COSC 460 Honours Research Project Report

Department of Computer Science

University of Canterbury

Christchurch

New Zealand

MPLS130

A High Level Language For Microprogramming The Eclipse S/130

Michael Timothy Kerrisk

October 13, 1982

Supervisor : Dr R E M Cooper

Abstract

An overview of microprogramming and of the methods and tools available for microprogramming is given. A review is given of the language MPLS130, a high level language for microprogramming the Data General Eclipse S/130. Next, a progress report on an implementation effort for this language is given. Finally a summary is made of further work to be done and some concluding remarks are made about the language.

Table of Contents

1 Introduction

1.1 Microprogramming

The concept of microprogramming was first outlined over three decades ago by Wilkes as a regular and systematic method for the design of control units of digital computers [Wilk51]. Since its inception, microprogramming has evolved to the point where it is now the standard method for implementing the instruction sets of modern digital computers.

Microprogramming enabled manufacturers to efficiently implement flexible machine architectures and with the advent of writable control stores the microprogramming facility of machines was made available to users.

By microprogramming small heavily used sections of code, users can speed execution of the code by a factor of five to ten, thereby greatly increasing the overall speed of their programs. As well microprogramming can be used to extend the instruction set of a machine or even to emulate an entirely different machine.

1.2 High Level Languages For Microprogramming

Despite its wide use by manufacturers and users, the tools available for microprogramming are still quite primitive. The standard tool available is a machine dependent microassembler, and (especially for horizontally microprogrammable machines) writing efficient microassembler programs is significantly more difficult than programming in conventional assembler. The situation is further complicated by the need for the microprogrammer to handle machine interrupts and page faults.

For the user the situation is made even more difficult by the limited degree of manufacturer support for user microprogramming. The support made available by manufacturers ranges from nothing at all to at best a microassembler, loader and a few manuals.

Because of the complexity of microprogramming and the primitive tools available, microprograms are usually hard to understand and therefore unreliable and difficult to maintain. The developement of a high level microprogramming language (HLML) would solve or at least reduce the magnitude of the problems facing the microprogrammer. A HLML would relieve the user of the problems of handling machine interrupts, and programs would be more readable and therefore more reliable and easier to maintain.

High level languages for microprogramming have been a topic of discussion for over a decade. Attempts to implement compilers for HLMLs have had varying degrees of success, however there are as yet no widely available compilers [Sint80].

1.3 Background And Aim Of This Project

In 1981, as part of a research project, a high level language was designed for microprogramming the Data General Eclipse S/130 by Soon Sow Lai, at the University of Canterbury [Soon81].

The language, named MPLS130, was loosely based on Pascal. MPLS130 programs have a similar layout to Pascal programs, with similar statement forms. The major differences are in the data structuring facilities which are much more restricted.

The main aims of this project were to review the design of MPLS130 and to implement it on the Eclipse S/130.

2 Review Of MPLS130

2.1 Alterations Made To MPLS130

As part of this project, certain alterations have been made to the design of MPLS130. These alterations are detailed in Appendix IV.

2.2 Goals In The Design Of A HLML

As a HLML, MPLS130 has to fulfil the following design goals as closely as possible :

2.2.1 Efficiency Of Generated Code

The only reason for writing microprograms is to increase execution speed. Therefore the language (and compiler) must be designed in a manner which allows the production of efficient microcode. If this cannot be done then the whole advantage of writing microprograms is lost.

2.2.2 Structured Microprogramming

A HLML should facilitate the writing of microprograms in a structured, readable manner. To this end programs should be written in a sequential order and contain control constructs which are efficient and easily understood. Structured HLMLs allow the production of programs which are more readily understood and therefore more reliable and easier to maintain.

2.2.3 Machine Independence

As far as possible a HLML should be machine independent. In particular, a HLML should relieve the user of the need to know specific details of the microarchitecture of any machine and of the problem of handling machine interrupts. Thus a machine independent HLML would enable the production of microprograms which would be portable from one machine to another.

Design of a machine independent HLML faces great difficulties of which the most important is that the microarchitecture of most machines is tailored for the implementation of the standard macroinstruction set of the machine. As a result, microarchitectures vary considerably from one machine to another. This makes the production of efficient microcode from the same HLML for two different machine very difficult.

2.3 An Overview Of MPLS130

As has been stated earlier, MPLS130 is a Pascal based language. The most notable differences are the lack of type declarations and the more restricted data structures. A full description of the language is given in Appendices I and II.

2.3.1 Numbers And Constants

Numbers may be denoted in binary, octal, decimal or hexadecimal. Constant declarations are made in the same way as in Pascal. A notable extension over Pascal is that constant identifiers may be equated to expressions which involve numbers and previously defined constants.

2.3.2 Variables

Variables may be declared in macromemory or local to the microprogram. (On the Eclipse, local variables are allocated in the scratchpad) The basic data type is the sixteen bit word and the only data structuring facility is the single dimension array.

MPLS130 does not allow explicit binding of registers to variables. Instead the programmer may declare heavily used variable using the reserved word word$ instead of word, in which case the compiler will allocate a register for the variable if one is available. The only register on which operations may be directly performed is the program counter.

Macromemory can be accessed as an array called MEM, with a lower bound of zero and an upper bound defined by the size of the macromemory.

2.3.3 Operators And Expressions

A wide variety of operators is available in MPLS130 including addition, subtraction, logical operators (not, and, or, exclusive or), linear and circular shifts and comparison operators. Notably absent are division and multiplication as they are not implemented in hardware in many machines (including the Eclipse S/130).

Expressions may be formed in the same way as in Pascal.

2.3.4 Assignment Statements

Assignment statements similar to those of Pascal are allowed, with any number of operators in the expression on the right hand side of the assignment operator.

2.3.5 Control Statements

The major difference between the control statements of MPLS130 and Pascal is the use of a bracketing reserved word to indicate the end of the statement. This means that a control statement may contain several statements as opposed to Pascal where the use of the reserved words begin and end is necessary. Within MPLS130, begin and end are used only to delimit the statement blocks of programs, procedures and functions.

