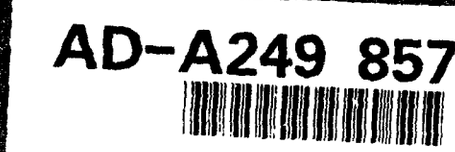


CONDITIONS OF RELEASE

0122002 309561

********** DRIC U

********** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

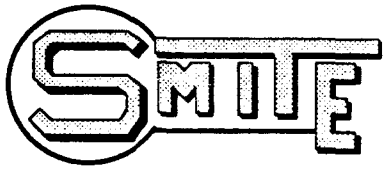
DEFENCE RESEARCH AGENCY

RSRE Memorandum 4568

Title: Applying Algebraic Techniques to Microcode Compilation

Author: C.J.Cant

Date: January 1992



ABSTRACT

This memo assesses the effectiveness of using algebraic techniques in the compilation of microcode. Microcode for an example system, the SMITE secure processor, is considered.

Contents

1. INTRODUCTION

The SMITE research programme is directed towards producing secure computing solutions. As part of this research, a processor for a secure workstation has been built [Field-Richards91]. The instruction set of this workstation is high-level language orientated and is based on that of the FLEX instruction set [Currie85]. The advantage of this architecture is that it uses capability addressing and this can be used as the basis of a scheme for fine grained protection of data [Wiseman89].

The firmware which supports the instruction set of SMITE/Flex is implemented in microcode. This document describes components of the SMAC (SMITE Microcode, Analysers and Compilers) system [Morrison89b] which was developed to aid the "correct" implementation of this microcode.

The development system was designed to meet three major requirements. Firstly, the problem of microcode readability was addressed. "Traditional" microcode is very difficult to read and hence understand. This lack of comprehension hinders effective maintenance, but more seriously in a secure environment, means that no confidence can be placed in the correctness of the microcode. The SMAC system, therefore, allows the use of high-level control constructs such as WHILE loops, IF statements and Procedures, whilst the view of storage is low-level and register based, though allocation of register addresses can be automated. Microcode written in this way is far easier to read and check, and this should help achieve a "correct" implementation.

Secondly, static analysis of certain properties of the microcode was required [Foster84]. The exact nature of the analysis was not known initially and so the development system had to be powerful enough to allow analyers to be added as and when required, with a minimum of effort. The ability to analyse properties of the microcode (which may include security relevant features) further aids a "correct" implementation.

Thirdly, the development system had to allow the microcode source to be as hardware independent as possible. This would allow the hardware platform to evolve and be optimised, without the need to change either the development system or completely re-write the microcode each time. The hardware independence of the development system means that should microcode be needed for subsequent hardware, it can be developed using the same development system.

This memo explains how an algebraic approach was employed to ensure that all three goals were satisfied in the SMAC system.

2. THE ALGEBRAIC APPROACH.

2.1. ALGEBRAS

An algebra signature consists of two major components. The first is a list of the sorts of objects that the system deals with. A sort is roughly (though not entirely) equivalent to a programming language type, however, it does not actually specify the values that can be taken. The sorts of an algebra define the kinds of objects that are to be manipulated. The list of sorts used in the SMAC system is given in the Appendix. The second part of the signature, is a list of constructor functions. These functions take a number of different sorts of objects as parameters and return one sort of object as their result. The constructors are essentially rules for describing how new objects can be built from existing objects. A list of the constructor functions in the SMAC system is given in the Appendix.

Some of the functions take no parameters and return a sort as a result. These are equivalent to (and listed in the Appendix as) constants of the particular sorts involved. In the SMAC system examples of these are the true and false conditions, "null_statement" etc.

Several implicit sorts, which correspond to a list of some sort of object, are also provided, along with constants for "empty list" and constructor functions to append elements to a list and concatenate lists.

2.2. COMPILING

To use the algebraic description in SMAC it is first necessary to instantiate the abstract sorts with some concrete programming types. This gives the actual values that can be manipulated. The constructor functions (and constants) are instantiated with concrete functions and constants of corresponding types.

There are of course many different ways of instantiating the sorts. However, a compiler can "encode" a program without needing to know the particular instantiation chosen. By choosing different instantiations of the sorts and constructor functions, the compiler can "encode" different information about the program.

