COSC460
Honours Research Project
Department of Computer Science
University of Canterbury

High Level Language for Microprogramming on the Eclipse S/130

Supervisor: Dr. R.E.M. Cooper

Soon Sow Lai

24.9.81

Acknowledgments

I would like to thank my supervisor, Dr. R.E.M. Cooper for his invaluable guidance and support in this project.

I would also like to thank Miss Sarah Frampton for typing this report.

Contents

1. Introduction

During the past two decades, microprogramming has evolved to be a standard mechanism for implementing the complex standard instruction set of a computer. With the advent of the writable control store, the number of microprogrammable computers has steadily increased and subsequently, there has been a significant growth of interest in user microprogramming. Despite the steadily increasing rate of interest, the microprogramming environment remains very much the way it was in the early years of the art.

This project surveys the problems associated with the current environment for user microprogramming. One appropriate solution to the problem would be to provide a higher level of programming through higher level language support. With this approach in mind, the main objective of this research is to propose a language that can be used efficiently for microprogramming on the Data General Eclipse S/130. Considerable effort has been expended on this area of research in the past few years and a study is made of the current state of the design and implementation of high level microprogramming languages. The final phase of the project involved the consideration of aspects of compiler construction for such a high level language, particularly the problems of efficient microcode generation.

2. The Microprogramming Environment

Much of the current microprogramming is done in a machine-dependent low-level microassembly language. The symbolic instructions are translated on a one to one basis into executable microinstructions. For machines with a vertical microarchitecture, this is very similar to conventional assembly level programming. Machines with a horizontal microarchitecture allow a user to specify several parallel microoperations within a microinstruction. For the rest of this paper, a horizontal microarchitecture is assumed and only languages developed for such a general machine architecture will be considered. Problems associated with microprogramming include the following:

- A microprogram exercises almost direct control over the hardware of a machine. Effective use of this low-level art of programming requires a firm understanding of the microarchitecture of the machine. Consequently, this results in microprograms that are very machine-dependent and hardly portable from one machine to another without major conversion.

- Machines with horizontal microarchitecture offer the user more effective and efficient hardware control via micro-parallelism. However, no optimization is carried out by the microassembler so the efficiency of a microprogram is determined by the skill and effort of the programmer. Furthermore, the microoperations are not strictly parallel - intricate timing constraints govern the execution of these operations. The microoperations are not necessarily initiated simultaneously nor do they necessarily terminate together. Some microoperations e.g. a memory reference may take more than one microcycle to complete.

- Most machines do not provide real hardware support for user microprogramming - the microarchitecture of the S/130 is tailored towards the efficient implementation of the standard machine instruction set. Some of the available features are so specialized that they can hardly ever be used while those features that are generally needed are not available.

With such minimal support for user microprogramming, it is not surprising that the art of microprogramming is still restricted to a few experts. The work is both complex and unwieldy resulting in inefficient and unreliable programs as the size of the problem increases.

As with traditional software development, one approach is to remove the programmer further away from the level of the hardware. The ideal solution would be a machine-independent high level microprogramming language that can be used efficiently on a variety of machine architectures.

There has proved to be a number of bottlenecks hindering the design of such a language, all of which are attributable to the nature of microprogramming itself.

2.1 High level language for microprogramming

The design of higher level language support for microprogramming faces problems that not normally encountered in the software domain.

The need for the programmer to work at a low level has hindered the development of machine-independent languages. This led most researchers to design a machine-dependent language for each machine. While such a tailored language can be used efficiently on one particular machine, its machine-dependency restricts its importance.

More importantly, microcode has to meet much higher efficiency standards than macrocode. The main motivation to microprogram is to gain speed, hence, a compiler that generates unoptimized codes is generally unacceptable. Optimization of microcodes is a non-trivial problem.

Microprogramming, by definition, is software control of the underlying hardware of a host machine and we cannot microprogram without having in mind some real machine interface. With the large variability in microarchitectures, a machine-independent language must be general enough to be capable of generating microcodes for a large variety of machines. But generality seldom goes along with efficiency and we are faced with the conflicting objectives of machine-independency and efficiency.