Two conditional control statements are available in MPLS130, the if statement and the case statement, with semantics similar to those of their Pascal counterparts.

For repetition there are the for, while, repeat, and loop statements. The loop statement executes its statement list continuously.

There is an exit statement available which will conditionally exit to the foot of the repetition statement in which it is located.

2.3.6 Procedures And Functions

Procedures and functions are declared and used in much the same way as in Pascal, but more control is allowed over the method of parameter passing.

MPLS130 is not a stack oriented language which has some important consequences. All variable allocation is done at compile time and as a result recursion is not allowed.

The scope rules of MPLS130 are similar to those of Pascal except in the case of variable identifiers. Within an inner scope the only variables from the next outer scope that are visible are those specified in a global declaration within the inner scope. Information about what outer variables are visible in an inner scope is used by the compiler in allocation and deallocation of registers.

Example of a procedure in MPLS130

```
   procedure multiply(in a, b : word$; out  c : word$);

      -- multiply the unsigned integers in a and b to give the
      --    product in c

      begin
         c := a;
          loop
            a := a sll 1;
            b := b srl 1;
            if set(b, 1) then
               c := c + a;
            endif;
            exit when b = 0;
         endloop;
      end;
```

3 Implementation Of MPLS130

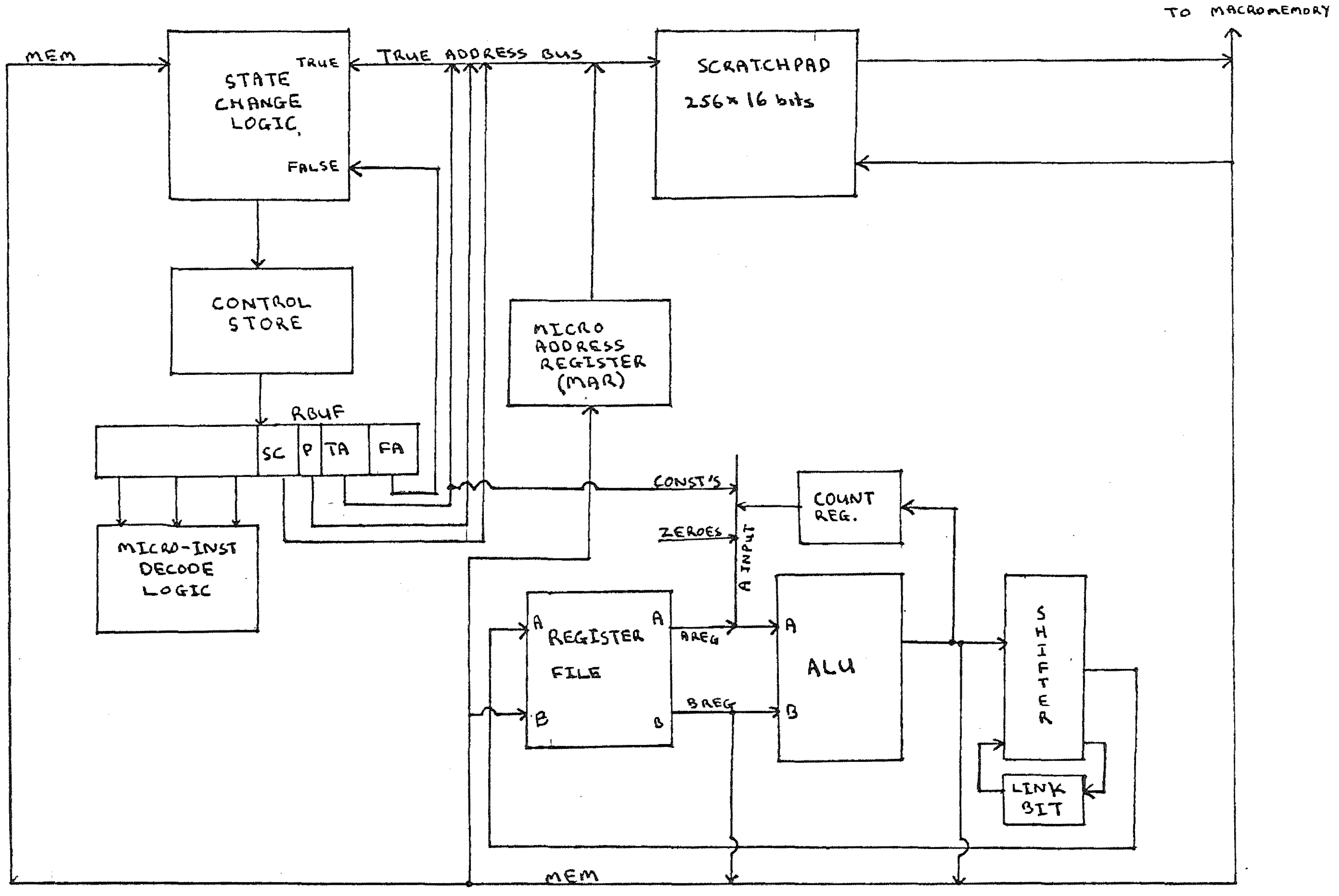
3.1 Parser Design

As MPLS130 is based on Pascal and defined in the same manner (Backus-Naur form), the technique of recursive descent was chosen as the most appropriate method of parsing. A parser has been written in Pascal using the recursive descent method with follow sets to aid error recovery, as described by Wirth [Wirt76]. Notes on some of the details of the design of the parser are given in Appendix V.

3.2 Microprogramming The DG Eclipse S/130

The Eclipse S/130 is a 16 bit horizontally microprogrammable machine with 1024 words of writable control store [DG77]. For the purpose of microprogramming, there are 256 * 16 bit words of high speed memory (called the scratchpad) available only at the microlevel. As well there are four registers available at the microlevel which are not available at the macrolevel. One of these registers is the macroprogram counter (PC). A simplified diagram of the Eclipse data paths is given on the following page.

The control words of the WCS are 56 bits wide and are divided into 15 fields allowing a large amount of parallelism. Each microinstruction contains two fields (the true and false address fields) from which the next microinstruction to be fetched is selected. The execution of microinstructions on the Eclipse is polyphase; in particular, the selection of the true or false address for the next microinstruction is made by tests on the ALU output after it has been generated. Thus the next microinstruction is being fetched while the shift and load phases of the present instruction are being executed.