Obviously recompiling the same program several times is not very efficient, therefore, an instantiation is chosen which encodes the structure of the program in terms of the algebras constructor functions. This data structure can then be used to call several different instantiations of the constructor functions to yield various different kinds of information about the program. These latter transformations are called "homomorphisms", since they preserve the structure of the object but may discard some information about the original.

3. TRANSFORMATIONS NECESSARY TO OBTAIN OBJECT CODE.

3.1. ADDRESS CALCULATION.

The SMAC system provides the programmer with the ability to use high-level control constructs, such as WHILE...DO...OD and IF...THEN...ELSE...FI, so there is not much use of explicit labels in a STAMP program. However, SMAC deals with labels and the compiler automatically generates all the labels, conditional and unconditional jumps that are required.

In order to generate object code, the addresses of each of the labels needs to be known. The initial sorts used by the compiler, encode the labels as abstract values which simply identify the label uniquely. What is required, is the relative displacement of each of the labels from the start of the code.

The sorts are instantiated so that the constructor functions, when called, produce a running total of how many microstore locations have been used. Functions such as "contents", "denote" and "invert" by themselves occupy no microstore locations, on the other hand a function that generates a statement occupies one location. Inside the block_statement function, labels of blocks are matched with the relevant offsets. Labels imported from other modules are dealt with differently, their values are absolute having been calculated previously.

The final value returned by the transformation is a first-class procedure [Stanley86]. This first-class procedure takes as a parameter the start address for the piece of code in question (calculated by a memory allocation procedure). The procedure adds this start address to the list of relative addresses, to give the absolute addresses for the labels. It is possible to generate relocatable code, and then allocate the addresses on code generation. However, to keep the SMAC system as simple as possible the addresses are fixed at this stage.

This use of first class procedures is a feature of Perq Flex Algol68 [Stanley86]. In any other implementation the list of relative addresses would have to be returned as a data structure and the absolute addresses calculated from it.

It is important to realise, that the calculation of addresses of labels within a piece of code, must be done after *all* the optimisations have been applied otherwise chaos will result.

3.2. TRANSLATION INTO STEP LANGUAGE.

This transformation takes the encoding produced by the compiler and generates a series of steps which represent the instruction steps. Each of the steps is made up of a series of sub-steps that detail the information flows that constitute the instruction. The transformation again takes the compiler generated encoding of the program and extracts information from it to produce a totally different representation of the program.

This transformation does the work of substituting abstract names for real locations. For example, the compiler generated abstract labels are replaced with real microstore addresses, using the results of the previous transformation. The real locations of registers are substituted for the names by which they were referred to in the source code. Unfortunately, declarations of store locations are wrapped around their scope. So when the sorts that represent the scope of the declaration are being processed, the information regarding the locations of the registers is not present. To maintain the context independence of the transformation, a first-class procedure is returned. This will carry out the required substitutions when the information regarding register locations is passed to it as a parameter.

The program is thus reduced to a series of reads, writes, operators and operands. From this "step language" it is possible to produce object code (given a description of the machine written in the machine description language Cudgel [Morrison89b]).

4. OPTIONAL TRANSFORMATIONS

4.1. REDUNDANT BRANCH REMOVER

The compiler is not very efficient in the way in which it encodes the program in the algebra. As a result there will be several places in the code where branches jump to instructions which are in turn unconditional branches. The redundant branch remover transformation modifies a program by eliminating redundant branches, but otherwise preserves the programs meaning. For instance :

```
   GOTO label1;                        GOTO label2;
   ...                  becomes...     .....
label1:                                label2:
   GOTO label2;                        .....
   ...
label2:
     ...
```

In this example, the block at "label1" has been removed (reducing the amount of space the code occupies) and the jump has been redirected (reducing the code execution time). In the above example, it is assumed that there is some intervening code between the branch and the block "label2". This will not necessarily be the case and it may be possible to remove the branch altogether.

