Micro SPL

Henry Baker and Clinton Parker

September 1979

Abstract

This document describes the Micro-SPL language and compiler. The Micro-SPL compiler converts programs in Micro-SPL, a high level programming language similar to Algol, directly into microcode for the Xerox Alto minicomputer. The generated microcode is designed to interface with main line programs written in BCPL and the compiler automatically generates an interface that allows the BCPL main line to reference Micro-SPL routines as if they were external BCPL routines.

The Micro-SPL compiler generates microcode which is competitive with hand microcode, yet takes only 30-50% as long to write and 10% as long to debug. Micro-SPL generated microcode runs over ten times faster than an equivalent BCPL program and perhaps half as fast as good hand written microcode without losing the advantage of writing in a high level language.

Chapter 1: Introduction

This document describes the Micro-SPL programming language and compiler. Micro-SPL is a high level programming language which will generate Alto microcode as its object code. The compiler described in this document generates object code which interfaces with BCPL. This compiler is still under development and therefore lacks some features.

Each chapter of this document covers a different aspect of the compiler and the Micro-SPL language. For those readers who are interested only in using the compiler, the chapters on Micro-SPL syntax and How to Use should be all that is needed. For the readers who must know how everything works there is a chapter which presents an overview of the implementation of the compiler. The last chapter of this document describes the BCPL interface code generated and the general philosophy of linkage between the microcode and BCPL. Finally, there are several appendices that can be used for quick reference.

In general, the following styles will be used throughout this document:

> *Italics will be used for all important words or topics.*
> **Bold face will be used for Reserved or Key words (in syntax).**
> A small font like this will be used for fine points (less important).

All example Micro-SPL code is indented to distinguish it from the surrounding text.

Chapter 2: Micro-SPL Language Description

This chapter describes the Micro-SPL language used for the current version of the Micro-SPL compiler. It is not intended to be used as tutorial for the language and assumes that the reader has familiarity with an Algol-like programming language. A full list of Micro-SPL syntax is given in Appendix A and a list of reserved and key words is given in Appendix B. Appendix C contains a sample program which demonstrates most of the features of Micro-SPL described below.

2.1 Programs

Micro-SPL is a high level programming language that resembles Pidgin Algol. A Micro-SPL program consists of a series of *routine declarations* followed by the reserved word **START**.

2.2 Routine Declarations

There are two kinds of routine declarations in Micro-SPL, these are *functions* (**FUNC**) and *procedures* (**PROC**). Both functions and procedures can be declared to be accessible from other separately compiled code (BCPL). This is specified by preceding the routine declaration by the key word (**ENTRY**). **ENTRY** should be used for all Micro-SPL routines since only external (BCPL) references can be made to them. All Micro-SPL routines must appear in the same compilation. There is no provision for separate compilations. Functions must have a type in Micro-SPL, but currently the only legal type is *integer* (**INT**).

Following the reserved words **PROC** or **FUNC** is the *heading*. The heading is used to declare the arguments to the routine if there are any. The heading consists of an *argument declaration list* which is enclosed by a pair of parentheses. The argument declaration list is very similar to a *declaration list* (described below) with the following exceptions. First, all the *argument declarations* (described below) must be separated by commas and the reserved word **REF** proceeds the declaration of arguments being passed by reference. Micro-SPL uses the BCPL calling convention (call by VALUE) unless **REF** is specified. Also, **REF** does not make much sense for arrays (see declaration below) since an array argument is assumed to be a pointer to the start of the array.

After the heading comes the *body* of the routine. The body of the routine consists of a *declaration list* (described below) followed by a *statement list* (described below). The declaration list defines the local variables that are to be used by the statement list and can be omitted if there are no local variables. The statement list is the actual executable code for the routine and should end with a *return* statement. The **RETURN** statement is not required for a procedure, but if a function has no *return*, then a warning message will be given and a value of *zero* will be returned. Micro-SPL considers *zero* to be *FALSE* and any *non-zero* value to be *TRUE*.

2.3 Declaration Lists

Declaration Lists are used to define the local variables that are used in the *statement list* of the *body* of a routine in Micro-SPL. A declaration list is, as the name implies, a list of *declarations*. A declaration is a *type specification* followed by a *variable list*. In Micro-SPL there are only two legal type specifications: **INT** and **INT ARRAY**. **INT ARRAY**s are actually only useful as pointers in Micro-SPL since no storage allocation is provided, thus no subscript is needed when declaring an *integer array*. A *variable list* is a list of *identifiers* separated by commas.

2.4 Statement Lists

A *statement list* is a list of *statements* optionally separated by semicolons. There are five basic statements in Micro-SPL and they are:

assignment statement
if statement
while statement
exit statement
return statement

The description of these statements is given below.

2.4.1 Assignment Statement

The *assignment statement* has the form *variable* ← *expression*. *Variable* can be any type defined above including a subscripted array. ← can be either ← or :=. *Expression* can be any legal expression as described below.

2.4.2 If Statement

The *if statement* has the form **IF** *expression* **THEN** *statement list* **ELSE** *statement list* **FI**. The **THEN** *statement list* is executed if *expression* is *TRUE* (non-zero), otherwise the **ELSE** *statement list* is executed (if there is one).

Only enough of the *expression* is evaluated to determine if it is *TRUE* or *FALSE*; once this has been determined, the remainder of the *expression* is ignored. This is called "short-circuit" evaluation.

2.4.3 While Statement

The *while statement* has the form **WHILE** *expression* **DO** *statement list* **OD**. *Expression* is evaluated as described for the *if statement* above. *Statement list* will continue to be executed as long as *expression* is *TRUE*. Note that if *expression* evaluates to *FALSE* initially, then *statement list* will *not* be executed.

2.4.4 Exit Statement

The *exit statement* has the form **EXIT** and causes the *flow of control* to continue following the **OD** of the inner most *while statement* in which the **EXIT** is contained.

2.4.5 Return Statement

The form of the *return statement* depends on the routine type. If the routine is a *procedure*, then the form is just the reserved word **RETURN**. For *functions*, the form is **RETURN**(*expression*), where *expression* is the value to be returned.