SIMPLIFIED BLOCK DIAGRAM OF THE ECLIPSE S/130

3.3 Problems Faced In Code Generation

3.3.1 Efficiency

The efficiency requirements for microcode produced by a compiler for a HLML are much more stringent than those facing compilers which produce macrocode. This is both because the efficiency of the whole computer system is defined by the efficiency of the microcode and because the main aim of writing microcode is to speed execution as much as possible.

3.3.2 Parallelism And Compaction

The structure of horizontal microcode is much more complicated than conventional macrocode. Each microinstruction is composed of several microoperations which operate in parallel to control the hardware of the machine. Thus, in the production of microcode, one of the major problems is packing as many microoperations as possible into each microword.

There are two difficulties to be faced in the packing of microoperations into microwords. The first of these is that the microoperations in a microword are not independent of each other. Some microoperations may not be coded together, while the input to some microoperations is taken from the output of other operations. For instance on the Eclipse the output of the ALU is used as the input to the shifter. The second problem is that some microoperations take several microcycles to complete, during which time the resources they use are not available to be used in other microinstructions. As an example, on the Eclipse S/130, the process of writing to macromemory takes three microcycles.

3.3.3 Tailored Microarchitectures And Machine Independence

The Eclipse S/130, like many microprogrammable machines, is not microprogrammable in a general sense. Its microarchitecture is strongly tailored towards the efficient implementation of the standard macroinstruction set. Thus, since MPLS130 is designed to be machine independent, there are details of machine microarchitectures of which it will be very difficult to take advantage. This lack of generality of microarchitectures makes it extremely difficult to produce the best possible microcode from a machine independent HLML.

3.4 Overview Of Code Generation

In order to compile MPLS130 to Eclipse S/130 microcode, the following actions need to be taken :

1. Variables must be allocated to registers, scratchpad and macromemory.
2. The constructs and operations of MPLS130 must be broken down into simple steps.
3. The simple steps have to be converted to microoperations.
4. Microoperations must be compacted into microwords.

To do this all in a single step is extremely difficult, so code production is broken into two separate steps.

3.4.1 Step 1 : Production Of Intermediate Code

The first step of the compilation is that of parsing the input MPLS130 program. From this input, the parser produces an intermediate language for use in step 2.

The parser allocates variables to registers, scratchpad and memory, and uses this information in generating the intermediate language. As well the parser converts the control statements of the language into constructs using labels and jumps.

Evaluation of expressions is broken down into simple two operand operations, memory and scratchpad references.

3.4.2 Step 2 : Production Of Microcode

The second step of the compilation takes the intermediate language generated in step 1 and produce equivalent microassembler source code. This step involves three parallel processes; the coder, the packer and the emitter.

The coder converts the intermediate language into microoperations as follows. For each kind of instruction in the intermediate language, there is one or more templates which indicate the way the instruction is to be converted to microcode. Each template is a microinstruction in which only certain microoperations are used. The template also contains information about what resources are used, and what fields are not used in the microinstruction.

As the coder converts intermediate instructions into templates, the templates are passed to the packer. The packer combines templates into single microinstructions where possible.

This is done by keeping a buffer of the most recently generated microinstructions and trying to move the newly received templates into one of the microinstructions in the buffer. To do this the packer may rearrange the order of the microinstructions already in the buffer (within certain constraints mentioned later). If the template cannot be moved into a microinstruction already in the buffer, a new microinstruction is started in the buffer.

The buffer is variable in length, holding all the microinstructions since the last labelled microinstruction. When a labelled microinstruction is added to the buffer, all the microinstructions currently in the buffer are cleared out and passed to the emitter for output. This is done because of the constraint on moving microinstructions past block boundaries, described below.

The emitter takes the microinstructions, fills in any unspecified fields with default values, and outputs the microinstruction.

The general rules which govern under what conditions two microinstructions (or templates) may be moved past each other in the buffer are as follows :

a) A microinstruction may not be moved past the start of a block. Effectively this means that that a microinstruction may not be moved past a labelled microinstruction.

b) The relative order of two microinstructions M and N (where N follows M) must be maintained if N is data dependent on M. N is data dependent on M if
   i) an output resource of M is an input resouce of N, or
   ii) an input resource of M is an output resource of N, or
   iii) an output resource of M is also an output resource of N.

As well, a template T may combined with an earlier microinstruction M if the input of T is generated by an earlier phase of M. In particular, for the Eclipse S/130, this applies where the ouput phase of M is an ALU result which is used as an input for T for a shifter or state change test microoperation.

Where possible the packer moves instructions in between microinstructions which take more than one microcycle, as is the case for starting and then reading from or writing to macromemory on the Eclipse S/130.

The packer also combines a template which contains an unconditional jump with the previous microinstruction.

An example is given in Appendix VI showing how this method can be applied to generate code for MPLS130.

4 Summary

The language MPLS130, a high level language for microprogramming, has been reviewed and improved. A reference manual has been produced for the language.

The basis has been laid for an implementation of this language om the DG Eclipse S/130. A parser has been written for the language in standard Pascal. A two phase method to produce microcode via an intermediate language has been designed. The method of generation of the intermediate language and its nature have been described. A method for conversion of this intermediate code into (locally optimised) microcode has been given.

5 Further Work

There is a large amount of further work to be done in the area defined by this project. In particular, the intermediate language needs to be fully defined, and the parser extended to produce it. As well, the second phase of the compiler, to convert the intermediate language into microcode, is yet to be constructed.

Another area to be explored is that of global optimisation. If a global optimiser is produced, it could be used as a further step in the compilation phase, to optimise the code produced by the second phase. A global optimiser would have the added advantage that it could also be used to optimise existing micro-assembler programs.

6 Conclusions

MPLS130 is a relatively machine independent language for microprogramming. Because of this, it will be extremely difficult for any compiler to produce code as efficient as that of an expert human microprogrammer. However the compiler should still be able to produce good code, and this coupled with the advantages of programming in a high level language (speed of program production, readability, reliability and portability) outweigh this disadvantage.

