© EUROMICRO
EUROMICRO Journal 5 (1979) 219-224

1979a

Towards Machine-Independent Microprogramming

J. F. Sommerville
Dept. of Comp. Science, University of Strathclyde, Livingstone Tower, 26 Richmond Street, Glasgow G1 1XH, UK

This paper describes a high-level microprogramming language called SUILVEN, which has been implemented to generate microcode for a B1700 computer. SUILVEN has facilities for both program structuring and data structuring and contains no machine-dependent features. Although designed as a machine-description and implementation language, SUILVEN may also be used as a general-purpose programming language. Examples are presented illustrating both of these applications and the implementation of the language is briefly discussed.

1. INTRODUCTION

Since the widespread introduction of micro-programmable machines there has been a good deal of research effort expended in the development of higher-level microprogramming languages. These languages have varied from what are essentially extended assemblers such as Burroughs MIL [2], to fairly sophisticated derivations of existing high-level languages such as Eckhouse's MPL [7]. A recent survey paper by Bushell [3] describes MPL, SIMPL [11] and PUMPKIN [8], three of the more ambitious higher-level microprogramming languages. These languages are tailored, at a high-level, for a specific machine and, as far as is known by the author, only DeWitt [6] has described a machine-independent microprogramming language.

DeWitt's language, EMPL, uses an interesting approach based on the class concept of SIMULA [4]. The language is designed to be extensible so that it may take advantage of special micro-instructions available on the underlying machine. The extensibility mechanism works by allowing the user to define types, consisting of data structures plus associated operations. For example, a type STACK with associated operations PUSH and POP may be defined and it is the task of the EMPL compiler to map these operations onto the underlying machine as efficiently as possible. If the machine supports a hardware stack, PUSH and POP may be implemented as single micro-instructions, otherwise they must be implemented as a sequence of more primitive instructions. DeWitt surmised that there were no insurmountable obstacle to the implementation of EMPL but, at the time of writing, had not produced a language translator.

In designing a machine-independent micro-programming language it is important to establish for what purpose the language is intended. Firstly, we do not consider that a machine-independent microprogramming language is an adequate substitute for a machine-oriented language if optimally efficient microprograms are required. The difficulties of mapping a language onto often vastly different architectures are such that implementing a commercial machine in this way is not economically realistic.

We see a need for a machine-independent microprogramming language however and have identified four principal application areas for such a language:

(i) As a machine-description language formally defining a machine architecture. This description may be manually translated into a machine-oriented language.

(ii) As a research tool allowing research workers to quickly produce and evaluate experimental machine architectures.

(iii) As a portability tool which can be used to emulate a particular architecture on another machine. This could be achieved without the loss of performance associated with software emulation yet at a much lower cost than producing a hand-coded emulator. "One-off" implementations of interpreters would become economically feasible.

(iv) As a tool for the systems programmer allowing him to extend a machine instruction set for his own application. If the microprogramming language is sufficiently close to common high-level programming language, the programmer should be able to accomplish this tailoring without detailed knowledge of the underlying micro-architecture.

Unlike many high-level microprogramming language implementations, we have not put the generation of optimally efficient microcode as our principal design criterion. Rather we have tried to produce a language which facilitates the production of well-structured programs, which is easy to read and learn, which is relatively easy to implement, and which does not contain blatantly inefficient constructs.

The language described here is not described formally but the features of the language are

illustrated by example. After describing the language we briefly discuss its implementation and uses.

2. THE MICROPROGRAMMING LANGUAGE SUILVEN

The microprogramming language described here developed from an earlier machine description language called SMDL [13]. Although originally intended as a description language for "soft" or abstract machines SUILVEN was extended and formalised to make it suitable for compilation into microcode. The language provides facilities for the programmer to specify exactly the size and structure of the data elements, allows programs to be split into independent procedures, and has control constructs conducive to structured programming. The goto statement is not a feature of the language but facilities are provided for "emergency" exit from loops and procedures. A SUILVEN program consists of a set of data declarations, a set of procedure declarations, and a set of SUILVEN statements.