2.2 Design Goals for high level microprogramming languages

Thus, it seems intuitively obvious that the criteria by which we design a macroprogramming language may be quite different from those by which we design a microprogramming language. In the design of a macroprogramming language, the objectives of ease of use and expressive power are of paramount importance.

A good design of any language should certainly stress the achievement of these objectives but in the domain of microprogramming, the language design is also restricted by factors which appear more as constraints than as objectives.

The following are some of the more important factors which should be taken into consideration in the design of a language for microprogramming:

1. facilitates the writing of microprograms in a sequential, structured and procedural manner.
2. improves the readability and reliability of the microprograms.
3. relieves the user from all the minor details of a specific microarchitecture.
4. should be as machine-independent as possible such that the language can be used to express microprograms for a large variety of machines and that the programs written are transportable from one machine to another with little or no conversion.
5. should be feasible to construct a compiler for the language which will generate efficient codes for execution on a target processor - efficient in terms of size and execution time of the codes.

3. Examples of Microprogramming Languages

The past two decades have seen several languages designed to improve the microprogramming environment. These belong to a hierarchy of languages, ranging from machine-oriented languages to machine-independent high level languages.

In this section, three languages (Schema, YALLL and EMPL) are reviewed. Unlike machine-oriented languages, these three languages were designed to be as machine-independent as possible and are hence, of wider interest.

3.1 Schema[1]

S. Dasgupta
Simon Fraser University, 1980.

Design rationale

The inherent machine-specificity of microprogramming has hindered the development of machine-independent high level languages. To avoid the neccessity of developing an individual tailored language for each microprogrammable machine, a language schema approach is adopted. The schema, S* is not a complete language but rather, it is a framework consisting of a declaration structure and a set of control structures.

For a given microarchitecture M_1 , S* is instantiated with respect to M_1 subject to the hardware constraints of the machine to produce a machine-dependent but fully defined language, $S(M_1)$.

Data types and data structures

The basic data type in S* is the bit. New data structures are constructed from this data type.

1. seq [i .. j] bit
   denotes a sequence of bits.
2. array [i .. j] of <type>
   denotes a vector of elements of an arbitrary type.
3. tuple
   fieldl : <type 1> ;
   :
   fieldn : <type n> ;
   end

3. (Continued)
   a tuple corresponds to a Pascal record.

4. stack [i] of <type> with <identifier>, <identifier>, ...
   denotes a stack structure of i elements with the identifiers as stack pointers.

Variables

Every symbolic variable has to be explicitly associated with one or more machine registers.

Control Structures and Statements

Statements in S* include the goto, nested if, while and repeat statements. Parameterless procedures are also allowed. In addition to these, parallel constructs in S* enable the user to specify explicitly operations which are to be executed in parallel. Two of these parallel constructs are:

1. cobegin
   S1;
   S2;
   coend
   specifies that statements S1 and S2 begin execution in the same phase of a microcycle.

2. cocycle
   S1;
   S2;
   coend
   specifies that statements S1 and S2 begin and end in the same microcycle.

A sample of some of the data objects of S*(V)

The following are some of the data objects that a user of S*(V), an instantiation of S* with respect to the Varian 75 can legally declare in a program. In the instantiation of S*(V), the set of constructs in S* is reduced to one that is directly supported by the hardware.

```
type word = tuple
               highbyte : seq [15..8] bit;
               lowbyte  : seq [ 7 ..0] bit
            end;

var mem : array [ 0..64K] of word with pc,mir,aluout;
var gpr : array [ 0..15] of seq  [15..0] bit;

/* the following are the operand register, program counter,
   instruction buffer register, instruction register, memory
   input register and I/O register, respectively  */

var opr,pc,ibr,ir,mir,ior : seq [15..0] bit ;

var aluina,aluinb,aluout : seq [15..0] bit; /* correspond to ALU*

var carry : bit ;                           /* carry-in to ALU */

/* the following is the processor status word */
var status : tuple
             key : seq [4..0] bit ;
             aluc : bit ;           /* carry flag */
             alusign : bit ;        /* sign flag */
             aluo : bit ;           /* all-ones flag */
             ovfl : bit ;           /* overflow */
             sc   : bit ;           /* shift counter */
             aluz : bit ;           /* all-zeros flag */
             supr : bit ;           /* supervisor mode */
             nu   : bit             /* unused bit */
        end;
```

Conclusion

Any instantiation of S* is a very machine-dependent low level language - the user of such a language still needs to have an intimate knowledge of the microarchitecture. Allocation of physical registers to symbolic variables of a program has to be managed by the programmer. S* does not offer the user much more ease of use but rather, it forces the programmer to make use of low level machine constructs in a more structured manner, thereby increasing program readability and reliability.