Appendix I : Reference Manual For MPLS130

1 Basic symbols

1.1 Character Set

The character set of MPLS130 is composed of

Upper case letters	A-Z
Lower case letters	a-z
Digits	0-9
Special characters	+ - < > = ( ) [ ] $. , : ; # _

Note that upper and lower case letters are treated equivalently in all respects

1.2 Reserved words

array	at	begin	case	const	do
downto	else	end	endcase	endfor	endif
endloop	endwhile	exit	for	forward	function
global	if	in	inout	loop	memory
not	of	out	procedure	program	pc
repeat	then	to	until	var	when
while	word	word$			

1.3 Special symbols

```
+    -    <    >    <=   >=   <>   :=   :    =
[    ]    (    )    ;    --   ..   ,    .    _
```

2 Identifiers

An identifier is composed of a sequence of letters, digits and the $ and _ characters starting with a letter or the $ character. Only the first ten characters of an identifier are significant.

```
<identifier> ::= <letter> { <letter> | <digit> | _ }

<letter> ::= a | b | c | d | e | f | g | h | i | j |
             k | l | m | n | o | p | q | r | s | t |
             u | v | w | x | y | z
             A | B | C | D | E | F | G | H | I | J |
             K | L | M | N | O | P | Q | R | S | T |
             U | V | W | X | Y | Z | $

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Examples of valid identifiers

```
abc     AnIdentifierLongerThan10Characters
$$      a$sign     product
```

3 Numbers

Numbers may be specified in decimal, binary, octal or hexadecimal. The default base is decimal. The base of the number is given by a base specifier at the start of the number of the form #<base> where base is one of the letters, 'B' for binary, 'O' for octal, 'D' for decimal or 'X' for hexadecimal. Numbers are constrained to the range representable by the 16 bit word size of the Eclipse S/130.

```
    <unsigned number> ::= <digit> { <digit> } |
                          #B <bit> { <bit> } |
                          #O <octal digit> { <octal digit> } |
                          #D <digit> { <digit> } |
                          #X <hex digit> { <hex digit> }

    <bit> ::= 0 | 1

    <octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

    <hex digit> ::= <digit> | A | B | C | D | E | F |
                    a | b | c | d | e | f
```

Examples of valid numbers :

```
#Xaaaa        #d1345        1345          #B0101111100000110
#xFF12        #o0007        #O377         #D255
```

4 Comments

Any text after the symbol -- to the end of the line is treated as a comment by the compiler.

Example

```
    a := b;      -- This text is ignored by the compiler
```

5 Constants And variables

5.1 Constant Definitions

A constant definition part is introduced by the reserved word _const_ which may be followed one or more constant definitions separated by semicolons.

Constant definitions allow an identifier to be equated to a constant expression. A constant expression is an expression in which the operands are numbers or previously defined constants. Any of the operators available in MPLS130 may be used in constant expressions.

```
<constant definition part> ::= const
                               <constant definition list> ; |
                               <empty>

<constant definition list> ::= <constant definition>
                                  { ; <constant definition> }

<constant definition> ::= <constant identifier> =
                             <constant expression>

<constant identifier> ::= <identifier>

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

<simple constant expression> ::= <sign> <constant term>
                                 { <addop> <constant term> }

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

<empty> ::=

<constant term> ::= <constant factor>
                       { and <constant factor> }

<constant factor> ::= <constant identifier> |
                      <unsigned number> |
                      not <constant factor> |
                      ( <constant expression> )

<addop> ::= + | - | or | xor

<logical operator> ::= = | <> | < | <= | > | >= |
                       sll | srl | src | slc
```

5.1.1 Predefined Constants

There are two predefined constants; true with the MPLS130 true value (-1) and false with the MPLS130 false value (0).

Example

```
const
   a = #Xaa;
   b = #B00110111;
   c = a - b;
   d = b + (a xor c);
```

5.2 Variables

A variable declaration part is introduced by the reserved word var, followed by one or more variable declarations separated by commas.

```
<variable declaration part> ::= var
                                <variable declaration list> ;
                                | <empty>

<variable declaration list> ::= <variable declaration>
                                { ; <variable declaration> }

<variable declaration> ::= <internal variable declaration> |
                              <external variable declaration>

<internal variable declaration> ::= <identifier list> :
                                       <type clause>

<external variable declaration> ::= <identifier> :
                                       <type clause>
                                       <memory location>

<identifier list> ::= <identifier> { , <identifier> }

<type clause> ::= word | word$ | <array type>

<array type> ::= array [ <lower bound> .. <upper bound> ]
                    of word

<lower bound> ::= <constant expression>

<upper bound> ::= <constant expression>

<memory location> ::= at pc <offset> |
                         at memory <constant expression>

<offset> := <constant expression> | <empty>
```

The basic variable type in MPLS130 is the sixteen bit word. The only structured type is the single dimension array. More complex data structuring is not allowed because of the inefficiency that would be incurred in implementation. Variables may be declared as being resident in main (macro) memory or local to the microprogram (in the scratchpad or registers).

5.2.1 Register Allocation

A variable location is defined by the use of the reserved word word. In the case of macromemory variables this means the variable is in main memory, while for local variables a word is allocated in the scratchpad. In order to speed execution of a microprogram a heavily used variable may be allocated to a register by declaring it using the reserved word word$. This will cause the compiler to allocate a register to this variable if there is one available.

Notes

1. The scratchpad of the Eclipse contains 256 sixteen bit words. Of these, some are used by the standard firmware for the Eclipse instruction set, leaving slightly more than 200 for general use. This places an upper limit on the amount of local storage available to a program.

2. On the Eclipse, a total of 8 registers are available, one of them being the program counter (pc).

3. If a macromemory variable is declared with word$ then on entry to the microprogram the variable will be copied into a register and then copied back out when the microprogram is exited.

5.2.2 Array Types

Variables may be declared as one dimensional arrays with an upper and lower bound.

Example