2.1. SUILVEN Data Declarations

The data declarations in SUILVEN allow the programmer to define arbitrarily-sized data areas and to ascribe structure to these areas. A type is not associated with a declared data element – each element is considered as a string of bits. The SUILVEN compiler allocates addresses to each data area thus removing the necessity for a run-time interface between a SUILVEN program and its associated machine.

There are four types of declarations in SUILVEN, BITS declarations, TEMPLATE declarations, DEFINE declarations and FLAG declarations.

BITS declarations allow the programmer to define data areas of a precise size. For example:

```
BITS (32) A,B,C;  BITS (4) P,Q;
```

This declares 3 32-bit areas A, B and C and 2 4-bit areas P and Q.

Elements specified in a BITS declaration may be grouped into arrays. For example:

```
BITS (24) ARRAY (8) SCRATCHPAD;
```

This declares an array of 8 24-bit elements called SCRATCHPAD.

TEMPLATE declarations are structure specifiers which define a structure that may be later ascribed to a data element defined in a BITS declaration. For example:

```
TEMPLATE LIST_ELEMENT =
GC (1), VAL (24), PTR (16);
```

This defines a structure called LIST_ELEMENT which consists of 3 fields – a 1-bit field called GC, a 24-bit field called VAL and a 16-bit field called PTR.

A DEFINE declaration specifies which areas are associated with which templates. Naturally there is no requirement that a data element should be given a template. The following example illustrates some declarations that might be made in a program emulating an IBM S/360 computer. Text enclosed in quotes is taken as a comment by the SUILVEN compiler.

```
"Defines registers"
BITS (32) ARRAY (16) R;
"Defines program status register"
BITS (64) PSW;
"Internal address registers"
BITS (4) BR,IR,DR;
"PSW Structure"
TEMPLATE PROG_STATUS =
SYSMASK (8), PROT_KEY (4), AMWP (4),
INTERRUPT (16),
ILC (2), CC (2), PROGMASK (4),
PROG_COUNT (24);
DEFINE PROG_STATUS : PSW;
```

Having assigned a template to a data area, the user may access the fields within the area by name. For example:

```
PSW.CC := 0;
PSW.PROG_COUNT := PSW.PROG_COUNT+4;
```

Alternatively the WITH notation, derived from PASCAL [16], may be used to access the fields of a structure:

```
WITH PSW DO
 £(
     CC := 0;
     PROG_COUNT := PROG_COUNT+4;
£)
```

The advantage of using the with notation is that the base address of the data object being accessed may be held in a register rather than recomputed for each assignment.

Flag declarations define single-bit variables which may be used to monitor exception conditions. For example:

```
FLAG INTEGER_OFLOW, DIV_BY_ZERO;
```

Most microprogrammable machines have a set of one-bit registers indicating conditions such as interrupts. For any implementation of SUILVEN, these bits are given names by the SUILVEN compiler and the user may access them as FLAG variables using these standard procedures TRUE and FALSE and given values using procedures SET and UNSET. For example:

```
IF TRUE (ANY-INTERRUPT) THEN
   UNSET (ANY-INTERRUPT)
```

This code tests the bit named ANY_INTERRUPT. If the bit is one, it will be switched off.

As well as data declarations, SUILVEN allows the user to define macros. A macro may be used to give a name to a constant or a string. For example:

```
MACRO STACKSIZE = '300';
MACRO STACKFLOW = '5, "STACK OVERFLOW"'
```

Alternatively, a macro may be used to name a section of code. For example:

```
MACRO PUSH = 'IF SP > STACKSIZE THEN
INTERRUPT(STACKOFLOW) ELSE
£( STACK[SP]:=ACC;  SP := SP+1;  £)'
```

2.2. SUILVEN Procedures

The procedure mechanism in SUILVEN allows the programmer to structure his program into named code sections and subsequently activate that code. Procedures may contain local data declarations but procedure declarations may not be nested.