2.5 Expressions

An *expression* is basically any combination of *boolean* and *arithmetic operators* applied to program *variables* and *constants*. Given below is a list of legal *operators* and their precedence. *Parentheses* can also be used to change the order of precedence. *Operators* of equal precedence are evaluated from left to right.

(expression)
NOT, **~**, **-** (unary minus)
RSH, **LSH**
+, **-**
<>, **~=**, **=**, **<**, **<=**, **~>**, **>**, **>=**, **~<**, **NE**, **EQ**, **LS**, **LE**, **GR**, **GE**
AND, **&**
OR, **%**
XOR, **!**

Where **(***expression***)** has the highest precedence and **XOR** has the lowest. Please refer to the syntax description **(***<exp>***)** in Appendix A for a complete description of the Micro-SPL *expression* syntax.

2.6 General

In addition to the above description, the following items of Micro-SPL syntax should be mentioned. First, comments can be included in the text by preceding the comment by //. Everything between // and the next carriage return will be ignored. *Octal numbers* can be entered by preceding the octal value with a #. Any character not defined in the syntax (Appendix A) is considered a legal separator and ignored otherwise.

Chapter 3: How to Use Micro-SPL

This chapter gives a brief description of what is needed to run Micro-SPL. It is assumed that the user already has the BCPL compiler and the BCPL loader (BLDR) on their disk in addition to the normal BCPL software support libraries.

3.1 Files Needed

The following files contain Micro-SPL and the non-standard routines needed to run Micro-SPL:

MicroSPL.RUN
RegDefs
Mu.RUN
PackMu.RUN
LoadRam.BR
AltoConsts23.mu

Any .BR files that are used by the BCPL program other than those mentioned above must be supplied by the user.

3.2 How to Execute the Micro SPL Compiler

The general format for compiling a Micro-SPL program is:

>MicroSPL.RUN/**<switches> <Micro-SPL source> <microcode output> <BCPL output>**

where:

The ">" before MicroSPL.RUN is typed by the executive.

<switches> are:

c — compile only, do not create command file to run Mu and PackMu on **<microcode output>** and BCPL on **<BCPL output>**.
d — debug mode. This option is useful only for debugging the compiler.
h — hold the screen for observation at end of compilation. If this switch is not given, Micro-SPL will return directly to the executive at the end of compilation if there are no errors or warnings.
l — show **<microcode output>** as it is being compiled.
s — show **<Micro-SPL source>** as it is being compiled.

<Micro SPL source> is the file name of the Micro-SPL program to be compiled.

<BCPL output> is the file name of the BCPL interface file to be generated. This file is optional and if not provided, the BCPL interface code generated will be displayed on the screen. **Nota Bene**: *If this file is present, it must NOT have the same 'base' file name as **<microcode output>** since this would result in two .BR files with the same name to be generated.*

3.3 Example

The following is an example of the command used to compile the Micro-SPL program contained in **Heap.SPL** (the sample program in Appendix C) and generate the microcode output file **HeapMu.Mu** (Appendix D) and the BCPL interface in the file **Heap.BCPL** (Appendix E). The Micro-SPL compiler will automatically set up a command file which causes the Mu assembly of **HeapMu.Mu** (generating **HeapMu.MB**), the conversion of **HeapMu.MB** to **HeapMu.BR** (using PackMu.Run), and the BCPL compilation of **Heap.BCPL** (generating **Heap.BR**). The file **HeapMu.MB** is not needed after **HeapMu.BR** is created.

>MicroSPL.RUN **Heap.SPL HeapMu.Mu Heap.BCPL**

To LOAD the program in the above example the following .BR files would have to be added to the list of "BR's" being loaded:

Heap
HeapMu
LoadRam

LoadRam contains the routines that will load the microcode RAM at run time with the microcode generated by Micro-SPL, while **Heap** and **HeapMu** are the routines just compiled.

Chapter 4: Compiler Description

This chapter describes the internal organization of the Micro-SPL compiler. The compiler is written in BCPL and uses a recursive-descent parsing technique, with the exception of the expression analyzer which uses an iterative technique to parse the expression into reverse polish notation. The sections of this chapter are broken into the six major functions of the compiler. These are *routine (segment) specification, declarations, statement lists, expressions, internal code generation* and *Mu source generation*.

4.1 Routine Specifications

Routine (segment specifications are handled by the procedure ***Segment***. This procedure determines the type of the routine (**PROC** or **INT FUNC**) and sets up the symbol table entry for the routine name. After parsing the routine name, ***Segment*** calls the procedure ***Heading*** if there is an argument list. This procedure parses the argument list by repeated calls to the procedure ***Declare*** (described below). ***Segment*** then calls ***Declare*** until the local declarations have been parsed. At that time ***StmtList*** (described below) is called to parse the body (statements) of the routine. Finally ***EndSeg*** is called to 'clean up' the current routine being parsed.

EndSeg performs several functions, the first is to save the number of arguments, the amount of local storage used in the table which keeps track of information about each procedure (pointed to by ***ptBase*** (base location of procedure table) and ***ptLoc*** (current position in procedure table)). ***EndSeg*** then frees any local variables that were defined in the current routine (see description of ***ValidName*** and symbol table below). Finally, EndSeg inserts a *return* if one is needed and inserts return labels in all of the locations in the linked list pointed to by ***retList*** (see ***ReturnStmt*** below).

4.2 Declarations

Declarations are parsed by the procedure ***Declare***. This procedure is simply a loop which is executed as long as there is a type specification (**INT** or **INT ARRAY**). If the type is **INT** then the procedure ***DclInt*** is called to parse the *integer* declaration. If the type is **INT ARRAY**, then ***DclArray*** is called to parse the *integer array* declaration.

DclInt is the procedure that checks to see if the identifier being declared is a legal variable, and if so, makes the symbol table entry for that identifier. The function ***ValidName*** (described below) is used to determine if the identifier is a legal variable name. If the identifier is a legal variable name, then ***DclInt*** fills in the symbol table information and allocates storage space for that variable. If the identifier being declared is an argument to the routine, then ***DclInt*** will generate internal code to properly load the argument so that it can be referenced in the routine (see Chapter 5 for more details on the BCPL/microcode interface). The **REF** type declaration is also handled by ***DclInt*** if it is given in an argument list (see Symbol Table Format below). Finally, for each variable declared, ***DclInt*** generates an internal code for the register/variable name equivalence comment (see Figure 1).

