a compiler writer. If fields in a record structure can designate bit-fields in registers, the compiler will have to introduce temporary variables in order to deal with field selection.

2.1.8. Has the language been implemented and if so, what results have been obtained?

This is not a design issue, but for the evaluation of a language, obtained results are as important as incorporated ideas.

2.2. Existing High Level Micro Languages

The boundary between high level and low level microprogramming languages is not very well defined. There exist several languages [13,15] which might be called high level as far as their control structure is concerned (they include for example if-then-else, while-do and case constructs, or allow sequential specification of the program), but which are definitely low level as far as their primitives are concerned (precisely the resources and microoperations of one specific machine). In the following, I only consider languages which are not tailored toward a single machine. This criterion is rather arbitrary; its main justification lies in the fact that such languages can be expected to be of interest to a larger group of readers. A detailed evaluation of four languages is presented. Taken together, they yield a comprehensive view on the different ways in which the problems mentioned in the previous section are approached. The design goals of each language, along with the examples, have been taken from the cited papers. Section 2.2.5. contains some brief remarks about six other languages.

2.2.1. SIMPL (Single Identity Micro Programming Language)

SIMPL [18] dates from 1974, and was developed at the University of California at Berkeley by C. Ramamoorthy and M. Tsuchiya .

Design goals.

The design goals of SIMPL follow from the following quotations:
"The [envisaged] high level language should enable the user to write microprograms in a conventional, sequential and procedural fashion and permit these programs to be compiled into efficiently executable microcode."
"The desirable properties of a high-level microprogramming language must be a compromise between machine dependence, ease of detecting and representing explicit and implicit parallelism, and the innate ´naturalness´ required of all programming languages to establish effective man-machine communications".

Primitives.

A fixed set of operators is included: addition, subtraction, logical and, or, exclusive or and negation, shift (both linear and circular), as well as relational operators. Explicit read and write statements are provided for references to main memory. When the target machine has microoperations of a higher level than those included in this set, it may be difficult to utilize them (see 2.1.2). This problem is not addressed.

Variables.

Variables are identified with machine registers. An equivalence statement is provided in order to enable the programmer to refer to a register by more than one name.

Parallelism.

A SIMPL program is specified sequentially. One of the underlying concepts of SIMPL is the single assignment rule, well known from dataflow languages [11]. This rule, which states that each variable may occur only once as the destination in an assignment, facilitates detection of potential parallelism, but it is incompatible with the register view of variables as it would imply that each register can contain only one value throughout the program. The single identity principle was invented to resolve this conflict. In single assignment languages, the order in which the source statements are specified is completely irrelevant. The order in which they are executed is determined only by their data dependencies: A statement which uses the value of a certain variable x, will not be executed before the (only) statement which assigns a value to x is executed. In SIMPL, the order of the source statements is used to distinguish the different values assigned to a variable. Consider the following statement sequence:

```
(S1)   x := v1;
(S2)   ...
 .
 .             (S2 to Sn do not contain an assignment to x)
 .
(Sn)   ...
(Sn+1) x := v2;
```

The single identity principle states that S1 should be executed before any Si (2<=i<=n) which uses x; and each such Si should be executed before Sn+1. Application of these rules for all variables yields a partial ordering of the source statements. Statements without a precedence relation can be executed in parallel. Loops cause trouble in this scheme; it is not completely clear from [18] how this problem was solved.

Control Structure.

The control structure of SIMPL resembles that of ALGOL 60. Procedures and blocks are allowed. It is not mentioned in [18] whether or not they may contain declarations, but the examples suggest that they may not. Expressions may contain only one operator. If-then-else, while-do and (probably) for-statements are included; but goto´s are not. A case-construct has been added to permit multiway branches, which are available on many machines.

Datatypes and datastructures.

The only datatype in SIMPL is the integer. No data-structuring mechanism is provided (presumably, the language does not even permit array declarations, because arrays cause trouble in relation with the single assignment rule).

Implementation.

A compiler for SIMPL was written in SNOBOL4. It produces code for the Tucker-Flynn dynamic microprocessor. As this was the first effort to translate a sequential program to horizontal microcode, algorithms had to be developed to detect potential parallelism and possible resource conflicts between microoperations. Unfortunately, no comparison has been made between code generated by the SIMPL-compiler and handwritten code.