There are two types of procedure in SUILVEN — pure procedures which are simply named sections of code and functions which return a value. Parameters may be passed to procedures either by value or by reference and local data and data structures may be defined. Examples of procedure headings are:

```
PROCEDURE PUSH(BITS(WORD_SIZE) VALUE V;);
FUNCTION  POP (  );
PROCEDURE UPDATE_DESCRIPTOR(BITS(WS)DESC;);
```

A procedure body consists of local declarations and SUILVEN statements and is terminated by the reserved word END. A function result is returned by writing the expression returning the result after the procedure ending. For example:

```
END = STACK[S];
```

Return from within a procedure may be accomplished using an EXIT statement.

A particularly useful feature of the procedure mechanism is the ability to redefine the structure of global data areas within the procedure. For example, consider an abstract machine programmed in SUILVEN which has a single 32-bit accumulator ACC. The machine has both integer and character operations. ACC would be declared a global variables and procedures INTEGER_OPS and CHARACTER_OPS could be declared. Within INTEGER_OPS we would make the following declarations:

```
TEMPLATE INT = SIGN (1), VAL (31);
DEFINE   INT : ACC;
```

This specifies that within INTEGER_OPS, ACC is to be regarded as having the structure of an integer variable. Within CHARACTER_OPS we would make the declarations:

```
TEMPLATE CHARS = C1(8),C2(8),C3(8),C4(8);
DEFINE   CHARS : ACC;
```

ACC is now considered as being made up of 4, 8-bit characters, thus allowing operations to be programmed in a natural fashion.

2.3. SUILVEN Statements

The statements available to the SUILVEN programmer are those common to a number of high-level languages viz assignment statements, procedure statements, and control statements. The control statements are an if statement, a while statement, a case statement, and a special repeat statement which allows exit to be made from within the body of the loop.

This repeat statement has the form:

```
REPEAT <statement1>
UNTIL <condition> DO <statement2>
```

Execution of a repeat statement proceeds as follows:

```
L1 : <statement 1>;
     if <condition> then goto L2;
     <statement 2>
     goto L1;
L2 :
```

Notice that if statement 2 is null (a permitted option) the repeat statement reduces to the more common repeat - until construct.

The repeat statement was included in SUILVEN to facilitate the programming of loops whose terminating condition occurs after some input but before information processing. Such loops may be clearly and efficiently programmed in SUILVEN. For example:

```
REPEAT
    READ.A.RECORD (  )
UNTIL
    TRUE (EOF) "Tests for end-of-file"
DO
    PROCESS_RECORD (  );
```

Other SUILVEN statements are illustrated in the example below which is abstracted from the SUILVEN implementation of a PASCAL machine described by Nori [10]. As well as illustrating individual statements the example gives the reader an idea of the style of a SUILVEN program.

There are no input/output statements defined in SUILVEN as we consider them to be a machine-dependent feature. In our version of the language, input and output is implemented using standard procedures which link to machine I/O routines. We envisage that I/O will be implemented differently for each machine on which SUILVEN is implemented. As SUILVEN programs seem to contain only a few I/O statements, we do not consider this to be a severe constraint on the portability of SUILVEN programs.

2.4. SUILVEN Examples

In this section, we present examples of SUILVEN code which have been abstracted from an implementation of an abstract machine interpreter for PASCAL. A complete listing of this interpreter is not given but rather sections of the program which illustrate SUILVEN features are described.

The abstract PASCAL machine (P-machine) is a fairly conventional stack machine with a reverse-Polish instruction set. Instructions may have zero parameters or one parameter. As well as the usual arithmetic. logical, load/store and jump operators the P-machine has a number of instructions specifically designed to support features of PASCAL. These include set operators, an instruction to check the range of a variable and an instruction to copy whole arrays.

Each machine instruction is structured into 3 fields - the operation field (OP) and fields called P and Q. Only the OP field is used in all instructions - P and Q are only used by one-address instructions such as LOAD.

In our implementation of the P-machine we have chosen to implement the top 2 stack elements as

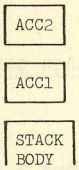

Fig. 1. The P-machine stack organisation.

global registers ACC1 and ACC2 as shown in Fig. 1.

Simulations indicated that this arrangement eliminated 90% of store accesses compared with implementing the stack wholly in store. Naturally stack access routines are a little more complex but we believe that the performance improvement resulting from fewer store accesses more than compensates for this.