```
a : array [-5..15] of word;
```

Notes

1. An array of word$ is not allowed.

2. Arrays whose lower bound is zero will be accessed more efficiently as there is no need to subtract an offset from the index.

5.2.3 Macromemory And Local Variables

By default a variable is declared local unless a clause is given specifying it to be located in macromemory.

A local variable declaration consists of a list of one or more identifiers (separated by commas) followed by a type clause.

Examples

```
a, b, c : word;        -- Allocated in the scratchpad
g, h : word$;          -- Allocated to registers if possible
al, bl : array [0..7] of word;
                          -- Arrays allocated in scratchpad
```

Macromemory variables are distinguished by a memory location clause starting with the reserved word at which specifies the variable as being at an absolute address in memory or at a location relative to the pc.

A macromemory variable declaration consists of (only) one identifier followed by a type clause and a memory location.

Examples

```
arglist : array [0..10] of word at pc + 3;
                                -- A pc relative array
status : word$ at memory 45;
                                -- status is at absolute
                                -- location 45
```

6 Operators And Expressions

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

<simple expression> ::= <sign> <term> { <addop> <term> }

<term> ::= <factor> { and <factor> }

<factor> ::= not <factor> | ( <expression> ) |
             <constant identifier> | <variable> |
             <function call> | <unsigned number>

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

<variable identifier> ::= <identifier>

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

In MPL$130, arithmetic, logical and comparison operators are available. Arithmetic and comparison operators are performed on a word as a twos complement integer, while logical operations are performed on a word as a sixteen bit string.

Arithmetic operators

-	unary subtraction
+	addition
-	subtraction

Logical operators

not	logical complement
and	logical and
or	inclusive or
xor	exclusive or
sll	left shift
srl	right shift
slc	circular left shift
src	circular right shift

Comparison operators

=	equality
<>	non equality
<	less than
<=	less than or equal
>	greater than
>=	greater than or equal

The comparison operators yield either the truth value (all ones = -1) or the false value (all zeroes = 0).

In expressions the priority of operators is :

Highest	not - (unary)
	and
	+ - xor or
Lowest	sll srl slc src = <> < <= > >=

The priority of operators may be overridden by the use of parentheses.

Examples of expressions

```
a
23 + a xor #XFF
a sll (b - c)
(x <> y) or z
```

7 Statements

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

7.1 Simple Statements

A simple statement is one of which no part constitutes another statement.

7.1.1 Assignment Statements

An assignment statement replaces the current value of a variable or function result with a new value given by an expression.

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

Examples

```
a := 3 + c
b[0] := z sll 3
```

7.1.2 Procedure Statements

A procedure statement invokes the procedure given by a procedure identifier. A procedure statement may contain a list of actual parameters in parentheses which are substituted for the formal parameters in the procedure invocation.

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

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

<actual parameter list> ::= <actual parameter>
                               { , <actual parameter> }

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

The actual parameter must be a variable if the corresponding formal parameter has been declared out or inout.

Example

```
multiply(x, y, z)
```

7.1.3 Exit Statements

```
<exit statement> ::= exit when <expression>
```

An exit statement causes the program to exit to the foot of the innermost repeat, while or loop statement in which it is embedded if its expression evaluates to nonzero result.

7.2 Structured Statements

Structured statements are constructs containing statement lists which are to be executed conditionally or repeatedly. A statement list is a list of one or more statements separated by semicolons.

```
<statement list> ::= <statement> { ; <statement> }
```

7.2.1 If Statements

```
<if statement> ::= if <expression> then <statement list>
                      <else clause> endif

<else clause> ::= else <statement list> | <empty>
```

The if statement specifies that the statement list following the reserved word then be executed only if the expression evaluates to a non-zero result. If the expression evaluates to zero, then if there is an else part, the statement list following the else is executed otherwise no statement is executed.

Example

```
if A + B then
   x := 0;
   y := y + 1;
else
   x := 1;
endif
```

7.2.2 Case Statements

The case statement contains an expression (the selector) and a list of statement lists each introduced by the reserved word when followed by a list of constant expressions (separated by commas and terminated by a colon).

```
<case body> ::= when <constant list> : <statement list>
              { when <constant list> : <statement list> }

<constant list> ::= <constant expression>
                  { , <constant expression> }
```

The selector is evaluated and if there is a statement list with a label equal to the value of the selector then it is executed. The effect of the case statement is undefined if the selector is not equal to any label.

Example

```
case opcode of
   when plus : A := A + W;
   when load : A := W;
endcase
```

7.3 Repetitive Statements

7.3.1 Loop Statements

```
<loop statement> ::= loop <statement list> endloop
```

A loop statement executes its statement list continuously. The only way to leave a loop statement is by the use of the exit statement.

7.3.2 While Statements

```
<while statement> ::= while <expression> do <statement list>
                         endwhile
```

A while statement repeatedly evaluates the controlling expression and then performs the statements in the statement list if the expression value is non-zero. The while statement is exited as soon as the controlling expression evaluates to zero.

The while statement

```
while B do S endwhile
```

is equivalent to

```
loop
   exit when B = 0;
   S
endloop
```

7.3.3 Repeat Statements

```
<repeat statement> ::= repeat <statement list> until
                           <expression>
```

The repeat statement repeatedly executes its statement list and evaluates the controlling expression until it gives a non-zero value.

The repeat statement

```
repeat
   S
until B
```

is equivalent to

```
loop
   S;
   exit when B
endloop
```

7.3.4 For Statements

```
<for statement> ::= for <variable> := <initial value>
                       <direction> <final value>  do
                       <statement list> endfor

<direction> ::= to | downto

<initial value> ::= <expression>

<final value> ::= <expression>
```

The for statement repeatedly executes a statement list while progression (in steps of 1) of values is assigned to a variable called the control variable.

On entry to the for statement, the initial and final value expressions are evaluated and the initial value is assigned to the control variable. If the step direction is to (downto) then if the initial value is greater (less) than the final value the statement list is never executed. Otherwise the statement list is repeatedly executed and the control variable incremented (decremented) by one for each repetition. When the control variable becomes greater (less) than the final value the for statement is terminated.

The for statement

```
   for v := I to F do S
```

is equivalent to

```
   v := I;
   loop
      exit when v > F;
      S;
      v := v + 1;
   endloop
```

8 Procedure Declarations

A procedure declaration associates an identifier with a block of declarations and statements which can be activated by a procedure statement.