Example.

Now follows a simplified version of the example given in [18]: the multiplication of two 64-bit floating point numbers. Both numbers are assumed to be positive. Mantissa and exponent overflow are not taken into account. The multiplicand resides in register R1, the multiplier in R2, and the product in R3. Floating point numbers have the following format (from high to low order bits): sign (1 bit), exponent(13 bits), mantissa (50 bits). M3 and M4 extract the exponent and the mantissa respectively.

```
begin
       comment extract and determine exponent for product;
       R1 & M3 -> ACC;          comment logical and;
       R2 & M3 -> R4;
       R4 + ACC -> ACC;
       R3 | ACC -> R3;          comment logical or;

       comment extract mantissas and clear ACC;
       R1 & M4 -> R1;
       R2 & M4 -> R2;
       R0 -> ACC;

       comment multiplication proper by shift and add;
       while R2 ≠ 0 do
       begin ACC ^ -1 -> ACC;    comment shift 1 to the right;
             R2 ^ -1 -> R2;
             if UF = 1 then R1 + ACC -> ACC;
                comment UF equals lowest bit shifted out of shifter;
       end;

       comment pack exponent and mantissa into f.p. format;
       R3 | ACC -> R3;
end
```

Conclusions.

SIMPL was the first language that allows sequential specification of horizontal microprograms. Its design triggered many similar efforts. The fact that the language was implemented and that the compiler actually produced horizontal code was very important. SIMPL is now mainly of historical interest. Especially its limited view of variables and the absence of even the most basic data-structuring mechanism make it awkward to use. It is also doubtful whether it can be implemented with an acceptable result on a machine that supports operations not included in the basic operator set.

2.2.2. EMPL (Extensible microprogramming language)

EMPL [8] dates from 1976, it was developed by David DeWitt at the University of Michigan.

Design goals.

Four goals are stated: 1) "The language must facilitate writing programs"...; 2) It "should be readable, so as to facilitate the task of redesigning a program"...; 3) It "should be machine independent so that programs written in it are portable"; 4) It "should be possible to compile the language into very efficient microcode for a variety of microprogrammable computers".

Primitives.

A small set of basic operators is included: addition, subtraction (both one- and two-complement), unary minus, multiplication, division, logical and, or, exclusive or and the logical negation of these three, negation, shift, rotate, and the standard relational operators. There are no primitives for reading from or writing to main memory. The user can declare additional operators. Such a declaration should always have a body expressing the operation in terms of built-in or previously declared operations; but if the machine has a special microoperation for it, this can be specified as well. (Machine testable conditions could be handled by the same mechanism if an operator is allowed to yield a value of type "logical"; this is not explicitly stated however.) This mechanism allows a user to tailor the set of operators to the available hardware, while retaining a certain degree of machine independence: if a certain microoperation is not available, the body of the operator will be compiled statement-by-statement (but see the remarks on implementation below).

Variables.

Variables in EMPL are not machine registers. All variables are global, in order to avoid procedure calling overhead.

Parallelism.

Programs are specified completely sequentially. There are no provisions in the language to facilitate detection of parallelism.

Control Structure.

The language provides both procedure declarations (without formal parameters) and operator declarations (with an arbitrary number of formal parameters). Only simple variables and constants are allowed as actual parameters. Statement forms provided are assignment, call and return, if-then-else, while-do and goto. Expressions are not allowed to contain more than one operator.

Datatypes and datastructures.

The only basic datatype is the integer. A concept akin to the SIMULA class [2], called an extension statement, is used for datastructuring. It allows the user to define a new datatype with associated fields (of a previously declared type), an initialization statement, and associated operators. Fields can be referenced only from the operators declared within the extension statement; they cannot be selected from outside the class. This is consistent with the view that primarily those datatypes which are hardware supported, are declared in an extension statement. No difference is made in the language between variables residing in registers and variables residing in main memory. As there are neither explicit statements for memory references, it is not clear how an EMPL-program can for example read instruction operands from main memory.

Implementation.