The transformation basically reproduces the original program, except when a branch is encountered. A list of labels and their replacements is maintained. As labels are encountered and substitutions made, entries are added to the list. When a branch is encountered, this list is checked to see if the target label has been encountered yet. It is not possible to substitute the label if the target has not been encountered. Once the target label has been encountered the substitution is possible. To cope with a branch to a branch to a branch and so on, a transitive closure is performed on the list. If a substitution is not possible, all the information needed to perform the substitution at a later date, and the intervening sorts, need to be remembered. A first class-procedure is returned with all the necessary information bound into it. This first class procedure takes as a parameter the list of labels and their substitutes. When the procedure is called with the list, the optimised code will be returned only if the list contains sufficient information to do the reduction, otherwise a first class procedure is returned. The final result of the transformation is a fully reduced program.

4.2. ALGEBRA FLATTENER

The compiler encodes the program in a highly structured way and this makes the optimisation of certain constructs difficult. It was found to be a great deal easier to implement the transformation (4.3) below, if the program was encoded without redundantly nested block statements. This transformation simply extracts all the constituent statements and blocks into a simple list and then wraps them up into a block statement.

Consider the following code scrap :

```
reg a:0, b:1, c:2, d:3;
a := a + 5;
b := b + a;
IF a < b
THEN
  a := a + 5;
  b := b + 7
ELSE
  c := c + 5;
  d := d + 4
FI;
```

When this has been compiled and optimised using the redundant branch remover, the program structure is as listing 1. The BEGIN...END represents a "block statement". Empty blocks and statements are represented by "NO_BLOCKS" and "NO_STATEMENTS". Listing 2 shows the flattened version. This is a simple list of two statements and five blocks, the last of which contains no statements.

```
BEGIN
   a := a + 5;
   b := b + a;
   BEGIN
      BEGIN
         NO_STATEMENTS
start:   IF NOT a >= b GOTO then
not:     GOTO else
      END
then: BEGIN
         a := a + 5;
         b := b + 7
      END
      GOTO fi
else: BEGIN
         c := c + 5;
         d := d + 4
         NO_BLOCKS
      END
fi:   BEGIN
         NO_STATEMENTS
         NO_BLOCKS
      END
   END
   NO_BLOCKS
END
```

Listing 1

```
BEGIN
   a := a + 5;
   b := b + a;
start: IF NOT a >= b GOTO then
not:   GOTO else
then:  a := a + 5;
       b := b + 7
       GOTO fi
else:  c := c + 5;
       d := d + 4
fi: NO_STATEMENTS
END
```

Listing 2

4.3. CONDITIONAL BRANCH OPTIMISER

The compiler (as has been said before) is rather inefficient in the way that it encodes certain structures. It is often the case that you are left, after previous optimisation stages, with the following:

```
IF NOT condition GOTO then_clause
GOTO else_clause
then_clause:
   Then code here
```

Instead of the more obvious :

```
IF condition GOTO else_clause
Then code here
```

which removes a superfluous GOTO.

Using the specially flattened version of the algebra, the transformation proceeds to copy the algebra until a conditional branch is met. If possible, the above optimisation is carried out, and the copying resumes. This reduces the amount of space that the code occupies - since many, if not all the conditional jumps are compiled as above. This also increases the speed of the code.

Taking the example from above in 4) the result of the transform gives:

```
  a := a + 5;
  b := b + a;
start:
  IF a >= b GOTO else
then:
  a := a + 5;
  b := b + 7
  GOTO fi;
else:
  c := c + 5;
  d := d + 4
fi:
  NO_STATEMENTS
```

An enhancement to this transformation would be to optimise further within CASE statements. It can happen that there are two consecutive unconditional branches in the same block (the second one is never executed). However, it is difficult to know when inside the CASE statement construct you can safely remove these consecutive branches and when you cannot. As the frequency of CASE statements within the microcode is not high, the effort that would need to be expended does not justify the results.

4.4. CONSTANT EXPRESSION EVALUATION

To enable microcode to be written in as clear a fashion as possible, the SMAC system allows expressions such as :

```
CONST step = 4, initial_offset = 9;
      offset := offset + ( step * initial_offset );
```

The above expression can not be executed in one microinstruction cycle because it consists of two arithmetic operations. It is possible however, to reduce the expression by evaluation at compile time to :

```
offset := offset + 36;
```

The transformation copies the original program and looks for an operation (an operator and one or more operands) which has operands that are all constants. An attempt is then made to evaluate the expression. The expressions are evaluated using Algol68 procedures supplied to the transformation as parameters. If the necessary Algol68 procedure is not supplied, the program scrap is simply reproduced, otherwise, the result of the expression (a constant) is substituted instead.