The first example below (Fig. 2) illustrates the main loop of the P-machine interpreter. This procedure has a local bits variable called INSTRUCT_REG which acts as the current instruction register. This register is structured, via a template declaration, into the 3 instruction fields OP, P and Q. The fetch-execute loop of the interpreter is controlled by the global flag variable ENDPROG. Instructions are fetched and executed until ENDPROG is set by a STOP instruction.

In each loop iteration, we check for interrupts, handle them if necessary, fetch an instruction from store, update the program counter PC, ensure registers ACC1 and ACC2 are full and finally, choose the appropriate code for execution using the instruction op-code as a case switch.

Figures 3 and 4 give examples of procedures which are part of the P-machine interpreter. The PUSH

```
PROCEDURE P_CODE_INTERPRETER (  );
      BITS(WS)  INSTRUCT_REG;
      TEMPLATE PINST = OP(OP_SIZE),P(P_SIZE),Q(Q_SIZE);
      DEFINE   PINST  :  INSTRUCT_REG;
      INITIALISE (  );
      WHILE FALSE (ENDPROG) DO
         £(
            IF TRUE (ANY_INTERRUPT) THEN
               HANDLE_INTERRUPT (  );
            INSTRUCT_REG := STORE [PC];
            PC := PC+1;
            "The stack is implemented using 2 registers ACC1 and ACC2
            as the top elements.   We always fill these registers if
            an arithmetic/logical instruction is to be executed."
            IF INSTRUCT_REG.OP < 37 THEN
               FILL_REGISTERS (  );
            CASE INSTRUCT_REG. OP OF
               ACC1 := ACC1 & ACC2;                "AND"
               ACC1 := ACC1 | ACC2;                "IOR"
                        :
               PUSH (STORE[INSTRUCT_REG.Q]);       "LDC"
                        :
               IF POP ( ) = FALSE THEN
                  PC := INSTRUCT_REG.Q;            "FJP"
               SET (ENDPROG);                      "STOP"
            ENDCASE;
         £);
      CLOSE_DOWN_GRACEFULLY (  );
END; "P_CODE_INTERPRETER"
```

Fig. 2. The Fetch/Execute Loop of the P-code Interpreter.

```
PROCEDURE PUSH(BITS(WS)VALUE V;)
      IF FALSE(ACC1FULL) THEN "Both registers empty"
         £( SET(ACC1FULL); ACC1:=V; £)
      ELSE
         £(
            IF TRUE(ACC2FULL) THEN "Both registers full"
               £(
                  IF SP > HP THEN "Test for stack overflow"
                     £( SET(ANY_INTERRUPT); SET(STACKOFLOW); £)
                  ELSE
                     £( STORE[SP]:= ACC1;  "Push into store"
                        SP := SP+1;  ACC1 := ACC2;
                     £)
               £)
            ELSE
                     SET (ACC2FULL);
            ACC2:=V;
         £)
END;  "PUSH"
```

Fig. 3. The Stack PUSH Procedure of the P-machine.

procedure (Fig. 3) adds items to the stack and the procedure MOV (Fig. 4) executes the P-machine instruction to copy arrays.

The procedure PUSH adds items to the top of the stack and this may necessitate copying the contents of ACC1 and ACC2 into the store portion of the stack. Flags ACC1FULL and ACC2FULL are used to ascertain if the top of stack registers contain needed information. Notice that it is impossible for ACC2 to be full and ACC1 empty.

PASCAL allows the assignment of arrays, providing they have been declared to be the same type. For example arrays declared:

var a, b : array [0..9] of integer;

may be assigned as follows:

a := b

and this will entail the copying of all elements of b to the corresponding elements of a. The procedure MOV in the P-machine accomplishes this operation. Its parameter Q indicates the number of elements to be copied, with the base addresses of the arrays on top of the stack. After completing the copy, the array base addresses are discarded.