```
<procedure declaration> ::= procedure <procedure identifier>
                               <formal parameter part> ;
                               <subprogram block> ;

<procedure identifier> ::= <identifier>

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

<formal parameter list> ::= <formal parameter>
                              { ; <formal parameter> }

<formal parameter> ::= <passing specification>
                         <local variable declaration>

<passing specification> ::= in | inout | out

<subprogram block> ::= <global declaration> <block> | forward

<global declaration> ::= global <identifier list> ; | <empty>

<body> ::= begin <statement list> end
```

8.1 Procedure Headings

The procedure heading specifies an identifier (which names the procedure) followed by a list of formal parameters (if there are any), in brackets.

Parameters may only be scalars, i.e. a whole array may not be passed as a parameter.

The method by which the parameter is passed must be specified by one of the reserved words in, inout or out as follows;

in	When the procedure is invoked, the actual parameter (which may be an expression) is evaluated and copied into the formal parameter location. Regardless of operations performed on the formal parameter, the actual parameter will be left unchanged.
inout	The actual parameter is copied to the formal procedure parameter. On exit the formal parameter value is copied back to the actual parameter. The actual parameter must be a variable.
out	No action takes place on invocation of the procedure. On exit the formal parameter value is copied into the actual parameter. The actual parameter must be a variable.

Note that a formal parameter must be declared as a local variable.

Notes

1. Recursive procedures are not allowed.

8.2 Forward Declarations

A procedure may be declared forward by giving the procedure heading followed by the reserved word forward. The body of the procedure must then be declared later by giving the procedure heading (minus the parameter list) followed by the procedure body.

8.3 Global Declarations

Within a procedure a global declaration specifies what variable identifiers in the next lexical level are going to be used in this procedure. Only the variables specified in the global declaration will be visible in the procedure. The main purpose of this declaration is to provide the compiler with information it can use for allocation of registers.

8.4 The Predefined Procedure Return

There is a predefined procedure declared as

```
procedure return(in loc : word)
```

whose affect is to exit the microprogram and return control to the macro program to continue execution at the location given by loc.

Example procedure

```
procedure multiply(in a, b : word$; out c : word$);

   -- multiply the unsigned integers in a and b to give the
   --    product in c

   begin
      c := a;
       loop
         a := a sll 1;
         b := b srl 1;
         if set(b, 1) then
            c := c + a;
         endif;
         exit when b = 0;
      endloop;
   end;
```

9 Function Declarations

```
<function declaration> ::= function <function identifier>
                              <formal parameter part> : word ;
                              <subprogram block> ;

<function identifier> ::= <identifier>
```

The declaration of functions is similar to that of procedures with the following differences.

9.1 Function Heading

Functions are declared using the reserved word function. In addition to the parameter list the function heading also specifies the result type of the function. The result type may only be word at present.

9.2 Result Assignment

The function result is assigned by an assignment statement contained within the body of the function.

9.3 The Predefined Function Set

There is a predefined function declared as

```
function set(in val, bit : word) : word;
```

which returns true if the bit given by the value bit is set in the value val, or false if the bit is not set.

Notes

1. In MPLS130 the bits are numbered 0 (most significant) to 15 (least significant).

9.4 Function Invocation

A function may be invoked in an expression by giving the function identifier followed by the parameters (if there are any) in parentheses.

Example

```
r := f(a, b);
```

10 Programs

```
<program> ::= program <identifier> ; <block> .

<block> ::= <constant definition part>
               <variable definition part>
               { <function declaration> |
                  <procedure declaration> }
               <body>
```

A program declaration is similar to a procedure declaration, except that :

1. The reserved word program is used instead of procedure.
2. There is no formal parameter list.
3. There is no global declaration.
4. The program block is terminated by a period.

Appendix II : Syntax Of MPLS130

The following pages give the syntax for MPLS130 in Backus-Naur form (BNF).

Note that the symbols { and } are BNF descriptors implying zero or more repetitions of the symbols between them.

```
<program> ::= program <identifier> ; <block> .

<identifier> ::= <letter> { <letter> | <digit> }

<letter> ::= a | b | c | d | e | f | g | h | i | j |
             k | l | m | n | o | p | q | r | s | t |
             u | v | w | x | y | z
             A | B | C | D | E | F | G | H | I | J |
             K | L | M | N | O | P | Q | R | S | T |
             U | V | W | X | Y | Z | $

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<block> ::= <constant definition part>
               <variable definition part>
               { <function declaration> |
                  <procedure declaration> }
               <body>


<constant definition part> ::= const
                                  <constant definition list> ; |
                               <empty>

<constant definition list> ::= <constant definition>
                                  { ; <constant definition> }

<constant definition> ::= <constant identifier> =
                             <constant expression>

<constant identifier> ::= <identifier>

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

<simple constant expression> ::= <sign> <constant term>
                                    { <addop> <constant term> }

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

<empty> ::=

<constant term> ::= <constant factor> { and <constant factor> }

<constant factor> ::= <constant identifier> | <unsigned number> |
                      not <constant factor> |
                      ( <constant expression> )
```

```
<unsigned number> ::= <digit> { <digit> } |
                      #B <bit> { <bit> } |
                      #O <octal digit> { <octal digit> } |
                      #D <digit> { <digit> } |
                      #X <hex digit> { <hex digit> }

<bit> ::= 0 | 1

<octal digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7

<hex digit> ::= <digit> | A | B | C | D | E | F |
                a | b | c | d | e | f

<addop> ::= + | - | or | xor

<logical operator> ::= = | <> | < | <= | > | >= |
                       sll | srl | src | slc

<variable declaration part> ::= var <variable declaration list> ;
                                | <empty>

<variable declaration list> ::= <variable declaration>
                                   { ; <variable declaration> }

<variable declaration> ::= <internal variable declaration> |
                           <external variable declaration>

<internal variable declaration> ::= <identifier list> :
                                       <type clause>