5. FUTURE ENHANCEMENTS

The number and type of transformations that can be applied to a program is large. However, the more optimisations you chose to use, the greater the resources needed to execute them (memory, disc space, time!).

An additional transformation that would have made the microcode easier to write would have been register allocation. When declaring a register, it is possible to allocate it a location in a register file explicitly. However, it is a difficult task for the programmer to keep track of which register locations have been used and which have not. This task is made all the more difficult, when several modules are involved and sub-routines may be nested to several levels deep. From an analysis of the information flow within the program it would be possible to allocate registers to locations automatically. This would be very worthwhile having, but is a highly complex analysis and transformation which would be very difficult and time consuming to implement. The application of the flattener transformation simplifies the problem by removing the recursive structure of the program, however, problems still remain about how to analyse loops, procedure calls and conditional jumps.

Experience of writing the SMITE microcode suggests that the maximum depth of procedure calls is 3-4, which means that it is possible manually to keep track of which registers have been allocated and which have not. The need for temporary locations is small, with a few exceptions, most of the registers are used globally and have static usage.

6. THE EFFECTIVENESS OF THE ALGEBRAIC APPROACH.

6.1. CONTEXT INDEPENDENCE

A major advantage of using homomorphisms is that they are context independent, it doesn't matter where a program fragment is used, it will always be transformed in the same way. A result of this context independence is that objects only need to be transformed once, the value can then be reused when the same object is encountered again. In the SMITE system, the size of sub-trees that actually lend themselves to being manipulated in this way, are quite small and so any advantage gained by not recalculating values would be overwhelmed by the time taken to decide whether the value had been calculated previously. The algebra is encoded as a sequence of bytes and so every time a reference to, say a label "fred", is made, the information is encoded again.

Not recalculating values would be essential if the sub-trees were larger, in the case where the system supported "macro-instructions", a piece of predefined code could be referred to at several points in the code. Here, considerable time would be saved by only calculating these sorts once.

6.2. OVERHEADS

The implementation is very wasteful in terms of the way it manipulates the information it uses. The major pieces of information are lists which are implemented as vectors (arrays). As a result a great deal of the operations involve copying of vectors as new elements are added. This means that there is a large amount of garbage collection going on, which slows the program down. With longer modules this gets excessive and the Perq can actually run out of mainstore because the amount of information being held in mainstore, in addition to the algebra, becomes too great for the computer to handle.

Execution speed would be improved if instead of the information being stored in vectors, linked lists were used for the larger structures. This would increase the program complexity, as linked lists are slightly more difficult to program but would mean that the garbage collector would not be forced to work as hard. Only the space required would be taken by the program as and when required and there would be no need to copy the whole structure when a new element was added.

The result of each transformation is written to disc. Keeping these intermediate results from the various homomorphisms in main store and not writing them to disc would not only speed up execution of the program but would also ensure that the disc would not need to be garbage collected as often. The only possible problem with keeping intermediate results in mainstore would be the shortage of memory.

Applying two transformations in parallel would be one solution to the storing of intermediate results, but this would increase programming complexity in the transforms and would be difficult to implement as many of the transforms rely on previous ones being finished. The constant expression evaluator could certainly be done in parallel with one of the other optimisers as this is one of the simplest of the transformations.

A large amount of time is taken to load the machine description off disc. The structure which represents the machine is large, largely due to the need in CUDGEL to explicitly state several instances of the same operation. Loading the machine description only once when dealing with several modules is essential to the speed of operation.

The largest amount of time involved in the code generation process takes the step language and attempts to map it onto the target hardware, to produce a list of fields and values. This can involve several attempts at the same instruction before a suitable configuration of data paths and registers accesses is found. No attempt has been made to optimise or improve this code.

6.3. LIFE WITHOUT FLEX FIRST CLASS PROCEDURES.

The implementation of these transformations on Flex makes use of first-class procedures implemented as closures [Stanley86]. This is very useful if the calculation of a sort is context dependant. For example in the case of the optimisation homomorphism it is not possible to calculate the sort of a branch until you know something about the target of the branch. To preserve the context independence, and hence make the homomorphism work, the information necessary to calculate the sort is bound into a first class procedure as its non-local data. In other systems it is not possible to use first class procedures in the same way. This means that some other method of applying the transformation to the algebra would be needed.