It is one of the few languages that allows the user to specify parallelism explicitly. In view of the level of the language, such a feature certainly ties in well with the rest of the language. As is the case with microassembly languages, the

efficiency of the microprogram is determined by the skill and effort of the programmer.

S* really defines a family of well-structured machine-oriented languages. Although programs written in these languages are certainly not portable, their common structure may well facilitate the transformation of programs from one language to another.

Though no S(M) has been implemented so far, the low level and machine-dependent primitives of the language suggests that it is feasible to construct such a compiler.

3.2 YALLL (Yet Another Low Level Language)[5]

D.A. Patterson, K. Lew, R. Tuck
University of California, 1979.

Design rationale

The complexity of generating efficient microcodes for an arbitrary microarchitecture from a machine-independent high level language is a consequence of the large gap that exists between these two levels. Rather than to try to solve this problem in one step, a machine-independent low-level language was implemented to study the feasibility of generating microcodes for different machines from the language.

Variables

There are no data types or data structuring facilities in YALLL. Variables are viewed as general purpose registers with the exception of MAR (memory address register) and MBR (memory buffer register). The user is required to bind the symbolic variables used to physical machine registers. This introduces machine dependency and decreases ease of use as the programmer has to keep track of the values residing in each register.

Operations

A set of arithmetic and logical operations are provided for data manipulations between registers. Instructions like LOAD and STORE can be used to transfer data between registers and main memory.

Control Structures

Structure of YALLL resembles that of conventional assembly language. It allows the user to specify a program in a completely sequential manner. The set of primitive control structures include conditional/unconditional branching, subroutine call and return, and indexed branching.

A sample program

```
        reg str = db              ; bind variables to physical
        reg tbl = sb              ;      registers
        reg char = mbr

loop:
        load   char , str         ; get addressed character
        jump   out  , if char = 0 ; quit if 0
        add    mar, char, tbl     ; add to table base address
        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
```

Conclusion

Owing to the moderate goals of the language, compilers of YALLL for two very different machines, HP300 and VAX 11/780, have been successfully implemented. Several sample programs were compiled for the two machines and it was observed that good codes were produced for the HP300 but codes for the VAX were not optimized due to the complexity of the VAX microarchitecture.

YALLL is built upon the common microarchitecture primitives of registers, busses and memory accesses. With machines which offer more sophisticated hardware features, it is doubtful that YALLL would be adequate. Though YALLL cannot cross significant

architectural boundaries, it is still transportable within families of computers. In transporting a program from 1 machine to another, the binding of symbolic variables to physical registers would have to be modified.

While the user will certainly write more YALLL instructions than microassembly instructions, the ability to specify a program in a purely sequential manner makes a YALLL program easier to read and write and improves its reliability.

3.3 EMPL[2]

D. DeWitt
University of Michigan, 1976.

Design rationale

Any high level language must provide the user with the capability of exploiting the unusual hardware features of each host machine if efficient microprograms are to be produced. With the diversity of available features on different machines, this has proved to be the major bottleneck in the design of machine-independent languages.

The core concept of extensible language design has been adopted in EMPL. A core language is designed which consists of a basic set of language primtives. The programmer can customize the language to fit the real hardware interface through use of extension statements and extension operators. Extension statements allow the user to define new data types and operations on the instances of the data types while extension operators define new operators.

Example of extension statement

If a machine supports a stack and operations on it, the extension statement can be used to manipulate this special resource by declaring a new data type and the legal operations on it. The compiler would then map references to this data type into references to the actual hardware stack. To preserve the portability of the source program, the user is required to model this resource and operations on it in the machine-independent core language. In the event that an operator or data type is

not hardware supported, e.g. when the program is executed on another machine, it is textually replaced by the statements in the body of the extension operators or statements.