<external variable declaration> ::= <identifier> : <type clause>
                                       <memory location>

<identifier list> ::= <identifier> { , <identifier> }

<type clause> ::= word | word$ | <array type>

<array type> ::= array [ <lower bound> .. <upper bound> ] of word

<lower bound> ::= <constant expression>

<upper bound> ::= <constant expression>

<memory location> ::= at pc <offset> |
                      at memory <constant expression>

<offset> := <constant expression> | <empty>

<procedure declaration> ::= procedure <procedure identifier>
                               <formal parameter part> ;
                               <subprogram block> ;

<procedure identifier> ::= <identifier>

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

```
<formal parameter list> ::= <formal parameter>
                              { ; <formal parameter> }

<formal parameter> ::= <passing specification>
                          <local variable declaration>

<passing specification> ::= in | inout | out

<subprogram block> ::= <global declaration> <block> | forward

<global declaration> ::= global <identifier list> ; | <empty>

<body> ::= begin <statement list> end

<function declaration> ::= function <function identifier>
                              <formal parameter part> : word ;
                              <subprogram block> ;

<function identifier> ::= <identifier>

<statement list> ::= <statement> { ; <statement> }

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

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

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

<simple expression> ::= <sign> <term> { <addop> <term> }

<term> ::= <factor> { and <factor> }

<factor> ::= not <factor> | ( <expression> ) |
             <constant identifier> | <variable> |
             <function call> | <unsigned number>

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

<variable identifier> ::= <identifier>
```

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

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

<actual parameter list> ::= <actual parameter>
                               { , <actual parameter> }

<actual parameter> ::= <expression>

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

<if statement> ::= if <expression> then <statement list>
                      <else clause> endif

<else clause> ::= else <statement list> | <empty>

<case statement> ::= case <expression> of <case body> endcase

<case body> ::= when <constant list> : <statement list>
                  { when <constant list> : <statement list> }

<constant list> ::= <constant expression>
                       { , <constant expression> }

<for statement> ::= for <variable> := <initial value> <direction>
                       <final value>  do <statement list> endfor

<direction> ::= to | downto

<initial value> ::= <expression>

<final value> ::= <expression>

<while statement> ::= while <expression> do <statement list>
                         endwhile

<repeat statement> ::= repeat <statement list> until <expression>

<loop statement> ::= loop <statement list> endloop

<exit statement> ::= exit when <expression>
```

Appendix III : MPLS130 Error Diagnostics

1	program expected
2	identifier expected
4	begin expected
5	end expected
6	then expected
7	endif expected
8	do expected
9	endcase expected
10	endloop expected
11	until expected
12	endwhile expected
13	endfor expected
14	to / downto expected
15	of expected
16	word / word$ expected
17	at expected
18	when expected
19	in / out / inout expected
20	: expected
21	= expected
22	; expected
23	.. expected
24	:= expected
25	[expected
26	] expected
27	(expected
28	) expected
29	. expected
30	+ or - expected
40	number or constant identifier expected
41	base specifier expected
42	digit expected
43	digit out of range given by base specifier
50	low bound exceeds high bound
51	illegal symbol
52	integer out of range
53	vectors may not be used in operations
54	global declaration not allowed at this level
55	type of variable is not array
56	memory address less than zero
57	only one variable allowed in list for external variable declaration
58	external variable declaration not allowed here
59	variable / function identifier not allowed in constant expression
60	word$ not allowed here

70	number of parameters less than formal declaration
71	number of parameters more than formal declaration
72	previous declaration was not forward
73	error in parameter list
74	forward declared; repetition of parameter list not allowed
75	forward declared function; repetition of result type not allowed
76	function result type must be scalar
77	missing result type in function declaration
78	already forward declared
79	recursive procedure / function calls not allowed
80	formal declaration requires variable here
81	type of actual parameter is not the same as type of formal parameter
90	identifier already declared
91	identifier not declared
92	identifier not of correct class
93	not a variable identifier
100	error in declaration part
101	error in constant
102	error in factor
103	error in type
120	too many errors in this line
121	too many nested procedures
122	invalid specifier for compiler option

Appendix IV : Changes Made To MPLS130

In the original design of MPLS130, a number of errors and omissions were made. These principally occurred because no example programs were produced as part of the design phase. Many of these errors affected the power of the programs which can be expressed in the language. In particular, the power to write programs for emulation was severely limited.

As part of this project, the language was improved by making the following changes and additions :

Macromemory References

Additions were made to the syntax of the language to allow the declaration of macromemory variables at specific locations in memory.

As well the facility to access macromemory as an array has been added to the language. The array is called MEM and has a lower bound of zero and an upper bound defined by the size of the macromemory of the machine. As a result of this alteration, the feature in the original language design allowing the declaration of indirect variables is now redundant and has been removed.

PC Operations

The language has been extended so that the program counter (PC) can be treated in a similar fashion to other variables. This facility is particularly necessary for the writing of micro-programmed emulators.

Exit Statements

The use of the exit statement has been clarified and its format changed to

exit when <expression>

If the expression evaluates to a nonzero result, then the for, loop, repeat or while statement in which the exit statement is located is terminated and execution continued at the foot of the statement.

Case Statements

The use of the reserved word when has been introduced to indicate the beginning of each part of the case statement. This change enables each case part to have a statement list instead of just a single statement.

This change has meant that the reserved words begin and end are no longer needed to bracket statement lists in MPLS130, and in fact they are only used to delimit the statement bodies of procedures, functions and programs.

As well each case part may now have several labels instead of just one as in the original language specification.

Parameters

The syntax for the declaration of parameters for procedures and functions has been changed in two ways :

First of all parameters declarations have been altered so that parameters can only be local variables and may not be arrays. These restrictions are made for the sake of efficiency in implementation.

Secondly, more control in specification of the method of parameter passing has been made available. Parameters must be declared as in, out or inout respectively. These changes not only clarify the intention of the program, but also allow the production of more efficient code.

The Predefined Procedure Return

The predefined procedure return was added to the language to provide a means of exiting from a microprogram.

Constant Declarations

The declaration of constants has been extended so that constant identifiers may be equated to expressions involving numbers and previously defined constants.

Miscellaneous Changes

Several minor changes were made to the language as follows :

1. Numbers can be specified in binary, octal, decimal or hexadecimal.
2. The syntax of the loop statement was corrected.

Appendix V : Notes On The Design Of The Parser For MPLS130

The MPLS130 parser is a 3000 line program written in standard Pascal. The source is largely self documenting and contains large numbers of comments. The following is a brief description of some aspects of the design of the parser.

Recursive Descent Parsing

The parser is designed using the technique of recursive descent. Follow sets are used as an aid to error recovery as described by Wirth [Wirt76].

Identifier Tables

The identifier table for the parser is organised as an array of binary trees similar to that of the Pascal P4 compiler [Amman]. Each index of the array DISPLAY contains a pointer to the tree for that level. The user program is considered to be at level one and predefined identifiers are declared at level zero.

All variable identifiers visible at each level are stored in the identifier tree of that level, that is, variables declared in a global declaration in a procedure are reentered in the identifier tree for that level.

Expressions

The parser generates trees for expressions using the techniques described by Aho and Ullman [Aho73]. The generated trees are organised for the optimal generation of code. As part of the process of tree generation, the parser evaluates constants in expressions where possible.

Appendix VI : An Example Of Code Generation

The following example is designed to approximately show the steps in code generation for a segment of MPLS130 code. The example piece of code shows how the sequential search and jump instruction of Slim (the intermediate language of BCPL) could be coded in MPLS130. This example is not meant to be definitive but rather to show the approximate principles of code generation.