A possible solution would be to use a tree structure to represent the algebra in mainstore, each instance of a signature function would be a node of the tree, with the parameters of the signature functions as nodes below it.

Consider a sample statement :

```
total := total + total;
```

This could be represented in a tree structure shown in Figure 1.

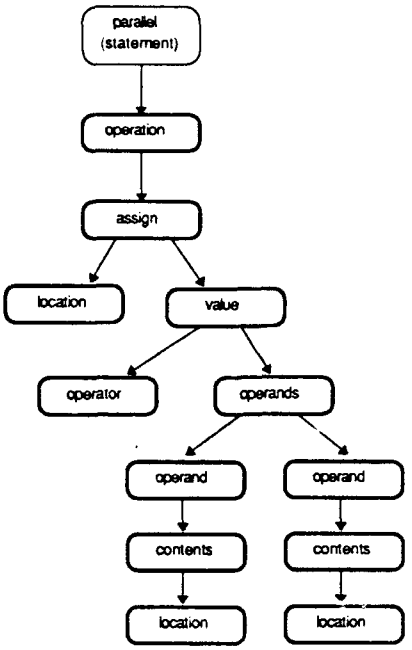


Figure 1: Tree representation of sample statement

Reading from the top of the tree the statement is : a single "operation" that "assigns" a "value" that is the result of an "operator" acting on two "operands" into a "location". The two "operands" are both the "contents" of a "location".

You can see that this particular representation does not reuse sub-trees, hence the two operand sub trees refer to the same storage location. They could therefore be represented as the same node.

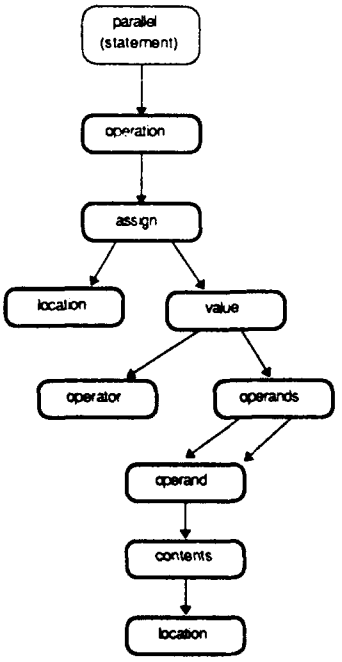


Figure 2: Tree representation reusing sub-trees

Figure 2 shows the above example, but reusing the "location" sub-tree. Clearly this is a trivial example, a better one would be to say that should the instruction be repeated somewhere else in the program, then the tree structure would use this sub-tree again, instead of generating a new one.

If sub-trees are reused, the amount of storage used is reduced (though if the size of the sub-trees that are actually re-used is small this figure is not significant). The main problem is how to decide efficiently whether the sub-tree has been calculated before. Clearly if the only sub-trees that are reused are very small, then the amount of time deciding whether every sub-tree about to be calculated has been met before, is going to far outweigh any advantage gained by not recalculating the sub-tree.

Once the algebra has been represented as a tree structure, the problem is how to apply the required transformations to the algebra. Assuming that there is no reusing of sub-trees, all the transformations can be applied to this structure one after the other, with no need to save intermediate results on disc.

The "compulsory" transformations extract information from the structure to use at a later date. These transformations simply have to traverse the tree in the correct manner to extract the necessary information.

The optimisation transformations involve manipulating the tree, which can be done on the original structure with no need to copy vast amounts of data. These operations are mostly on pointer data and so can be performed very quickly. For example, to remove a sub-tree which represents an unwanted branch, simply remove the pointer to that sub-tree. To alter the destination of a branch, overwrite the label information with the new destination.

There is very little recalculation of objects — new branches being an exception. Once all transformations have been completed, the resulting algebra can be encoded and placed on disc. The results of the translations can then be used to generate object code for the target hardware.