DclArray is similar to ***DclInt*** except that it does not permit an array argument to be declared *reference*. As one might assume, ***DclArray*** sets the type to *integer array* instead of *integer*.

ValidName is a function that returns true if the current identifier is a legal variable name. If the identifier is a legal variable name, then ***ValidName*** also sets up the table that will be used by ***EndSeg*** (described above) to free the local variables at the end of the current routine being parsed.

4.3 Statement Lists

Statement Lists are parsed by the procedure ***StmtList***. ***StmtList*** consists of a loop which is repeated as long as there are legal statements (see section 2.4). *Integer assignment statements* are parsed by the procedure ***AssignStmt***. *Integer array assignment statements* are parsed by the procedure ***AAssignStmt***. *If statements* are parsed by the procedure ***IfStmt***. *While statements* are parsed by the procedure ***WhileStmt***. *Return statements* are parsed by the procedure ***ReturnStmt***. *Exit statements* are parsed by the procedure ***ExitStmt***. All of these procedures are described below.

AssignStmt first determines the type of the assignment (**INT** or **REF INT**) and then checks for the assignment operator. After the assignment operator is parsed, the right hand side of the assignment is parsed by ***ExpHandler*** (see Expressions below). After the right hand side has been parsed, ***CodeGen*** is called to generate the code for the actual assignment (see Internal Code Generation below).

AAssignStmt first determines the type of the assignment (*array* or *pointer* (array without subscript)) and then checks for the assignment operator. If the type is a *subscripted array*, then ***ExpHandler*** is called to parse the subscript. After the assignment operator is parsed, the right hand side of the assignment is parsed by ***ExpHandler*** (see Expressions below). After the right hand side has been parsed, ***CodeGen*** is called to generate the internal code to store the right hand side value into the array element (see Internal Code Generation below).

IfStmt first evaluates the conditional expression by calling ***ExpHandler*** with the static variable ***labType*** equal to 2 (the numeric code for generating **IF** conditional labels, see section 4.5.1). This causes the expression handler to generate conditional code instead of arithmetic code for relational operators. On return from ***ExpHandler***, ***IfStmt*** checks to see if the top item on the variable stack (***varStack0***) is of type ***TEST***. If it is not then ***TestZero*** is called to generate the equivalent of ((top of stack) **NE** 0). In this state the global static ***compRel*** (comparison relation list) contains conditional labels for the **true** and **false** cases of the conditional expression parsed (see Internal Code Generation below for a description of the format of the *comparison relation list* and conditional labels). An internal condition label instruction (**label** field is TRUE, see figure 1) is then generated for each **true** condition label in the comparison relation list. Now that the labels for the **THEN** part of the if statement have been generated, ***StmtList*** is called (recursively) to parse the statements of the **THEN** part. ***IfStmt*** then checks to see if there is an **ELSE** part. If there is, then internal code is generated for each **false** condition label in the comparison relation list as described above for the **THEN** part. Before this is done though, another label is generated and appended to the last statement of the **THEN** part (this is to jump around the **ELSE** part). After the **false** condition labels are processed, the comparison relation list is replaced with one that contains only the label generated to jump round the **ELSE** part in the **false** position. ***StmtList*** is then called to parse the statements in the **ELSE** part. Finally, whether there is an **ELSE** part or not, internal code is generated for each **false** condition label in the comparison relation list.

WhileStmt first saves the current value of ***endWH*** (see ***ExitStmt*** below) and then generates a label (the **lab** local variable) and appends it to the last internal instruction generated. ***WhileStmt*** then evaluates the conditional expression by calling ***ExpHandler*** with the static variable ***labType*** equal to 3 (the numeric code for generating WH conditional labels, see section 4.5.1). This causes the expression handler to generate conditional code as opposed to arithmetic code for relational operators. On return from ***ExpHandler***, ***WhileStmt*** checks to see if the top item on the variable stack (***varStack0***) is of type ***TEST***. If it is not then ***TestZero*** is called to generate the equivalent of ((top of stack) **NE** 0). In this state the global static ***compRel*** (comparison relation list) contains conditional labels for the **true** and **false** cases of the conditional expression parsed (see Internal Code Generation below for a description of the format of the comparison relation list and conditional labels). An internal condition label instruction (**label** field is TRUE, see figure 1) is then generated for each **true** condition label in the comparison relation list and ***endWH*** is set to the first **false** condition label in the comparison relation list. Now that the labels for the body of the while statement have been generated, ***StmtList*** is called (recursively) to parse the statements in the body. ***WhileStmt*** then appends the label generated at the top of the loop (**lab**) to the last statement of the body (this is the branch to the top of the while loop to execute the conditional code). Finally, internal code is generated for each **false** condition label in the comparison relation list (exit point of loop) and ***endWH*** is restored to its previous value.

ReturnStmt first checks to see if there is a return value. If there is, then ***ExpHandler*** and then ***CodeGen*** are called to get the return value in AC0 (see Chapter 5 for a description of the BCPL return conventions). The global static ***retList*** is then appended to the last internal instruction generated and ***retList*** is updated to point to where it was appended (linked list of return labels, see description of ***EndSeg*** above).

ExitStmt simply appends ***endWH*** (described in ***WhileStmt*** above) to the last internal instruction generated using ***AppendLab***. Since ***endWH*** is set by ***WhileStmt*** to be a label to the end (**false** condition label) of the innermost while loop currently being parsed, this is all that is needed to do an **EXIT**.

4.4 Expressions

Expressions are parsed by the procedure ***ExpHandler*** as follows. ***ExpHandler*** parses the current expression into *reverse polish notation* using two stacks. The first stack (global statics ***varStack0***, ***varStack1***, and ***varStack2*** referenced by ***varStackTop***) is used to hold the description of the variables being parsed and temporaries used in the course of the evaluation of the expression. The other stack (locals ***stack***, ***sprec***, and ***stackaddr*** referenced by local ***stackTop***) is used to hold the operations pending.