EMPL has not been fully implemented. A sketch of a two-pass compiler is contained in [8]. The first pass translates the source text to a sequence of microoperations, without allocating resources. The second pass allocates resources and composes microinstructions; this pass has been completed and is described in [9]. It is hard to judge whether the results of a complete compiler would meet efficiency requirements. Some possible problems are:

- In the proposed implementation, a call to an operator which is not hardware supported is textually replaced by the statements that form its body. This is undoubtedly done to avoid parameter passing. If the operator mechanism is heavily used, this will lead to an increase in the size of the produced code.

- I doubt whether the control statements are sophisticated enough to allow optimal microinstruction sequencing. There is neither a case-construct nor a cascaded conditional statement (if t1 then s1 elif t2 then s2 ... etc). Multiway branches will therefore be hard to utilize.

Example.

The example shows an EMPL extension statement to declare the type "stack", followed by the declaration of a stack object. The microoperation specifications are in a format based on a control word model developed earlier by the same author [7]. After the stack "address-stk" has been declared, the initialization statement of the type declaration is executed. It sets the associated stackpointer to zero.

```
TYPE STACK
     DECLARE STK(16) FIXED;  /* an array of 16 integers */
     DECLARE STKPTR FIXED;
     DECLARE VALUE FIXED;

     INITIALLY DO; STKPTR = 0; END;

     PUSH: OPERATION ACCEPTS(VALUE)
           MICROOP: PUSH 3 0;    /* indicates to the compiler that a  */
                                 /* PUSH microoperation is available */
           IF STKPTR = 16
           THEN ERROR;     /* overflow */
           ELSE DO; STKPTR = STKPTR + 1; STK(STKPTR) = VALUE; END
           END;
     END,

     POP:  OPERATION RETURNS(VALUE)
           MICROOP: POP 3 0;
           IF STKPTR = 0
           THEN ERROR;     /* underflow */
           ELSE DO; VALUE = STK(STKPTR); STKPTR = STKPTR - 1;
           END;
     END,
ENDTYPE:

DECLARE ADDRESS_STK STACK;
```

Conclusions.

From all microprogramming languages considered in this survey, EMPL most closely resembles a conventional high level language. By grouping all operations on objects of a single datatype in a class, program modularity and portability are improved. The way in which the user may invoke specific microoperations is elegant. The specification of a "machine dependent microoperation" in an operator body guarantees a certain degree of machine independence while retaining the chance to produce reasonably efficient code.

Viewing classes mainly as a way to extend the language with datatypes and operations that are supported by specific microoperations, is unnecessarily restrictive. It has caused useful constructs like field selection to be excluded from the language - compare the EMPL fields for example with the tuples of S*. It has also lead to an inefficient implementation scheme for classes and associated operators that are not hardware supported. The desirability of a case-construct has been mentioned already. The integer as basic datatype does not seem to be the best possible choice (again, compare with S*).

2.2.3. S*

S* [4] dates from 1978, it was developed by Subrata Dasgupta at Simon Fraser University.

Design goals.

Three design goals are mentioned: "(1) the ability to construct control structures for designating clearly, and without ambiguity, both sequential and parallel flow of control; (2) the ability to describe and name arbitrarily, microprogrammable data objects or parts of such data objects; (3) the ability to construct microprograms whose structure and correctness can be determined and understood without reference to any control store organization."

Primitives.

S* is described as a language schema, rather than a complete language. It consists of a framework providing a declaration structure and a set of compound statements with associated semantics in the form of pre- and postconditions. For a given machine M, S* is instantiated to a complete language S(M), the elementary statements of which are determined by the microoperations of M. Registers, constants residing in read only memory and testable machine conditions must be declared explicitly by the user before they can be referenced. In order to complete the semantics of S(M), it may be necessary to declare additional pre- and postconditions. For instance, the increment operation of S* is defined on arbitrarily large integers:

```
{X+1 = v} INC X {X = v}
```

In a specific instantiation S(M) allowance will have to be made for the possibility of overflow and the above rule will have to be modified accordingly. For a 16-bit two-complement integer range it is modified as follows:

```
{X+1 = v & v < 32768} INC X {X = v};
{X+1 = v & v = 32768} INC X {X = -32768 & OVERFLOW = 1}
```

These pre- and postconditions should be used by the programmer, to prove that her program is correct, i.e. that it conforms to a specification in the form of additional sets of pre- and postconditions.

Variables.