So in this scheme, the vast majority of objects are only processed once (at compile time) and the subsequent transformations do the minimum amount of work needed. There is very little copying of information, no unnecessary disc transfers and the use of a tree structure means that storage usage is very efficient. The method does have the drawback of being slightly harder to implement, dealing as it does with linked lists and trees.

6.4. SCALING

As the implementation stands, there are overheads associated with each of the transformations, as has been said before. These overheads vary in direct proportion to the size of the module of code. The larger the code segment, the longer it takes to transfer to and from disc. The time taken to calculate each of the transformations also takes longer as the module size increases. The complexity of the code has a bearing on the time taken, especially for the optimisation transformations, though the increase in execution time appears to increase roughly with increasing code size as a linear function.

It should also be noted, that the speed of operation is very much dependant on the Flex garbage collector. The amount of time spent garbage collecting can be excessive and in fact mainstore may be exhausted by the program. This makes an assessment of the performance of the program very difficult. In this implementation there is a trade off to be made, between the speed of execution and the amount of information that can be stored in mainstore.

6.5. THE "CORRECT" COMPILER.

The transformations involved in turning high level source code into very low level microcode are very complicated. Whilst a specification of a transformation could be short and simple, the implementation would be very complex to program.

From a point of view of "correctness" the inherent type checking of high level languages is an aid in both implementing the transformations and ensuring what they do is correct. Hence it is impossible to transpose a "label" and a "location" simply because they are of different types.

The complexity of a transformation is influenced by the structure of the algebra. The algebra in the SMAC system stresses the recursive "block" nature of the high level source code. Some transformations are easier to perform on a "flat" algebra and so it was necessary to write an algebra flattening homomorphism. Clearly the choice of the correct algebra is crucial in simplifying and reducing the transformations. The simpler the transformations, the more confidence can be placed in the compiler.

The use of algebraic techniques allows a direct mathematical link to be made between the source code and the final compiler output. The algebraic approach makes it possible to analyse the code for errors and inconsistencies. Optimisations can be performed easily to obtain efficient code. All these features add to the confidence that can be placed in the compiler.

7. CONCLUSIONS

This report describes the use of algebraic techniques in the production of microcode for some target hardware. The SMAC system, as it was applied to the production of microcode for the SMITE/Flex processor, has been used as an example. The ability to write microcode in a relatively high-level form is a great advantage. Source code is much easier to read and understand, which in turn aids the process of debugging and maintenance.

The higher level nature of the microcode language enables the "hard-work" of microcode development to be done automatically, relieving the programmer of much of the stress of this type of programming. Using homomorphisms to implement parts of the compilation system increases confidence that the object code is a "correct" compilation of the source code. However, the algebraic approach would be better exploited if "code reuse" were supported at the source level, which would lead to much more efficient execution of the transformations.

The SMAC system would benefit from further work to change the algebra slightly, so that the implementation of the more complex transformations would be simplified, allowing more confidence to be placed in their correctness.

8. REFERENCES

[Currie85] **Perq Flex Firmware**
I.F. Currie, J.M. Foster, P.W.Edwards.
RSRE Report No. 85015, December 1985.

[Field-Richards91] **The STAB II Secure Processor Main CPU Hardware**
H.S.Field-Richards
RSRE Memorandum No. 4496, July 1991

[Foster84] **Checking Microcode Algebraically**
J.M. Foster
RSRE Memorandum No. 3748, October 1984.

[Foster85] **Regular Expression Analysis of Procedures and Exceptions**
J.M.Foster
RSRE Report No. 85008, June 1985.

[Morrison89a] **The STAMP micro-programming language**
J.B.Morrison
RSRE Memorandum No. 4317,September 1989.

[Morrison89b] **The SMAC microcode system**
J.B.Morrison
RSRE Memorandum No. 4322, September 1989

[Stanley86] **Using True Procedure Values in a Programming Support Environment**
M. Stanley
RSRE Memorandum No. 3916, February 1986.

[Wiseman89] **Basic Mechanisms for Computer Security**
S.R.Wiseman
RSRE Report No. 89024, January 1990

9. APPENDIX A

9.1. SMAC SORTS

Class	Name	Label
Operator	Seq_op	Cond_op
Store	Value	Condition
Operation	Statement	Block
CaseElem	Range	Import

In addition to the above, there are the plurals : Stores, Values, Operations, Statements, Blocks, CaseElems, Ranges.