The body of ***ExpHandler*** consists of a *while loop* to parse the expression followed by some cleanup code. The while loop is executed as long as two variables or operators do not occur back to back (with the exception of unary operator following a binary operator). *Operators* are handled by the procedure ***StackMgr*** (described below) and *variables* are handled by the procedure ***PushST*** (described below) with the exception of *arrays* which are handled by the procedure ***ArrayElt*** (described below). At the end of the while loop all of the operators remaining in the operator stack are *popped* off as described in the description of ***StackMgr*** below.

StackMgr is used to push operators onto the operator stack in accordance with their precedence (this means that code is generated 'on the fly' for expressions as they are parsed, rather than waiting until the whole expression has been parsed). When ***StackMgr*** is called with a new operator, all operators of equal or higher precedence are *popped* from the operator stack and passed to ***CodeGen*** (see Internal Code Generation below) and then the new operator is *pushed* onto the operator stack. *Popped* means that the entry in the operator stack pointed to by ***stackTop*** is removed and ***stackTop*** is decremented by one. *Pushed* means that ***stackTop*** is incremented by one and then is used as the location in the operator stack to store the new operator.

PushST is used to push a variable's mode onto ***varStack0*** and its address onto ***varStack1***. This routine also checks to see if the ALU L register needs to be saved by calling ***SaveL***. ***SaveL*** checks to see if the mode in ***varStack0*** is LREG and if so, it then allocates a register from the free register pool (by calling ***NextReg***) and generates an internal instruction to save L in that register (see section 4.5.2 below). ***NextReg*** searches the free register pool (pointed to by the static ***regStack***) for a positive entry. Once one is found, it remembers the register number for the return value and negates the value in the table to indicate that it has been allocated. The routine ***FreeReg*** deallocates the register passed to it so that it can be reallocated in the future.

ArrayElt is used to push an array variable onto the expression stack. First it calls ***PushST*** with the array variable as an argument. It then checks to see if there is a subscript. If there is, it calls ***ExpHandler*** recursively to parse the subscript and then calls ***CodeGen*** to generate the code to get the array element.

4.5 Internal Code Generation

CodeGen generates internal code for ten types of expressions. These are *Assignment*, *Plus*, *Minus*, *Shifts*, *And/Or*, *Exclusive Or*, *Relations*, *Relational Negation*, *Unary Minus*, and *Array References*. The type of the current expression is determined by the *opcode* passed to ***CodeGen***. This opcode is used as the index to a case statement which determines the appropriate routine to use to generate internal code for that opcode. The actual description of these routines is given below, but first the internal code format and the routines that generate it will be described. It is a good idea to refer to Appendix D while reading this section since examples of most of the microcode described below can be found there.

4.5.1 Internal Code Format

The *internal code format* is designed to represent the information needed to generate Mu source code. There are four types of internal instructions. The type is specified by the two high order bits of the first word of the instruction (see Figure 1). Only instructions of type 0 generate actual Mu instructions. Type 1 instructions generate label declarations for branching. Type 2 instructions are used for inserting comments about register/variable equivalences in the output. Type 3 instructions are used for formatting information. Currently the only formatting done is the insertion of extra carriage returns into the output.

Type 0 instructions can be of four forms: the first is a label declaration while the other three are variations of a basic instruction. If the label field (bit 2, bits are numbered from left to right starting at 0, see figure 1) is **true** (non-zero) then this is a *label instruction*. There are two kinds of label instructions. The first is a *textual label* which is denoted by ALU functions fieldbits 12 through 15 (bits 2 through 15) being zero. If this is the case then the second word of a label instruction contains a pointer to the BCPL format string that contains this label's name. If the ALU Function field is non-zero then the second word of the label instruction is a *conditional label description* (see Figure 3). The type field (bit 2 through 4) of the conditional label description specifies the type. The type is used to generate a label of the form *type*#, where *type* is COND for 0, IF for 1, and WH for 2. The # specifies that the value in the number field (bits 5 through 15) is to be appended to name for the type field. Thus, if the type field is 1 and the number field is 7, then the label IF7 would be generated.

If the label field of a type 0 internal instruction is **false** (zero) then this is a normal microcode instruction as described in figures 1, 2 and 3. For further information as to the function of the fields, see the Hardware manual.

4.5.2 Basic Routines

The basic routines come in two flavors. First, there are the routines to generate the actual internal code and then there are the routines to load variables into the ALU registers L and T. There are three primitive routines to generate the internal instructions. They are ***Push0***, ***Push2***, and ***Push3***. All of these routines put one or more words into the internal code list and check for too much code being generated. They also print the words being put in the instruction list if the debugging flag is on. ***Push0*** does this for one word. ***Push2*** inserts two words and ***Push3*** inserts three words. These two routines also set the static ***lastOp*** to the current value of ***codeTop*** (pointer to last word in instruction list) before inserting the words into the list. The static ***lastOp*** is used for code improvements described below.

LoadL and ***LoadT*** are used to load the T and L ALU registers from variables. They are both passed three arguments. The first is the ALU function to be used when the register is loaded, the second is the mode of the variable, and the third is the address of the variable. Both routines have five different possible modes. They are *constant* (CONST), *register on the stack* (STACK), *local variable* (LOCAL), *reference variable* (REFVAR) and *ALU L register* (LREG). CONST, STACK, and LOCAL generate the same code for both routines with the exception that ***LoadL*** sets the L field (bit 6, see figure 1) of the first instruction word generated and ***LoadT*** sets the T field (bit 7). The code for CONST generates a three word internal instruction in which the constant field (bit 8 of the first word) is set to true and the address of the value of the constant used in the third word, LOCAL generates a two word internal instruction in which the register field (bit 9 of the first word) is set to **true** and address is used for the register number (bits 10 through 15 of the second word). STACK generates the same code as LOCAL, except that it also returns the register specified by the address to the free register pool by calling ***FreeReg*** described above.