The following is an example of the use of an extension statement to manipulate a hardware stack.

```
TYPE STACK
     DECLARE  STK(16) FIXED;
     DECLARE  STKPTR  FIXED;
     DECLARE  VALUE   FIXED;
     INITIALLY DO;
                 STKPTR = 0;

               END;

     PUSH : OPERATION ACCEPTS (VALUE)    /* PUSH operation */
            MICROOP   : PUSH 3 0 ;       /* translate into this
                                            microoperation */
            /* if machine does not support a stack , translate
                     into the following */

            IF STKPTR=16 THEN ERROR;
            ELSE DO;
                  STKPTR = STKPTR+1;
                  STK(STKPTR) = VALUE;
                 END;

            END;

     POP : OPERATION RETURNS (VALUE)    /* POP operation */
           MICROOP : POP 3 0 ;          /* translate into this
                                           microoperation*/
           /* if machine does not support a stack, translate into
                    the following */

           IF STKPTR = 0 THEN ERROR;
           ELSE DO;

                 VALUE = STK(STKPTR);
                 STKPTR = STKPTR-1;

                END;
           END;

ENDTYPE
```

The core language of EMPL

The core was designed to be simple and its available features appear to be rather limited.

Data Types and Data Structures

The only defined data type is integer and the only data-structuring facility is the vector or 1-dimensional array. To simplify the run-time environment, all variables are global.

Operators

The core contains a basic set of logical and arithmetic operators : +, -, *, / and logical shifts, and, or, exclusive-or and negation.

Control Structures

Procedures are allowed but these may neither have formal parameters nor local declarations. Statement forms include the assignment, procedure call and return, if-then-else, while-do and goto. Expressions are limited to only one operator.

Conclusions

Extensibility seems to solve, at least in part, the problems of machine-dependency and portability at the same time. It facilitates the use of target machine features and yet preserves the portability of the programs. A compiler for EMPL has not been fully implemented and it is difficult to guess at this stage, whether codes generated from such a compiler will meet efficiency standards.

The compiler textually replaces calls to user-defined data types and operators that are not hardware supported by their core statements equivalent. This could well result in a large increase in the size of the codes if these user-defined features are used frequently.

The main weakness of EMPL lies in the design of the core language. In trying to keep the language as simple as possible for easy implementation, the expressive power of the language has been sacrificed. Storage locations for symbolic variables are allocated automatically by the compiler. It is unclear how the main store may be referenced since there are no read or write statements and no distinction is made between variables residing in register or main memory.

4. Design of MPL/S130

The main objective of this project was to propose a high level language that would be suitable for microprogramming on the Eclipse S/130 (as well as on other microprogrammable computers).

EMPL appears to be the most promising high level language designed for microprogramming. The extensibility concept is an elegant solution to the conflicting goals of machine-independence and efficiency. The language is portable and yet capable of being translated into efficient microcode for different target processors.

The Eclipse S/130 is less sophisticated than most machines, however, and it seems unlikely that the user will ever need to extend the core language any further. Dissatisfaction with the structure of the core language of EMPL prompted the design of a new language MPL/S130 that could well be used in conjunction with the extensibility concept of EMPL.

4.1 Features of MPL/S130

In the following, the features of MPL/S130 are described together with the justification for certain design choices. Appendix A gives a formal definition of the syntax of the language in BNF.

Data Types and Data Structures

The basic data type is the word (16 bits) and the only data structuring facility is the array, which is restricted to one-dimensional as multiplication is often not hardware-supported.

The data facilities of MPL/S130 appear unduly restrictive but it ensures that the user is still working within the basic hardware constraints of a machine. Attempting to include more sophisticated data types (e.g. record type in Pascal) would result in an interface that is at such an abstract level from the underlying microarchitecture that the programmer can no longer be said to be microprogramming, in the true sense of the art.