In S*, declaration of a variable is meaningless. In an instantiation S(M), each variable must, in its declaration, be associated with one or more specific machine registers or main memory locations of M. This association can be quite complex. It is possible for example to declare an array consisting of five elements corresponding to the low order 4 bits of register R1 through R5.

Parallelism.

Parallelism is explicit, the programmer has to compose the microinstructions herself. This is a logical choice in view of the design goals, which emphasize verifiability and well-structuredness of programs rather than simplification of microprogramming by raising the level of primitives. Three constructs to aid in microinstruction composition are provided in the language:

cobegin S1; S2; ... ; Sn coend

denotes that the (elementary) statements S1; ...; Sn should all be executed in the same microcycle.

cocycle S1; S2; ... ; Sn coend

is a construct which is meaningful in S(M) only if execution of a microinstruction on M is split into n phases. It denotes that S1 to Sn should all be placed in the same microinstruction, and that S1 should be executed in the first phase, S2 should be executed in the second phase, etc. The execution of the microoperations S1 to Sn should not overlap. Si can be a cobegin-coend statement.

dur S0 do S1; S2; ... ; Sn end

specifies that S0 runs concurrently with the sequence S1 to Sn. S0 takes a maximum of n microcycles to complete.

Control Structure.

The syntax of S* is basically that of PASCAL. Parameterless procedures are allowed. In the declaration, the procedure name must be followed by a parenthesized list of the variables used in the body. Procedures and blocks may contain local declarations, but of course the variables must be linked to machine registers. Expressions can be arbitrary complex. Statement forms provided are a cascaded if-statement (if t1 then S1 elif t2 then S2 ... elif tn then Sn fi), while-do and repeat-until, call and return, sequential (begin-end) and parallel block structures (as specified above), and a region-end statement which delimits a hand-optimized section where the flow of control may not be changed by the compiler. Tests are elementary statements occurring only in an instantiation S(M); they must correspond to hardware testable conditions of M.

Datatypes and datastructures.

Datatypes and datastructures.

The only primitive datatype in S* is the bit. There are four ways to construct new datatypes:

seq [i..j] bit
denotes a bitstring with i and j as high and low order indices. Arithmetical, logical and shift operations have their standard meaning on bitstrings.

array [i..j] of type
denotes a sequence of elements of an arbitrary type.

tuple field1: type1; ... fieldn: typen; end
corresponds to the PASCAL record. Field selection is done as in PASCAL with one addition: if X is a tuple data-object and all fields in X are of type seq [] bit, then a reference to X without selection refers to the concatenation of all fields. This is very convenient; if IR is a variable of type "instr" with fields opcode, addr and index, then one can refer to the complete instruction (IR) as well as to separate fields (IR.opcode).

stack [i] of type with ident {, ident}
declares a stack of depth i, with stackpointers denoted by the identifiers.

Implementation.

No S(M) has been implemented so far, but the level of the elementary statements suggests that writing a compiler is certainly feasible. Although not mentioned in [4], an automatic verifier to check the validity of the program proof provided by the user, would fit very well in an S(M) implementation.

Example.

The following example shows a microprogram for the multiplication (performed by repeated addition) of two positive, 16-bit integer operands, "mpr" and "mpnd". The result of the multiplication is stored in a data-object called "product". In [4] the complete development the program is shown, including the necessary conditions to assure correctness. Here only the final result is presented.

```
program MPY;
        var localstore: array [0..31] of seq [15..0] bit;
        const minus1 = dec (16) -1;
              # a 16-bit constant with decimal value -1 #
        var left_alu_in, right_alu_in, aluout: seq [15..0] of bit;
```

```
                syn mpr     = localstore[0],
                    mpnd    = localstore[1],
                    product = localstore[2];
                    # Three locations of localstore are renamed #

                begin
                     repeat
                          cocycle
                               cobegin left_alu_in := product;
                                       right_alu_in := mpnd
                               coend;
                               alu_out := left_alu_in + right_alu_in;
                               product := alu_out
                          end;
                          cocycle
                               cobegin left_alu_in := mpr;
                                       right_alu_in := minus1
                               coend;
                               aluout := left_alu_in + right_alu_in;
                               mpr := aluout
                          end
                     until aluout = 0;
                end
         end
```

Conclusions.