The REFVAR and LREG code is different for ***LoadL*** and ***LoadT***. The REFVAR code for ***LoadL*** checks to see if the last internal instruction generated was a load T register. If it was, then it saves this last instruction, backs up ***codeTop*** to delete the old instruction, and then generates the memory request for the reference variable. The saved instruction is then inserted after the instruction that started the memory request. This sequence makes better use of the memory fetch delay than the other code which just starts the memory request and puts in a NOP for the delay. In both cases, the final instruction generated is to load the L register with the memory value. ***LoadT*** for REFVAR generates the last case described above for ***LoadL***, with the exception that the last instruction loads the T register instead of the L. The LREG for ***LoadL*** just checks to see if the operation passed in is non-zero. If it is then we want to perform an operation on the L register and put the result back into the L register, so an instruction to do this is generated. If the operation is zero, then nothing is done, since it doesn't make sense to load the L register with itself. ***LoadT*** for the LREG case, though, is more complicated. First we want to see if the last instruction that loaded the L register is one that could have loaded the T register. If it is, then the previous instruction is modified so that it loads the T register. You can see an example of why this is useful in section 4.5.4 below. If the last instruction cannot load the T register, then an instruction to load the T register from the L register is generated.

4.5.3 Assignment

The code generated for an assignment depends on the mode of the variable being assigned. There are three modes which are recognized by the assignment code. They are *local variables* (LOCAL), *reference variables* (REFVAR) and *local arrays* (LOCARRAY).

The code for LOCAL loads the ALU L register (if it is not already loaded) by calling ***LoadL***. It then sets the TASK field bit 2 of the second instruction word) of the last generated instruction (pointed to by ***lastOp***) so that proper TASKing will occur. Then it generates an internal instruction to load the local register from the L register.

The code for REFVAR generates an instruction to start a memory request of the address pointed to by the reference variable. It then generates a TASK instruction and finally stores the top item of the stack in that memory location.

The code for LOCARRAY loads the subscript of the array into the ALU T register unless the subscript is zero or one. It then generates a memory request at the address pointed to by the base of the array plus the subscript. If the subscript was one, this is done by using the increment function of the ALU. Then, as in the REFVAR case, a TASK and store instruction is generated.

4.5.4 Plus

The routine to generate code for the Plus operator checks to see if the second operand is one. If it is, then an instruction to use the increment function of the ALU is generated. If it is not an increment, then the T register is loaded with the second operand. If the code to load the T register does not involve an ALU function, then the ALU function of that instruction is set to a decrement instruction and the add is performed by generating an increment T ALU instruction. Otherwise, a simple add instruction is generated. The reason for generating the increment T instruction when possible, is that the T register can be used for the output of this instruction. Thus, the microcode generated for A(b+c) would be equivalent to:

```
T ← c-1 ;
T ← b+T+1 ;
MAR ← A+T ;
```

4.5.5 Minus

If the second operand is one, then an instruction to use the decrement function of the ALU is generated. If it is not, then the T register is loaded with the second operand and an instruction to subtract the first operand from the T register is generated. In both cases, the result is left in the L register.

4.5.6 Shifts

Shift instructions are generated by the routine ***Shift***. This routine requires two arguments, the first is the type of shift and the second is the mask for that type of shift. The mask is 177400 (octal) for a left shift and 377 (octal) for a right shift. ***Shift*** first allocates a temporary R register (reg) by calling ***NextReg***. It then checks to see if the shift count is a constant. If it is and it is greater than 15 then the constant 0 is placed on the variable stack and reg is freed. Otherwise, if the shift is more than 7 bits then the operand is rotated 8 bits using the swap bytes instruction, an instruction to AND it with the mask is generated and the shift count is decreased by 8. The remainder of the shifts are done by the routine ***Shift1*** which is performed once for each shift. ***Shift1*** generates instructions of the following form:

```
L ← reg, TASK ;
reg ← L LSH 1 ;
```

Note that the first time Shift1 is called the first 'reg' will be the operand to the shift. The variable stack is then set to indicate that the result is in the temporary reg.

The compiler currently cannot handle shift counts which are not constants. The plan is to call a microcode subroutine in that case. However, this scheme has not yet been implemented.

4.5.7 AND and OR

The logical and arithmetic functions for **AND** and **OR** are performed by the routine ***ConjExp***. This routine requires two arguments. The first is the type of operation (**AND** or **OR**) and the second is the address of where the second operand for this operator started generating code. This address is saved by the ***StackMgr*** described in section 4.4 and is passed as an argument to ***CodeGen***. The first thing that ***ConjExp*** does is to determine if either of the operands is a TEST (see section 4.5.9 below). If either of them are, then this means that logical interpretation is to be used for the operator, otherwise it is an arithmetic operation. If it is an arithmetic interpretation, then the T register is loaded with the second operand and then the L register is loaded using the operation passed as the first argument. If the interpretation is logical, then each of the operands is checked to see if they are TESTs. If an operand is not a TEST then ***TestZero*** (described below) is called to make it into a TEST. After this is done, all of the instructions following the address passed as the second argument are shifted down two words in the instruction list to make room for a new label. If the operator is **AND**, then this label is filled in with the label for the **true** result for the first TEST operand from the comparison relation list (this is ***compRel!(addr2+1)***, see a description of the comparison relation list below) and the list entry is set to zero. If the operator is **OR**, this label is filled in with the **false** label for the first TEST operand list (this is ***compRel!(addr2+2)***) and the list entry is set to zero. This causes the second TEST to be performed only if the first one was **true** for the **AND** case or was **false** for the **OR** case.

TestZero is used to generate a comparison relation for the operand passed to it. This is done by loading the L register with the operand and using the BUS=0 microcode comparison. After this is done, two labels are generated and a label declaration is generated. Finally, a NOP instruction is generated with the false label and a comparison relation list entry is made (see section 4.5.9).

4.5.8 Exclusive Or

The *exclusive or* operation is performed by loading the T register with the second operand and then loading the L register using the XOR ALU function. Thus, a XOR b would generate code similar to:

```
T ← b ;
L ← a XOR T ;
```

4.5.9 Relational Operators

CondExp is called to generate internal code for relational operators. This routine requires four arguments, the first is the type of ALU operation that is to be performed on the two operands before the comparison, the second is the type of comparison to use, the third is the normal mode for the false branch of the ALU comparison and the last is true if this is an unsigned compare. For example, if the operation was Greater Than (**>**), the arguments would be MINUST1, LESS, **true** and **false**. MINUST1 means generate an instruction of the form L¨ a-T-1; and LESS means use SH<0. Thus, for a>b code similar to the following would be generated:

```
T ← b ;
L ← a-T-1 ;
NOP, SH<0, TASK ;
!1,2,IF5,IF6 :
NOP, :IF5 ;
```

In the above example, control would go to IF5 if a is greater than b, otherwise it would go to IF6.