Data Objects

Most user microprograms are called from the macrolevel but in existing languages, the importance of the micro-macro interface seems to have been ignored. MPL/S130 defines a clean interface between these two levels - variables can be declared as resident in main memory at locations relative to the program counter. References to these variables within the program would cause the compiler to generate the necessary instructions to access the main store. As there are no input/output statements in the language, data is transferred between an MPL/S130 program and the macro-level via the memory variables.

Variables not declared as memory variables will be allocated storage by the compiler. For efficiency consideration, the user may flag heavily used variables by suffixing the type with "$" to signify to the compiler that these variables should be allocated as efficiently as possible (e.g. assigned a physical register).

Constant identifiers may be defined to contain constant values in base 2 or 10.

Operators

In MPL/S130, a word is viewed as a 16-bit string and arithmetic and logical operations may be performed on identifiers of type word. Arithmetic operations are performed on a word as a two's complement integer while logical operations are performed bit-wise on a 16-bit string. Expressions can be of arbitrary length.

1. arithmetic operators

   \+ , -

   Multiplication and division operators have been left out of the language as these more complex operators are commonly not part of the set of processor primtives.

2. logical operators

   NOT, AND, OR, XOR

   SLL, SRL (linear shifts, left and right)

   SLC, SRC (circular shifts, left and right)

   = , >, >=, <, <=

   The left operand of a shift operation is shifted left or right by the number of bits specified by

the 4 least significant bits of the right operand. While relational operators are not S/130 machine primitives, they can easily be implemented with a few instructions.

Built-in Function

The built-in logical function, BIT facilitates the testing of individual bits of a word - returns true or false depending on whether the bit tested is 0 or 1 respectively.

BIT (<expression> <bit number>)

Control Structures

MPL/S130 provides a comprehensive set of control statements which includes

1. nested if statement
2. while statement
3. repeat statement
4. case statement with otherwise clause
5. for statement with default increment/decrement of 1
6. loop with exit statement

The case statement is useful for multi-way "branching". A default step of 1 in the for-statement generates more efficient codes while the loop-statement can be used to implement a more general for-statement.

The set of control structures is adequate and more importantly, can be implemented on most machines efficiently.

Functions and procedures of MPL/S130 are similar to the same features in Pascal except that the user is required to list the global variables that will be used within the function or procedure using the global statement. This facilitates the efficient allocation of storage to local variables of the module as the compiler can determine which global variable to deallocate first.

Structure of a sample MPL/S130 Program

```
PROGRAM MICRO ;
CONST
   I = #B 1010101101 ;                    - - bit string of I
   J = #D 24 ;                            - -J is constant decimal 10
VAR
   A : WORD$ ;                            - -A is a simple variable
   B : ARRAY [1..10] OF WORD ;            - -B is an array of 10 words
   C : WORD AT  PC+1 ;                    - -C is MEMORY [PC+1]
   D : ARRAY [0..5] OF WORD AT PC+2a;     - -address of D is at MEMORY
                                          - -                  [PC+2]
FUNCTION X (<parameter list>) : WORD ;
GLOBAL A ;                                - -global variable A will be
                                          - -    used in X
<local declarations>
BEGIN
     .
     .
     .
END ;
BEGIN                                     - -main program
     .
     .
     .
END.
```

4.2 Conclusions

MPL/S130 is Pascal-oriented. In the design of the language, feature selection was guided by consideration of:

1. constructs that are supported directly by the hardware of most computers
2. constructs that can be efficiently implemented on most microarchitectures.

In particular, the implications of certain language design choices for the code generation and optimization phases had to be considered. While more sophisticated constructs will no doubt increase the power of the language, the programs written may be inefficient not because unoptimized codes were generated but because, the user was not aware that such structures were not hardware-supported.

For this reason, tuples as described in the schema, S* were not included in the language as the Eclipse does not support selection of bit fields from a 16-bit word.

MPL/S130 was designed to be as machine-independent as possible so that programs written in the language would be portable. The main weakness of the language lies in the fixed size of type word which could hinder its use on machines with a different physical word size.