9.2. SMAC SIGNATURE FUNCTIONS.

introduce_op
Int, Str -> Operator

An operator is constructed from an integer and a string. The integer is the "operator number", taken from the Context, which is used by the code generator to invoke the appropriate hardware function. The string is used for informational purposes and is the textual name of the operator.

introduce_seq_op
Int, Str -> Seq_Op

A sequencer operator is constructed from an integer and a string. The integer is the "sequencer operator number", taken from the Context, which is used by the code generator to invoke the appropriate hardware function. The string is used for informational purposes and is the textual name of the operator.

introduce_class
Int, Int, Int, Int, Str -> Class

Defines a class of registers. The integer parameters specify the start address of the class, the end address of the class, and the operator numbers to read and write from the class of register. The string parameter is the textual name.

introduce_cond_op
Int, Int, Int, Int, Int, Int, Str -> Cond_op

Defines a conditional operator, such as <, <=, =. The integer parameters designate the operator numbers necessary to perform the test (see below), and the conditional operator number. The string in the textual name.

introduce_test_op
Int, Int, Str -> Cond_op

Defines the test operators used above. The integer parameters define the operator numbers and the string parameter is the textual name.

make_import
Module -> Import

Modules are imported using the USE command. This function makes each module encountered an import so that the information it keeps can be imported.

introduce_name
Int, Str, Place -> Name

Constants and Variables (registers) are defined using declare constant and declare store functions respectively. This function is used to define a unique identifier for the object being declared. The integer parameter is the "magic" identifying number, the place is the position of the object in the source text and the string is the textual name.

introduce_label
Int, Str, Place -> Label

All labels have to be introduced before they are used (to make forward jumps easier to handle). This function is used to define labels used within this module. For parameter details see above.

use_label
Import, Int -> Label

Introduces a label defined in another module. The import parameter is the other module, and the integer is the= magic number.

use_constant
Import, Int -> Value

Introduces a Value defined in another module. Constants are imported as a single value, but registers are imported as two values using the use_constant function. There is no use_register in the algebra.

compute_label
Value, Labels -> Label

Given a list of labels and a value indexing into them, select the correct label from the list. When generating a CASE statement, the compiler generates a list of labels to branch to which select the various options on the CASE. An expression which returns a value indexes this list of labels.

null_operation
Constant of type Operation

Used to define an null operation - used for concatenation of operations.

no_label
Constant of type Label

Used to define an null label - used for concatenation of operations.

not_a_store
Constant of type Store

Used to denote an undefined store.

not_a_value
Constant of type Value

Used to denote an undefined value - for instance the initial value of a register when it is declared and not initialised.

null_statement
Constant of type Statement

Used to denote an undefined or null statement

false_condition
true_condition
Constants of type Condition

Used to denote true and false conditions for tests.

denote
LongInt -> Value

This makes a tagged value out of a 32' integer constant. For instance when a register is assigned the value of an integer constant in the source text.

```
a := 120374;
```

contents
Store -> Value

Read the contents of a store and produce a value.

extend
Values -> Value

Produce an extended precision value from a number of values. Not used very much if at all.

declare_store
Name, Value, Value, Value, Class, Statement -> Statement

A store Name of type Class is declared. The three values supplied are start address, end address and initial value (optional). The supplied statement represents the scope of the declaration.

declare_constant
Name, Value, Statement -> Statement

A constant Name of Value is declared. The statement supplied is as above.

constant_value
Name -> Value

Supply the value of a Named and predefined constant.

location
Name -> Store

Supply the storage location of a Named register, so that it can be accessed.

index
Store, Value -> Store

Index into an array of store. In STAMP it is possible to declare a store in terms of a start address and its size (default size of one). Hence

```
reg list_of_pointers:10[8];
```

defines a register "list_of_pointers", whose start address is 10 and whose size is 8 locations. Accessing this location is done thus :

```
new_ptr := list_of_pointers[8]
```

The function of index is to take the supplied index and calculate the actual store being referred to.

operate
Operator, Values -> Value

Apply an operator to arguments to produce a resultant value. The values can be constants, or the contents of a register location.