3. THE IMPLEMENTATION OF SUILVEN

As SUILVEN is a machine-independent language, it is important that its compiler be carefully constructed to generate code tailored to the microprogrammable host machine. The present system generates microcode for the B1700 computer series. The structure of this machine [15] with bit addressability and a vertically encoded mi-

```
PROCEDURE MOV (  );
      BITS(WS) COUNT; COUNT := 0;
      WHILE COUNT < Q DO "Copy all elements"
         £(
            STORE[ACC2+COUNT]:=STORE[ACC1+COUNT];
            COUNT := COUNT+1;
         £)
      UNSET (ACC1FULL); UNSET(ACC2FULL);
END; "MOV"
```

Fig. 4. The MOV Procedure from the P-machine Interpreter.

croinstruction set make it particularly suitable as a host machine for a high-level microprogramming language. The syntax of SUILVEN has been carefully designed for simple lexical analysis and to permit an LL (1) parsing technique to be used [1].

As SUILVEN does not contain features which rely on the B1700 architecture there are no obstacles to implementing the language on other machines. The BITS declarations of SUILVEN take advantage of the bit addressability of the B1700 but should the language be implemented on, say, a byte oriented machine it would be the responsibility of the compiler to translate bit-oriented declarations to the corresponding byte-oriented representation.

The design of SUILVEN makes no assumptions whatsoever about the availability of fast registers and the allocation of registers/store is entirely the responsibility of the compiler. In order to aid the compiler in making decisions on register allocation we have included in our compiler a set of compiler directives which allow the programmer to specify the variables to be located in the fast scratchpad registers of the B1700. For example:

```
#SPAD PROGRAM_COUNTER, ACC1, ACC2, SP
```

would specify that the above variables be held in the scratchpad. Scratchpad allocation may be static, as above, or a variable, normally retained in store, may be copied into the scratchpad for part of the program. This latter feature is particularly useful if a variable is used frequently in a routine but not outside that procedure. We have dicovered that allocating variables to the scratchpad when required can decrease the number of microinstructions in a program by about 25% and provide a threefold increase in execution speed. Other optimisations include "remembering" register contents and not reloading a register with its value and replacing instruction sequences, where possible, by a single, special-purpose instruction.

As discussed in section 1, the production of optimal microcode was not the principal requirement of our project. Hence the compiler only performs simple optimisations of small code sections (peephole optimisations [9]) possible in a one-pass system. As uncontrolled jumps are not permitted in SUILVEN the language is particularly amenable to global data flow optimisations [1] and we would hope to include this in future compiler implementations. We do not, however, envisage that microcode generated from a SUILVEN program will be as efficient as that generated from a machine-oriented language.

4. SUILVEN IN USE

In section 1, we identified four principal functions of a machine-independent microprogramming language. These are as a machine description language, as a research tool, as a portability tool, and as a tool for the application programmer.

We have so far used SUILVEN for two of these applications – as a machine description language and for the construction of experimental machine architectures. The same program both defines the machine and emulates the machine code. To date we have built machines for PASCAL, based on the P-machine described by Nori [10] and for SASL [14], an expression language. These implementations are described fully elsewhere [12].

To give an indication of the power of SUILVEN, implementation of the PASCAL machine in SUILVEN, PASCAL and machine code may be compared. The SUILVEN version of the system consists of 570 statements generating 2342 microinstructions. This is comparable to the PASCAL implementation (about 400 statements) but very much smaller than the machine-code version of the same system (about 2000 instructions).

This discrepancy in program size is reflected in the time taken to program each implementation of the system. The implementation of the SUILVEN version of the system took about six weeks whereas the machine-code system was not complete after four months of work. Although less efficient than the machine code system, initial performance indications suggest that the SUILVEN system is about three times faster than a software version of the interpreter.

We envisage, however, that machine-independent microprogramming will be most useful as an application programming tool. In a recent paper Dearnley [5] has described enhancements made to the microinstruction set of Data General computer, to improve its performance as a database machine.

To make such enhancements requires considerable knowledge of the host machine microinstruction set – a considerable learning task for the applications programmer. If a microprogramming language similar to commonly used high-level languages is available, the programmer's burden is lightened as it is much easier to learn such a language than to master the intricacies of a micro-architecture. Although the resultant microcode may be less efficient than hand-tailored code, it would almost certainly be more efficient than a software implementation of the same function.

As an example of how SUILVEN might be used for this purpose, consider an application which involves a considerable amount of table searching. The machine running this application could have its instruction set enhanced by a binary search instruction shown in Fig. 5.