5. Microcode Compaction

The compilation process involves the translation of the source program into a sequence of microinstructions for execution on a target processor. This process is typically divided into two main phases:

1. syntactic and semantic analysis phase which outputs a sequence of microoperations as input for the following phase.
2. code generation phase which composes the sequence of microoperations into a sequence of microinstructions relative to the control word format of the target processor.

The first phase of the compilation has been well understood in relation to the macrocode compilers and should not pose any major problems. To satisfy the strict efficiency demands of microcodes, the code generation phase may not only involve composing microoperations into microinstructions but also of compacting the sequence of microoperations into a sequence of microinstructions that will execute in the minimum time possible.

The complexity of microcode compaction is determined by the following factors:

1. combinatorial complexity which is proportional to the number of microfields within a control word.
2. timing considerations.
3. inter-dependencies of the microoperations within a microinstruction.

Any of the above factors alone will complicate the process but their combined effect results in a problem which belongs to the class of NP-hard problems. Hence, any algorithm that produces optimal microcodes must be of at least exponential complexity.

This section of the report surveys some of the techniques that have been proposed for a block-oriented approach to microcode compaction.

The block-oriented approach divides a microprogram into basic blocks where each block is an oredered sequence of microoperations with no entry points, except at the beginning, and no branches, except possibly at the end. Analysis of an individual block is called local compaction while analysis of more than one block is called global compaction.

5.1 Local Microcode Compaction[4]

To compact the microoperations of an individual block, two distinctly separate analyses have to be carried out.

Data dependency analysis

This analysis serves to preserve the data integrity of the microoperations of a block. The relative order of two micro-operations, m_i and m_j (where m_i precedes m_j) in the original block has to be preserved if m_j is data dependent on m_i, i.e. if they satisfy any of the following conditions:

- an output resource of m_i is an input resource of m_j.
- an input resource of m_i is an output resource of m_j.
- an output resource of m_i is an output resource of m_j.

Basically, it implies that m_j must not execute in an earlier microinstruction than m_i. For a machine with a polyphase micro-instruction cycle, they could be placed in the same microinstruction if m_i finishes before m_j begins.

Conflict analysis

Restrictions imposed by the target machine itself could prevent two microoperations from being placed into the same microinstruction if the two microoperations require exclusive control over a hardware resource (e.g. register, ALU, etc.) at the same time.

Local compaction algorithms

Based on the two analysis techniques, several algorithms have been designed for local compaction:

- the linear or first-come-first-served algorithm which scans the microoperations of a block linearly in the order that they appear.
- the well-known branch-and-bound algorithm which generates an optimal solution through exhaustive search.
- the list scheduling algorithm - the compaction problem has been shown to be analogous to the processor scheduling problem and this adapted version is very similar to a heuristic version of the branch-and-bound algorithm.

Unlike the branch-and-bound algorithm which explores every legal combination, the others are nonoptimal algorithms which run in polynomial time.

5.2 Global Microcode Compaction[3]

After local compaction of the individual blocks is completed, the blocks may still contain unused microinstruction fields that can potentially hold additional microoperations if movement of microoperations across block boundaries is allowed. Global analysis results in better achievement of parallelism.

The flow of control between the basic blocks of a program is depicted by a directed graph called a flow graph. The blocks are represented as nodes and the directed edges denote the flow of control between the blocks.

Definitions

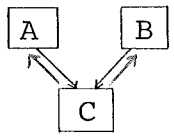
1. A microoperation is said to be free at the top of its block if it is data independent of all the other microoperations.
2. A microoperation is said to be free at the bottom of its block if no microoperation of the block is data dependent on it.

3. A data resource (input or output resource) is dead at the entrance to a block if the value stored in it will not be read in that block or some successor block without having been overwritten.

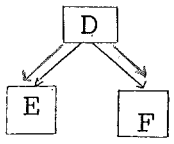
Rules governing the movement of microoperations across blocks

The following flow-graphs illustrate a set of criteria that have been proposed for controlling the transfer of micro-operations from one block to another such that the resulting blocks will still be semantically equivalent to the original blocks. The rules differ in effectiveness and are ranked in an approximate order of decreasing effectiveness. The arrows in red denote the direction of movement of microoperation m_I from block I to some other block.