When ***CondExp*** is called, it checks to see if the second operand is a constant that can be subtracted from the first operand without loading the T register. If it can be, the appropriate instruction is generated, otherwise the T register is loaded with the second operand and the L register is loaded by applying the ALU function passed as the first argument to the first operand and the T register. Then the comparison operation passed as the second argument is used to generate the comparison instruction third line in the above example). Two labels are generated and a label declaration is generated (fourth line above). Finally, a NOP instruction with the label to the normal mode (third argument) is generated. The static ***labType*** (used for generating conditional labels) is checked to see if this relational operation is a conditional expression or an arithmetic expression (right hand side of an assignment). If it is a conditional expression, then the true and false branches are entered into the comparison relation list (described below). Otherwise, internal code is generated to load the L register with -1 for the **true** case and 0 for the **false** case.

A *comparison relation list* is used by the compiler to keep track of the true and false labels for conditional expressions. The first word of the comparison relation list is its size (in words). Following the size are three word groups (an *entry*), one for each conditional expression. The first word of each *entry* is a pointer to where the code for that comparison begins. The second word is a conditional label (see section 4.5.1) for the **true** branch for the conditional and the third word is a conditional label for the **false** branch. A zero in either the **true** or **false** branch word means no label is defined for that case.

4.5.10 Relational Negation

If the operand to be negated is on the right hand side of an assignment then the operand is exclusive or'ed with minus one. Otherwise, the operand is made into a TEST (see ***TestZero*** above) if it is not already one, and the **true** and **false** cases for the operand are reversed in the comparison relation list.

4.5.11 Unary Minus

If the operand is a constant then it is just negated. Otherwise, the L register is saved by calling ***SaveL*** (see section 4.4), the operand is loaded into the T register and then the T register is subtracted from zero leaving the result in the L register.

4.5.12 Array Reference

The routine to generate code for an array reference checks to see if the subscript is zero or one. If it is zero, then an instruction to use the base address of the array is generated to start the memory request. If it is 1, then an instruction to increment the base address is generated to start the memory request. Otherwise, the T register is loaded with the subscript and the T register is added to base address to start the memory request. Following this, an instruction to save the address in the SAD register is generated, and finally, the L register is loaded with the result of the memory request.

4.6 Mu Source Generation

The primary output of Micro-SPL is source level Alto microcode which can be compiled by Mu. This section describes how this output is generated from the internal code format. Please refer to Figures 1, 2, and 3 and section 4.5 before reading this section.

The first step in generating the microcode output is to generate code to get the microcode definitions that are in AltoConsts23.mu and RegDefs. Next the code used to return from the Micro-SPL routine is generated. Following this, the dispatch table that will be used to reference each of the Micro-SPL routines on entry to the microcode is generated. This is followed by the code to load the first, second and third arguments and save a pointer to the other arguments if there are any. After this, the actual microcode for the routines is generated. This is done by translating the fields of the internal format into textual Mu instructions (see section 4.5 for a description of internal format).

Chapter 5: BCPL/Microcode Interface

This chapter describes the BCPL code generated to link the microcode routines generated with other BCPL routines. The goal was to make the Micro-SPL routines look as much as possible like BCPL routines to the person who is using them and at the same time make them as efficient as possible.

In order to make the Micro-SPL look like BCPL routines it was necessary to load the control RAM automatically for the user. This is done by the static ***loadFlag*** which is initialized to **true** in the BCPL interface routine. A test is inserted at the beginning of each dummy procedure (described below) to test if this variable is true. If it is, then ***MicroInit*** routine (defined in the BCPL interface code) is called to load and initialize the microcode. This routine sets ***loadFlag*** to **false** so that the microcode will not be initialized a second time.

For each routine that was defined in the Micro-SPL source, a dummy BCPL routine is generated in the interface routine. This dummy routine initializes the microcode (as described above) and generates the calling sequence to execute the microcode for that routine. This calling sequence consists of a trap instruction (illegal Nova instruction = #70000+routine offset) followed by the size of the stack frame for that routine. The calling sequence is also permanently changed as a side effect so that the microcode routine is henceforth called directly instead of going through the dummy interface routine again. This calling initialization scheme makes it possible for the control RAM to be loaded and efficient linkages to be set up completely automatically.

0 1	2	3	4	5	6	7	8	9	10	11	12 15
type	label	MD<-	Reg<-	MAR	L	T	cf	rf	<-MD	DISP	ALU Func

Word 1 of Instruction

type: **0** normal opcode instruction.
1 label declaration for conditional branching. Word 2 contains a conditional label, see figure 3.
2 register, variable name equivalence comment. **ALU Func** is the register number and word 2 of the instruction contains a pointer to the variable name.
3 formatting information, currently means insert CR in output file. **ALU Func** field contains size of this instruction.

label: **TRUE** if there is a label at the beginning of the following instruction. If **ALU Func** is zero then word 2 contains a pointer to the label, otherwise word 2 contains a conditional label, see figure 3.

MD←: Load Memory Data from BUS.

Reg←: Load Register from ALU output, see figure 2.

MAR: Load Memory Address Register from ALU output.

L: Load L register from ALU output.

T: Load T register from ALU output.

cf: **TRUE** if BUS source is a constant, constant in word 3.

rf: **TRUE** if BUS source is a register, see figure 2.

←MD: **TRUE** if BUS source is Memory Data.

DISP: **TRUE** if BUS source is Displacement field of IR.

ALU Func: Arithmetic Logic Unit (ALU) Function.

Figure 1.