Although the extension of machine instruction sets requires changes to language processors we believe that for large projects such as database implementation that it is worthwhile. Indeed such enhancement may be the only way to produce acceptable performance in systems requiring the processing of large amounts of data.

5. CONCLUSIONS

We have described the design of a micropro-

```
FUNCTION BINARY_SEARCH(BITS(WS) VALUE KEY, TABSIZE; BITS(WS) ARRAY TABLE );
      BITS(WS) INDEX, UPPER, LOWER;
      LOWER := 0 UPPER := TABSIZE;
      WHILE LOWER LEQ UPPER DO
         £(
            "In SUILVEN, all operators have the same precedence and
            expressions are evaluated from left to right.   A right
            shift implements integer division by 2"
            INDEX := LOWER + UPPER SHR 1;
            IF TABLE[INDEX] = KEY THEN
               EXIT = INDEX
            ELSE
               IF TABLE[INDEX] < KEY THEN
                  UPPER := INDEX-1
            ELSE
                  LOWER := INDEX+1;
         £)
END = KEYNOTFOUNDVALUE;
```

Fig. 5. A Binary Search Procedure in SUILVEN.

gramming language which does not contain machine-dependent features. Although the generated microcode is not optimal, we believe that the advantages of a machine-independent microprogramming system are such that the area is worthy of further research. In particular we believe that such a language is particularly useful to the programmer wishing to tailor an instruction set to a particular application. We can envisage a future situation where the programmer may select from a library of microprograms in the same way as he may select from a subroutine library code today.

The present version of SUILVEN has very little provision for string manipulation. Constructs to handle character strings are therefore a desirable extension to the language, if it is to be used for instruction set enhancement.

The present implementation of SUILVEN was designed simply to demonstrate the feasibility of the language. Further developments in this research should include the re-implementation of the compiler to generate an intermediate code which may be optimised and from which code for different machines may be generated. In conjunction with this we hope to examine and measure the performance gain achieved by extending machine instruction sets.

REFERENCES

[1] A.V. Aho and J.D. Ullman, Principles of Compiler Design (Addison-Wesley, 1977).
[2] Burroughs Corporation Micro-Implementation Language (MIL) Reference Manual, 1972.
[3] R.G. Bushell, Higher Level Languages for Microprogramming, Euromicro Journal 4 (1978) 67–75.
[4] O.B. Dahl and K. Nygaard, The Simula 67 Common Base Language, Norwegian Computing Centre, 1970.
[5] P.A. Dearnley, The Development of User-Level Microprograms for the Data General Eclipse Computer, Euromicro Journal 4 (1978) 82-86.
[6] D.J. DeWitt, Extensability - a new approach for designing machine-independent microprogramming languages, ACM SIGMICRO Newsletter 7 (September 1976).
[7] R.H. Eckhouse, A High-Level Microprogramming Language (MPL), AFIPS SJCC (1971) 169-177.
[8] G.R. Lloyd, PUMPKIN (Another) Microprogramming Language, SIGMICRO Newsletter 5 (1974).
[9] W.M. McKeeman, Peephole Optimisation, CACM 7 (1965) 443-533.
[10] K.V. Nori et al., The Pascal P Compiler: Implementation Notes, Technical Report 10, Eidgenössische Technische Hochschule Zürich.
[11] C.V. Ramoorthy et al., A High-Level Langauge for Horizontal Microprogramming, IEEE Transactions on Computers C-23 (August 1974).
[12] J.F. Sommerville, An Experiment in High-Level Microprogramming, Ph.D. Thesis, University of St. Andrews, Scotland (1977).
[13] J.F. Sommerville, Soft Machine Description Language, TR/75/8, Department of Computer Science, University of St. Andrews, Scotland (1975).
[14] D. Turner, SASL Language Manual, CS/75/1, Department of Computational Science, University of St. Andrews, Scotland (1975).
[15] W.T. Wilner, The Design of the B1700, AFIPS FJCC (1972) 489-497.
[16] N. Wirth, The Programming Language PASCAL, Acta Informatica 1, 1 (1971) 35-63.