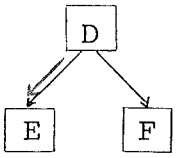
Rule 1: M_C is free at top of C.



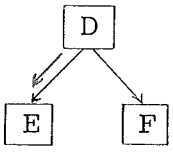
Rule 2: M_D is free at bottom of D.



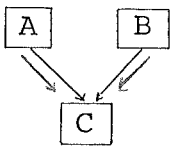
Rule 3: M_D is free at bottom of D and data resources written by M_D are dead in F.



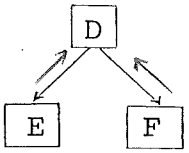
Rule 4: M_E is free at top of E and data resources written by M_E are dead in F.



Rule 5: identical copies of M_A and M_B are free at the bottoms of A and B respectively.



Rule 6: identical copies of M_E and M_F are free at the tops of E and F respectively.



Unlike local compaction, global compaction has not been as widely implemented. Each microoperation transfer across blocks requires recompacting the blocks involved, a time-consuming process.

6. Conclusions

The advantages of microprogramming remain largely unexploited because of the inadequacy of the tools and interfaces provided for the microprogrammer. One solution to the problem would be to improve the current environment of microprogramming through higher level language support. The purpose of this project has been to consider the feasibility and effectiveness of such an approach and to reveal the problems associated with the design of a high level language for microprogramming.

While it has not been conclusively proven that automatic microcode optimization can compare favourably with hand optimization, the theoretical complexity of microcode compaction suggests that for larger programs, compiler generated codes could prove to be far superior to hand-written codes. As the problem grows, it becomes more difficult for the programmer to keep track of the entire program while a compiler equipped with suitable compaction algorithms could still guarantee optimized codes.

In the last decade, active research into high level microprogramming has brought forth a vast amount of new ideas and knowledge. With interest in this firmware domain steadily increasing, it is hoped that manufacturers will begin to provide better hardware support for user microprogramming so that their machines will be truly microprogrammable.

7. Appendix A

The following is a formal definition of MPL/S130 in BNF. The syntax and semantics of the language is oriented towards that of Pascal.

7.1 Program block

```
<program> ::=  PROGRAM <identifier>; <declaration part><body>.

<body> ::=  <compound statement>
```

7.2 Statements

```
<compound statement> ::= BEGIN <statement list> END

<statement list> ::=  <statement> | <statement> ; <statement list>

<statement>  ::=  <assignment statement>|
                  <if statement>|
                  <while statement>|
                  <repeat statement>|
                  <case statement>|
                  <for statement>|
                  <loop statement>|
                  <compound statement>|
                  <procedure statement>|
                  <exit statement>|
                  <empty statement>|

<assignment statement> ::= <variable> := <expression>|
                           <function identifier> := <expression>

<if statement> ::= IF <expression> THEN <statement>  ENDIF |
                   IF <expression> THEN <statement>  ELSE
                   <statement>  ENDIF

<while statement> ::= WHILE <expression> DO <statement list>
                      ENDWHILE

<repeat statement> ::= REPEAT <statement list> UNTIL <expression>
```

```
<case statement> ::= CASE <expression> OF <case body> ENDCASE

<case body>  ::= <constant> : <statement>|
                 <constant> : <statement> ; <case body

<for statement> ::= FOR <variable> := <expression> TO <expression
                        DO <statement list>  ENDFOR |

                    FOR <variable> := <expression> DOWNTO
                    <expression>
                        DO <statement list>  ENDFOR

<loop statement> ::= LOOP : <statement list>  EXIT

<procedure statement> ::= <procedure identifier><actual
                          parameter part>

<actual parameter part> ::= <empty part>|
                            (<actual parameter list>)

<empty part> ::=

<actual parameter list>  ::=  <actual parameter>|
                              <actual parameter>,
                              <actual parameter list>

<actual parameter> ::= <expression>|
                       <variable>

<exit statement> ::=  EXIT

<empty statement> ::=
```