0 1	2	3	4	5	6 9	10 15
shift	task	next	lab	fill	F2	Register Number

Word 2 of Normal Instruction

shift: **0** no shift.
1 left shift (LSH 1)
2 right shift (RSH 1)
3 swap bytes (LCY 8)

task: **TASK**.

next: **NEXT** address label is in word 3 (or 4 if constant) of instruction.

lab: Label type. If **lab** is **TRUE** then **NEXT** address label is a conditional label (see figure 3), otherwise it is a pointer to the label.

fill: not currently used.

F 2: **0** no activity. (see hardware manual for description of **F2** field)
1 BUS=0.
2 SH<0.
3 SH=0.
4 BUS.
5 ALUCY.
10 BUSODD.
other values not currently used.

Register Number: Register Number for bus source or destination.

Figure 2.

0 1	2 4	5 15
fill	type	number

Conditional Label Internal Format

fill: not currently used

type: **0** generate **COND#**, where **#** is the value of **number**.
1 generate label **IF#** as above.
2 generate label **WH#** as above.
all other values not currently used.

number: number to be appended to end of label being generated.

Figure 3.

APPENDIX A — Micro SPL Syntax

<program> ::= {<segment list>} **START**

<segment list> ::= <segment list> <segment> | <segment>

<segment> ::= <segment type> <id> {<heading>} {<dcl list>} <stmt list>

<segment type> ::= {**ENTRY**} <routine type>

<routine type> ::= **PROC | INT FUNC**

<heading> ::= **(**<arg dcl list>**)**

<arg dcl list> ::= <arg dcl list> , <arg dcl> | <arg dcl>

<arg dcl> ::= **REF** <dcl> | <dcl>

<dcl list> ::= <dcl list> <dcl> | <dcl>

<dcl> ::= **INT** <int list> | **INT ARRAY** <int array list>

<int list> ::= <int list>, <id> | <id>

<int array list> ::= <int array list> , <int array> | <int array>

<int array> ::= <id> {(<constant>)}

<stmt list> ::= <stmt list> <stmt> | <stmt>

<stmt> ::= <assign>
    | <if>
    | <while>
    | **EXIT**
    <return>

<assign> ::= <var> <assign op> <exp>

<var> ::= <id {(<exp>)}

<assign op> ::= := | ←

<if> ::= **IF** <exp> **THEN** <stmt list> {**ELSE** <stmt list>} **FI**

<while> ::= **WHILE** <exp> **DO** <stmt list> **OD**

<return> ::= **RETURN** {(<exp>)}

<exp> ::= <exp> <xor op> <equiv> | <equiv>

<equiv> ::= <equiv> <or op> <logical prod> | <logical prod>

<logical prod> ::= <logical prod> <and op> <relation> | <relation>

<relation> ::= <relation> <rel op> <add exp> | <add exp>

<mult exp> ::= <mult exp> <mult op> <factor> | <factor>

<factor> ::= <unary op> <primary> | <primary>

<primary> ::= <constant> | <var> | (<exp>)

<xor op> ::= **XOR | !**

<or op> ::= **OR | %**

<and op> ::= **AND | &**

<rel op> ::= **<> | ~= | = | < | <= | ~> | > | >= | ~<**
| NE | EQ | LS | LE | GR | GE | UGE | UGR | ULE | ULS

<add op> ::= **+ | -**

<mult op> ::= **LSH | RSH**

<unary op> ::= **~ | + | - | NOT**

<id> ::= <id> <letter> | <id> <digit> | <letter>

<constant> ::= <decimal num> | <octal num>

<decimal num> ::= <decimal num> <digit> | <digit>

<octal num> ::= **#**<octal digits>

<octal digits> ::= <octal digits> <octal digit> | <octal 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> ::= <octal digit> **| 8 | 9**

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

APPENDIX B — Reserved and Key Words

RESERVED WORDS

ABORT
ARRAY
CALL
CASE
CHAR
DEFINE
DO
ELSE
ESAC
EXIT
FI
FUNC
GET
IF
INT
OD
PROC
REAL
REF
RETURN
START
STRING
THEN
WHILE

KEY WORDS

AND
EQ
ENTRY
EXT
GE
GR
LE
LS
LSH
NE
NOT
OR
REM
RSH
UGE
UGR
ULE
ULS
XOR

APPENDIX C — Sample Micro SPL Program

```
ENTRY INT FUNC Delete(REF INT length, INT ARRAY heap)
  INT new, l2, root, son, cur, last, len, t

  len ← length

  IF len = 0 THEN RETURN(0) FI

  last ← heap(len)
  new ← heap(1)
  l2 ← (len-1) RSH 1
  length ← len - 1
  root ← 1
  son ← 2
  WHILE root <= l2 DO
    cur ← heap(son)
    t ← heap(son+1)
    IF t < cur THEN
      cur ← t
      son ← son + 1
    FI
    IF cur <= last THEN
      heap(root) ← cur
      root ← son
      son ← root LSH 1
    ELSE
      EXIT
    FI
  OD
  heap(root) ← last
  heap(len) ← #77777
RETURN(new)

ENTRY PROC Insert(REF INT length, INT ARRAY heap, INT new)
  INT l2, cur, t

  cur ← length + 1
  length ← cur
  l2 ← cur RSH 1
  t ← heap(l2)
  WHILE new < t DO
    heap(cur) ← t
    cur ← l2
    l2 ← cur RSH 1
    t ← heap(l2)
  OD
  heap(cur) ← new
RETURN

START
```