```
tl := mem[H];           -- pop the stack into tl
H := H - 1;
repeat                  -- search for the value equal to tl
   t2 := mem[pc];       -- get a value
   pc := pc + 2;        -- move the pc to point to the next value
   A := A - 1;          -- A holds the number of values left
                        --    in the search table
 until (A = 0) or (tl = t2);
                        -- exit if there are no more values or
                        --    the case value has been found
if tl = t2 then         -- if found, place case address
   pc := mem[pc-1]      --    in pc
else                    -- else put the default label
   pc := w;             --    in pc
endif
```

The parser reads the high level code, allocates variables to the registers, scratchpad and macromemory and generates the intermediate code.

For this section of code, assume the following declarations and register allocations were made in the high level code :

```
var A, tl, t2, W, H : word$
```

resulting in :

```
A  = AC0
tl = AC1
t2 = AC2
W  = AC3
H  = GR2
```

The following output intermediate code (in approximate symbolic form) would be generated.

```
readmem    AC1 GR2       ; pop stack
dec        GR2
label      1             ; start of repeat statement
readmem    AC2 PC        ; read a value
regassign  PC + 2        ; move pc to next value
dec        AC0           ; A := A - 1;
zerotest   AC0   4   2   ; if AC0 = 0 goto label 2 else goto 4
label      2
equaltest  AC1 AC2 4   3 ; exit if tl = t2 else goto 3
label      3
goto       1             ; go back to top of loop
label      4             ; exit point for loop
equaltest  AC1 AC2 5 6   ; if tl = t2
label      5
```

```
readmem     PC   PC - 1     ;     then pc := mem[pc];
goto        7
label       6
regcopy     PC   W          ;     else pc := w
label 7
```

From this intermediate language the following templates would be produced :

```
    AR  GR2     A        N     S
            AC1                  READ      BMEM
    AR  GR2     AM1 FA   L
L1
    AR  PC      A        N     S
            AC2                  READ      BMEM
    CON PC  PC  APB FA   L
    AR  AC0     AM1 FA   L
    AR  AC0     A        N                      ALUZ      L4  L2
L2
    AR  AC1 AC2 AMB      N                      ALUZ      L4  L3
L3
                                                NC            L1
L4
    AR  AC1 AC2 AMB       N                     ALUZ      L5  L6
L5
    AR  PC      AM1       N     S
            PC                    READ     BMEM
                                                NC            L7
L6
    Z   PC  W   APB FA    L
L7
```

This code will be compacted by the packer, and the default values for unspecified fields filled in by the emitter, to give :

```
    AR  GR2 -   A   -     N CLR S -    -    -    NC      -
    AR  GR2 AC1 AM1 FA    L CLR - READ -    BMEM NC      - -
L1  AR  PC  -   A   -     N CLR S -    -    -    NC      - -
    CON PC  PC  APB FA    L CLR - -    -    -    NC      - -
    AR  AC0 AC2 AM1 -     L CLR - READ -    BMEM ALUZ    - L4  L2
L2  AR  AC1 AC2 AMB -     N CLR - -    DCH  -    ALUZ    - L4  L1
L4  AR  AC1 AC2 AMB -     N CLR - -    DCH  -    ALUZ    - L5  L6
L5  AR  PC  -   AM1 -     N CLR S -    -    -    NC      - -
    -   -   PC  -   -     - CLR - READ -    BMEM NC      - -   L7
L7
```

References

[Aho73] A V Aho & J D Ullman
The Theory Of Parsing, Translation, And Compiling;
Vol II : Compiling
Prentice-Hall 1973

[Amman] U Amman et alii
The Pascal P-Compiler
Implementation Notes & Source Listings

[DG77] Programmer's Reference :
S/130 Microprogramming With WCS Feature
Data General Corporation 1977

[JW78] K Jensen & N Wirth
Pascal User Manual And Report (2nd edition)
Springer-Verlag 1978

[Kern78] B W Kernighan & D M Ritchie
The C programming Language
Bell Telephone Laboratories, Incorporated 1978

[Sint80] M Sint
A Survey Of High Level Microprogramming Languages
Sigmicro Vol 11; 3,4 1980

[Soon81] Soon Sow Lai
A High Level Language For Microprogramming
On The Eclipse S/130
B Sc Honours Project Report;
University of Canterbury 1981

[Wilk51] M V Wilkes
The Best Way To Design An Automatic Digital Computer
Manchester University Computer Inaugral Conference
Proceedings, Manchester, England 1951

[Wirt76] N Wirth
Algorithms + Data Structures = Programs
Prentice-Hall 1976