store_start
Store -> Value

Find the start address of the store given.

assign
Stores, Value -> Operation

Assign a value to a group of stores. This is the general way that values are assigned to register locations (with stores being one store). However, it is possible to assign a value to a group of stores, an array for example.

parallel
Operations -> Statement

Depending on the architecture of the target CPU and the operations required, it may be possible to do more than one operation in the same microinstruction cycle. For instance, it may be possible to assign to a register and jump at the same time.
Normally the number of operations is one, these operations are converted to a statement by this function.

address
Label -> Value

This function determines the microstore address of the supplied label. This is useful for doing scruffy things with the interrupts and other sequencer things.

test
Cond_op, Values -> Condition

Apply a conditional operator to a number of values (usually two) to produce a condition. For instance, a statement such as :

```
IF a <= b
THEN
  a := a + 5
FI;
```

would use the values produced by reading registers a and b and the conditional operator "<=" to produce the required condition.

invert
Condition -> Condition

Used to implement the NOT function for conditions.

sequence
Condition, Seq_op, Label -> Operation

blind sequence
Condition, Seq_op -> Operation

These two functions can be considered together. Both of them are used to generate conditional sequencer operations. The difference between them is that the **sequence** function invokes sequencer operations that use a label, for instance conditional GOTOs and CALLs. An unconditional sequencer function is implemented by supplying the constant true_condition (defined earlier), this is most often used in unconditional GOTOs.

CASE statements

```
CASE option REG case_temp
  WHEN 0; a := a + 45
  WHEN 1; b := b + 45
  WHEN 2; c := c + 45
ESAC
```

Each of the clauses has a label associated with it. Each of the WHEN statements generates a Range of values (in the above case the range is 1). The ranges are then associated with the correct labels to form individual case elements. Finally a CASE statement is constructed. A list of branches is constructed which corresponds to every value of "option" specified. The correct clause is selected by branching to an address which is the result of adding the value in "option" to the address of the first of the list of branches (in this case the intermediate result is stored in a register called "case_temp").

make_range
Value, Value -> Range

Designate a range between two values for CASE statements.

no_range
Constant of type Ranges, used to concatenation individual ranges.

make_case_elem
Ranges, Label -> CaseElem

Produce a CASE element from a number of ranges and a label which specifies the target of the branch.

make_case
Value, CaseElems, Seq_op, Store -> Statement

Produce a CASE statement from constituent parts.

value_place
Value, Place -> Value
operation_place
Operation, Place -> Operation
statement_place
Statement, Place -> Statement

The above take an already calculated sort and wrap place information around it. The place information is supplied from the compilers lexical analyser. If an error should occur, the place information is used to mark the position of the offending item.

make_block
Label, Statements -> Block

A label followed by some statements forms a block.

block_statement
Statements, Blocks -> Statement

A set of statements, followed by some blocks represents a block statement. This is the basic building block of the program. It is used recursively to define the program structure.

keep_label
Int, Str, Statement -> Statement
keep_store
Int, Int, Str, Statement -> Statement
keep_value
Value, Int, Statement -> Statement
keep_constant
Int, Str, Statement -> Statement

These functions are used to place declarations in the keep list, so that they can be imported into other modules. The Int parameters are the magic numbers used to refer to the items, the Str parameters hold the name in characters. The Value parameter is the sort that produced the value, for instance a field definition. The Statement parameters represent the program scope.

REPORT DOCUMENTATION PAGE

DRIC Reference Number (If known) .....

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification, eg (R), (C) or (S).

Originators Reference/Report No.	Month	Year
MEMO 4568	JANUARY	1992

Originators Name and Location
RSRE, ST ANDREWS ROAD
MALVERN, WORCS WR14 3PS

Monitoring Agency Name and Location

Title
APPLYING ALGEBRAIC TECHNIQUES TO MICROCODE COMPILATION

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (In the case of translations)

Conference Details

Agency Reference	Contract Number and Period

Project Number	Other References

Authors	Pagination and Ref
CANT, C J	17

Abstract

This memo assesses the effectiveness of using algebraic techniques in the compilation of microcode. Microcode for an example system, the SMITE secure processor, is considered.

Abstract Classification (U, R, C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)
UNLIMITED