Instantiations of S* can be expected to be well structured languages with well defined semantics, which can be an important aid in the development of reliable microprograms. Although a program written in any specific instantiation S(M) is highly machine dependent, the good structure of the S* schema will facilitate the transformation of programs from one instantiation to another. Because parallelism and timing are explicit, the programmer must have intimate knowledge of the specific machine for which the program is written.

2.2.4. YALLL (Yet Another Low Level Language)

YALLL [16] dates from 1979. It was developed by a team from the University of California at Berkeley.

Design goals.

..."Rather than to try to bridge the gap between a [machine independent] HLL to microarchitecture in one step, we have designed a low level language that is capable of producing microcode for different machines..."

Primitives.

The primitives of YALLL correspond to commonly available microinstructions: add, subtract, increment, decrement, and, or, exclusive or, negation, several types of shift, load from and store into main memory, register-to-register move, and put-constant-in-register. It is hoped that these primitives will permit compilation to microcode for a large class of interesting machines.

Variables.

Variables are viewed as general purpose registers with the exception of "mar" (memory address register) and "mbr" (memory buffer register). The compiler should take care of the correct utilization of special purpose microregisters. A program is preceded by a declaration part in which YALLL register names are bound to physical machine registers. It is not clear from the description whether binding is required for all variables, or whether it is optional.

Parallelism.

The program is specified completely sequentially.

Control Structure.

In accordance with the above mentioned design goals, the structure of YALLL is that of a conventional assembly language. YALLL includes a conditional and an unconditional jump, a procedure call and return, an exit-with-value instruction, and a multiway branch. This branch facility is fairly sophisticated, it allows the comparison of the contents of a register with a constant mask that may contain ´true´, ´false´ and ´dont-care´ bits.

Datatypes and datastructures.

There are no datatypes and consequently no data-structuring facilities in YALLL. Five types of constants exist: binary, octal, decimal and hexadecimal numbers, and masks.

Implementation.

YALLL has been implemented on two different machines, the DEC VAX-11 and the Hewlett-Packard HP300. Several example programs were tried out on both machines and the results were compared with each other and with equivalent hand-written code. The HP implementation performed a lot better than the VAX implementation. The baroque structure of the VAX micro architecture, combined with the complete lack of documentation for this machine at the time [16] was written, discouraged the implementers from attempting any code optimization. Another interesting observation is that both compilers consisted of about 5000 lines of high level language code. This suggests that a full optimizing compiler for a high level microprogramming language of the complexity of EMPL for example, will be huge.

Example.

The following example shows a program which transliterates a character string according to a table. The string is addressed by register "str", and ends with a null byte. The table is addressed by register "tbl". Each byte of the string is examined, and, if not zero, is replaced in memory by the byte in the table which it addresses. The program shown is that for the HP300. It differs from the VAX-version only in the declaration part (which binds YALLL registers str, tbl and char to HP registers db, sb and mbr respectively) and in the lay-out conventions.

```
     reg str = db
     reg tbl = sb
     reg char = mbr

loop:
     load char,str               ;get addressed character
     jump     out if char = 0    ;quit if zero
     add mar,char,tbl            ;add to table base adress
     load char,mar               ;fetch character from table
     stor char,str               ;replace character in string
     add str,str,1               ;bump string address
     jump     loop
out: exit
```

Conclusions.

The value of YALLL lies in its moderate design goals which enabled the designers to produce implementations for two different machines. This has never been done for any other machine independent microprogramming language. The fixed set of primitives included in YALLL will at least for some machines render the generation of efficient code difficult.

2.2.5. Other languages

This survey will be concluded with some brief remarks about an additional six languages. Their approach to any of the issues considered above is not fundamentally different from that of the previous four languages.

MPL [10] represents the earliest effort to design and implement a high level microprogramming language, with the objective to generate code for a vertical machine. Its structure is comparable to that of SIMPL, but it offers somewhat better data-structuring facilities. It permits the declaration of one-dimensional arrays and virtual registers consisting of the concatenation of physical ones. When [10] was written only part of the compiler had been completed.

Strum [17] can be considered a somewhat less general forerunner of S* as far as its design goals are concerned. Its primitives are based on the Burroughs D-machine [19]. Programs are developed together with their proofs; they should contain assertions which can be used to generate verification formulas, the validity of which can be checked by an automatic verifier. A non-optimizing compiler was completed.