7.3 Declarations

```
<declaration part> ::= <constant definition part>
                       <variable definition part>
                       <module declaration part>

<constant definition part> ::=  <empty statement>|
                                CONST <constant definition list>;

<constant definition list> ::=  <constant definition>|
                                <constant definition>;
                                <constant definition list>
```

```
<constant definition> ::= <identifier> = <base> <constant>

<base> ::=  #B | #D

<variable declaration part> ::= <empty statement>|
                                VAR <variable declaration list>;

<variable declaration list> ::= <variable declaration>|
                                <variable declaration>;
                                <variable declaration list>

<variable declaration> ::= <identifier list> : <type>|
                           <identifier list> : <type>
                           <memory location>

<identifier list> ::= <identifier>|
                      <identifier> ; <identifier list>

<type> ::= WORD|
           WORD$|
           <array type>

<array type> ::=  ARRAY [<lower bound> .. <upper bound>] OF WORD

<memory location> ::= AT PC + <constant>|
                      AT PC + <constant> @

<module declaration part> ::= <empty statement>|
                              <function declaration>
                              <module declaration part>|
                              <procedure declaration>
                              <module declaration part>

<function declaration> ::= FUNCTION <function identifier>
                           <formal parameter part> : WORD ;
                           <declaration part> <global statement>
                           <body> ; |
                           FUNCTION <function identifier>
                           <formal parameter list> FORWARD ;
```

```
<formal parameter part> ::= <empty part>|
                            (  <formal parameter list>)

<formal parameter list> ::= <formal parameter>|
                            <formal parameter> ;
                            <formal parameter list>

<formal parameter> ::=  VAR <identifier list> : <type>|
                        <identifier> : <type>

<global statement> ::= GLOBAL <identifier list> ;

<procedure declaration> ::= PROCEDURE <procedure identifier>
                            <formal parameter part> ;
                            <declaration part><global statement>
                            <body> ; |
                            PROCEDURE <procedure identifier>
                            <formal parameter part>  FORWARD;
```

7.4 Expressions

```
<expression> ::= <simple expression>|
                 <simple expression><logical operator>
                 <simple expression>

<simple expression> ::=  <sign><term>|
                         <sign><term><term list>

<sign> ::= + | - | <empty part>

<term list> ::=  <empty part>|
                 <addop><term>

<addop> ::= + | - | OR | XOR

<term> ::= <factor>|
           <factor> AND <term>
```

```
<factor> ::=  <constant>|
              NOT <factor>|
              ( <expression>)|
              <variable>|
              <constant identifier>|
              <function call>

<variable> ::= <variable identifier>|
               <variable identifier>[ <simple expression>]

<function call> ::= <function identifier><actual parameter part>

<logical operator> ..=  = | < | <=| > | >= |
                        SLL | SRL | SRL | SRC
```

7.5 Miscellaneous

The consecutive minus signs ("--") denote that the rest of the program line is a comment.

Only integer constants are allowed. Identifier names may contain digits and/or letters but must begin with a letter.

The built-in function BIT has the following syntax:

```
BIT ( <expression>,<bit number> )
```

where <bit number> is in the range 0 - 15.

Memory variables are declared to be resident in main memory at locations relative to the program counter (PC). "@" denotes that this location contains the variable address (i.e. one level of indirection).

8. References

The literature contains a rich collection of articles which were found to be of relevance to this project. The following is a list of those references which particularly influenced this final report.

1. S. Dasgupta
Some Aspects of High-Level Microprogramming
- ACM Computing Surveys Volume 12, Sept. 80.

2. D.J. De Witt
Extensibility - a New Approach for Designing Machine-independent Microprogramming Languages
- Sigmicro Volume 7, Sept. 76.

3. F.A. Fisher, D. Landskov and B.D. Shriver
Microcode compaction : Looking backward and looking forward
- AFIPS Volume 50, 1980.

4. D. Landskov, S. Davidson, B. Shriver and P.W. Mallett
Local Microcode Compaction
- ACM Computing Surveys Volume 12, Sept. 80.

5. D.A. Patterson, K. Lew and R. Tuck
Towards an efficient machine-independent language for microprogramming
- Sigmicro Volume 10, Dec. 79.

6. M. Sint
A survey of high level microprogramming languages
- Sigmicro Volume 11, Nov. 30 - Dec. 3, 80.

*