APPENDIX D — Microcode Generated for Sample Program

```
; 10-JUN-79 17:21:35
#AltoConsts23.mu;
#regdefs.;

!20,1,START ;
!37,1,TRAP1 ;
ret: L ← AC3+1,TASK ;
START: PC ← L ;                                   get return address
   SWMODE ;
   NOP, :START ;

!37,40,MicroInit,
   Delete,
   Insert ;

TRAP1: T ← 3 ;
   MAR ← L ← AC2+T ;
   T ← M-1 ;
   L ← MD+T, T ← MD ;
   SAD ← L, L ← T, TASK ;                         SAD now contains address of 2nd arg

   R44 ← L ;
   L ← AC0 ;
   R42 ← L ;
   L ← AC1 ;
   R43 ← L ;
   SINK ← DISP, TASK, BUS ;
   IR ← 0, :MicroInit ;

MicroInit:
   L ← AC0, TASK ;
   STATICS ← L ;
   L ← AC3+1, TASK, :START ;

; register R42 is used for length

; register R43 is used for heap

; register R36 is used for new

; register R35 is used for l2

; register R17 is used for root

; register R16 is used for son

; register R15 is used for cur

; register R14 is used for last

; register XREG is used for len

; register PC is used for t
```

```
Delete: MAR ← R42 ;
   NOP ;
   L ← MD, TASK ;
   XREG ← L ;

   L ← XREG, BUS=0, TASK ;

!1,2,IF1,IF2 ;

  NOP, :IF1 ;

IF2: L ← 0, TASK ;
  AC0 ← L, :ret ;

IF1: T ← XREG ;
   MAR ← L ← R43+T ;
   SAD ← L ;
   L ← MD, TASK ;
   R14 ← L ;

   MAR ← L ← R43+1 ;
   SAD ← L ;
   L ← MD, TASK ;
   R36 ← L ;

   L ← XREG-1, TASK ;
   AC0 ← L RSH 1 ;
   L ← AC0, TASK ;
   R35 ← L ;

   L ← XREG-1 ;
   MAR ← R42 ;
   NOP, TASK ;
   MD ← M ;

  L ← ONE, TASK ;
  R17 ← L ;

  L ← 2, TASK ;
  R16 ← L ;

WH1: T ← R35 ;
  L ← R17-T-1 ;
  NOP, SH<0, TASK ;

!1,2,WH2,WH3 ;

  NOP, :WH2 ;

WH3: T ← R16 ;
   MAR ← L ← R43+T ;
   SAD ← L ;
   L ← MD, TASK ;
   R15 ← L ;
```

```
      T ← R16+1 ;
      MAR ← L ← R43 + T ;
      SAD ← L ;
      L ← MD, TASK ;
      PC ← L ;

      T ← R15 ;
      L ← PC−T ;
      NOP, SH<0, TASK ;

   !1,2,IF3,IF4 ;

     NOP, :IF3 ;

   IF4: L ← PC, TASK ;
     R15 ← L ;

     L ← R16+1, TASK ;
     R16 ← L ;

   IF3: T ← R14 ;
      L ← R15−T−1 ;
      NOP, SH<0, TASK ;

   !1,2,IF5,IF6 ;

      NOP, :IF5 ;

   IF6: T ← R17 ;
      MAR ← R43+T ;
      NOP, TASK ;
      MD ← R15 ;

      L ← R16, TASK ;
      R17 ← L ;

      L ← AC0, TASK ;
      AC0 ← L LSH 1 ;
      L ← AC0, TASK ;
      R16 ← L, :IF7 ;

   IF5: NOP, :WH2 ;

   IF7: NOP, :WH1 ;

   WH2: T ← R17 ;
      MAR ← R43+T ;
      NOP, TASK ;
      MD ← R14 ;

      T ← XREG ;
      MAR ← R43+T ;
      NOP, TASK ;
      MD ← 77777 ;
```

```
   L ← R36, TASK ;
   AC0 ← L, :ret ;

; register R42 is used for length

; register R43 is used for heap

; register R44 is used for new

; register R36 is used for l2

; register R35 is used for cur

; register R17 is used for t

Insert: MAR ← R42 ;
   NOP ;
   L ← MD+1, TASK ;
   R35 ← L ;

   MAR ← R42 ;
   NOP, TASK ;
   MD ← R35 ;

  L ← R16, TASK ;
  R16 ← L RSH 1 ;
  L ← R16, TASK ;
   R36 ← L;

   T ← R36 ;
   MAR ← L ← R43+T ;
   SAD ← L ;
   L ← MD, TASK ;
   R17 ← L ;

WH4: T ← R17 ;
  L ← R44-T ;
  NOP, SH<0, TASK ;

!1,2,WH5,WH6 ;

  NOP, :WH5 ;

WH6: T ← R35 ;
  MAR ← R43+T ;
  NOP, TASK ;
  MD ← R17 ;

  L ← R36, TASK ;
  R35 ← L ;

  L ← R16, TASK ;
  R16 ← L RSH 1 ;
  L ← R16, TASK ;
  R36 ← L ;
```

```
  T ← R36 ;
  MAR ← L ← R43+T ;
  SAD ← L ;
  L ← MD, TASK ;
  R17 ← L, :WH4 ;

WH5: T ← R35 ;
  MAR ← R43+T ;
  NOP, TASK ;
  MD ← R44, :ret ;
```

APPENDIX E — BCPL Interface Generated for Sample Program

```
external RamImage
external LoadRam

static loadFlag = true

external Delete
external Insert

let Delete(arg1, arg2) = valof
[
  if loadFlag then MicroInit()
  Delete = table [ #70001 ; 10 ]

  resultis Delete(arg1, arg2)
]

and Insert(arg1, arg2, arg3) be
[
  if loadFlag then MicroInit()
  Insert = table [ #70002 ; 6 ]

  Insert(arg1, arg2, arg3)
]

and MicroInit() be
[
  loadRam(RamImage)
  loadFlag = false

  let t = table [ 0;  ]
  let Init = table [ #70000 ]
  Init(t)
]
```

APPENDIX F — BCPL Comparison Program

```
external Insert1
external Delete1

let Delete1(length,heap) = valof
[
  let len = @length
  if len eq 0 then resultis 0

  let last = heap!len
  let new = heap!1
  @length = len - 1
  let l2 = (len-1) rshift 1
  let root = 1
  let son = 2
  while root le l2 do
  [
    let cur = heap!son
    let t = heap!(son+1)
    if t ls cur then
    [
      cur = t
      son = son + 1
    ]
    test cur le last
    ifso
    [
      heap!root = cur
      root = son
      son = root lshift 1
    ]
    ifnot
      break
  ]
  heap!root = last
  heap!len = #77777

  resultis new
]

and Insert1(length,heap,new) be
[
  let cur = @length + 1
  @length = cur
  let l2 = cur rshift 1
  let t = heap!l2
  while new ls t do
  [
    heap!cur = t
    cur = l2
    l2 = cur rshift 1
    t = heap!l2
  ]
  heap!cur = new
]
```