MPGL [1] is comparable to SIMPL or MPL. Programs are specified sequentially and entirely in terms of machine primitives. Its structuring facilities are rather poor; the programmer has to specify details like which registers the compiler should use to store intermediate results during expression evaluation, and at what address in control memory a procedure should be placed. One feature of MPGL is unique, however. A complete machine specification is part of the program and the compiler uses this specification to generate code. Thus, the MPGL compiler can be fed with a program for any machine having a microarchitecture that can be described in the specification formalism of MPGL. Note, that this does not imply that MPGL programs are machine independent. An MPGL-compiler has been written in assembly language. It produces efficient code. For the examples presented in [1], code size did not increase by more than 15% in comparison with equivalent hand written microprograms.

A machine independent (nameless) language, tailored towards emulator and interpreter construction is described in [14]. It allows declaration of all kinds of primitives (registers, stacks) in the emulated machine. The authors have emphasized design rather than implementation problems and this will make efficient implementation of the language rather difficult.

CHAMIL [23] is a PASCAL-based language. Its control structure and its data structuring mechanism are adequate. Like most other languages, CHAMIL identifies variables and physical machine registers. The programmer is allowed to abstract from physical datapaths: the statement "reg_a := reg_b" is legal as long as there exists a (possibly indirect) path from reg_a to reg_b that can be traversed within one microcycle. The microprogrammer must group statements into microinstructions. A compiler for CHAMIL was written in PASCAL.

Finally, the development at IBM of a microcode compiler for a subset of PL/I (called PL/MP) should be mentioned. Unfortunately, too little information is available on this project to allow a more elaborate treatment. Reports [20] and [12] which both treat implementation issues, suggest that the language is not register oriented.

3. CONCLUSIONS

The greatest problem in high level language microprogramming is not the design, but the implementation of high level languages. The attention paid to various implementation problems is rather unbalanced.

From the ten languages reviewed in the previous paragraphs, eight allow complete sequential specification while only two (S* and CHAMIL) leave composition of microinstructions to the programmer. On the other hand, only two or three (EMPL, PL/MP and in a certain sense YALLL) allow the programmer to work with symbolic variables instead of physical registers - a feature which also simplifies programming and is a prerequisite to achieve at least some machine independence. No language allows the passing of parameters to subroutines. The problem of microinstruction composition is far from trivial. Complicated algorithms have been developed to make this language feature possible. There is no reason to assume that the register allocation problem is intrinsically more difficult, or that its solution would be less useful, but for some reason or other it received far less attention. Another substantial problem, the incorporation of interrupt and trap handling, has even been completely neglected.

I think it is justified to state that the instruction composition problem, though important in its own right, has been overemphasized. Two circumstances may have encouraged this course of events: first, the fact that algorithms for microinstruction composition can be applied to both hand-written and compiler generated microcode, and second, the fact that the designers of the first language for horizontal machines (SIMPL) almost exclusively concentrated on the composition problem. Since composition depends on used resources, it will be clear that the alternative in which the programmer has to specify microinstruction composition while the compiler allocates resources, is not possible. It may be worthwhile though to investigate further the compromise suggested in section 2.1.4 in which the programmer has to specify data dependencies between statements but can leave resource allocation and therefore also the treatment of resource dependencies to the compiler.

A final remark concerns the emphasis on efficiency. Experience with conventional high level languages has shown that high level language programs, even when compiled by a highly optimizing compiler, are generally less efficient than assembly language programs written by an expert. The same is likely to be true in the area of microprogramming.

Whether or not the loss of efficiency is compensated by the advantages of high level language programming, will depend on the circumstances. A user may find it more attractive to speed up a heavily used procedure by a factor of five with comparatively little effort and without needing expert knowledge, than to gain a factor of ten only after mastering a complicated microassembly language which requires an intimate knowledge of machine details. Ideally, high level microprogramming languages and conventional high level languages should coincide, so that a user only has to denote which procedure(s) should be compiled to microcode and which to macrocode. Such an ideal is still far from realization however. It is likely that microprogramming will remain the business of only a few experts, until languages of at least the level of EMPL will have been satisfactorily implemented.

References

1. Baba, T., "A Microprogram Generating System - MPG," Information Processing 77, pp. 739-744 North Holland Publishing Company, (1977).

2. Dahl, O. B. and Nygaard, K., The Simula 67 Common Base Language, Norwegian Computing Centre (1970).

3. Dasgupta, S. and Tartar, J., "The Identification of Maximal Parallelism in Straight-Line Microprograms," IEEE Transactions on Computers, Vol. C-25, (10) pp. 986-991 (Oct 1976).

4. Dasgupta, S., "Towards a Microprogramming Language Schema," Proceedings of the 11-th Annual Workshop on Microprogramming, pp. 144-153 (1978).

5. Dasgupta, S., "The Organization of Microprogram Stores," Computing Surveys, Vol. 11, (1) pp. 39-65 (March 1979).

6. Davidson, S. and Shriver, B. D., "An Overview of Firmware Engineering," Computer, Vol. 11, (5) pp. 21-33 (May 1978).

7. DeWitt, D. J., "A Control Word Model for Detecting Conflicts Between Microprograms," Proceedings of the 8-th Annual Workshop on Microprogramming, pp. 6-12 (1975).

8. DeWitt, D. J., "Extensibility - A New Approach for Designing Machine Independent Microprogramming Languages," Proceedings of the 9-th Annual Workshop on Microprogramming, pp. 33-41 (1976).

9. DeWitt, D. J., "A Machine Independent Approach to the Production of Optimized Horizontal Microcode," Ph.D. Thesis, University of Michigan (June 1976).

10. Eckhouse, R. H., "A High Level Microprogramming Language (MPL)," AFIPS Conference Proceedings, Vol. 38, pp. 169-177 (1971).

11. Gelly, O. and others,, "Lau System Software: A High Level Data Driven Language for Parallel Programming," Proceedings 1976 International Conference on Parallel Processing, p. 255 (Aug 1976).

12. Kim, J. and Tan, C. J., "Register Assignment Algorithms for Optimizing Micro-code Compilers--Part I," Report RC7639, IBM T.J. Watson Research Center, Yorktown Heights (March 1979).

13. Lloyd, G. R., "PUMPKIN - (Another) Microprogramming Language," SIGMICRO Newsletter, Vol. 5, pp. 15-44 (April 1974).

14. Malik, K. and Lewis, T., "Design Objectives for High Level Microprogramming Languages," Proceedings of the 11-th Annual Workshop on Microprogramming, pp. 154-160 (1978).

15. Marti, J. B. and Kessler, R. R., "A Medium Level Compiler Generating Microcode," Proceedings of the 12-th Annual Workshop on Microprogramming, pp. 36-39 (1979).

16. Patterson, D., Lew, K., and Tuck, R., "Towards an Efficient, Machine-Independent Language for Microprogramming," Proceedings of the 12-th Annual Workshop on Microprogramming, pp. 22-35 (1979).

17. Patterson, D. A., "Strum: Structured Microprogram Development System for Correct Firmware," IEEE Transactions on Computers, Vol. C-25, (10) pp. 974-985 (Oct 1976).

18. Ramamoorthy, C. V. and Tsuchiya, M., "A High-Level Language for Horizontal Microprogramming," IEEE Transactions on Computers, Vol. C-23, (8) pp. 791-801 (Aug 1974).

19. Reigel, W., Farber, V., and Fisher, D. A., "The Interpreter - A Microprogrammable Building Block System," AFIPS Conference Proceedings, Vol. 40, pp. 705-723 (1972).

20. Tan, C. J., "Code Optimization Techniques for Micro-code Compilers," AFIPS Conference Proceedings, Vol. 47, pp. 649-655 (1978).

21. Tokoro, M., Tamura, E., Takase, K., and Tamaru, K., "An Approach To Microprogram Optimization Considering Resource Occupancy and Instruction Formats," Proceedings of the 10-th Annual Workshop on Microprogramming, pp. 92-108 (1977).

22. Tsuchiya, M. and Gonzalez, M. J., "Toward Optimization of Horizontal Microprograms," IEEE Transactions on Computers, Vol. C-25, (10) pp. 992-999 (Oct 1976).

23. Weidner, T. G., "CHAMIL - A Case Study In Microprogramming Language Design," SIGPLAN Notices, Vol. 15, (1) pp. 156-166 (Jan 1980).