AN IMPLEMENTATION OF
SEAL ON MULTICS

by

PAUL ADRIAN GREEN II

Submitted in Partial Fulfillment
of the Requirements for the
Degree of Bachelor of Science
at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1973 i.e. February, 1974

Signature of Author  Paul Adrian Green II
Department of Electrical Engineering, May 22, 1973

Certified by  Michael A. Schroeder
Thesis Supervisor

Accepted by  David Adler
Chairman, Departmental Committee on Theses
AN IMPLEMENTATION OF
SEAL ON MULTICS

by

PAUL ADRIAN GREEN II

Thesis Supervisor: Professor Michael D. Schroeder.
Title: Assistant Professor of Electrical Engineering.

ABSTRACT

This thesis describes the implementation of a
code generator for the Seal language on the
Multiplexed Information and Computing Service. The implementation developed extensive
error handling techniques for both the code
generator itself, and the Seal programs it compiles.

Work reported herein was sponsored by Project MAC and Honeywell
Information Systems, Inc., and by the Advanced Research Projects
Agency of the Department of Defense.
An Implementation of Seal on Multics.

Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table of Contents.</td>
<td>3</td>
</tr>
<tr>
<td>Acknowledgements.</td>
<td>4</td>
</tr>
<tr>
<td>I. Introduction.</td>
<td>5</td>
</tr>
<tr>
<td>I.1 Outline of the Thesis.</td>
<td>5</td>
</tr>
<tr>
<td>II. Error Controlling Methods.</td>
<td>10</td>
</tr>
<tr>
<td>II.1 Why Control Errors?</td>
<td>10</td>
</tr>
<tr>
<td>II.2 Language Design.</td>
<td>11</td>
</tr>
<tr>
<td>II.3 Code Generator Design.</td>
<td>14</td>
</tr>
<tr>
<td>II.4 Code Generator Implementation.</td>
<td>17</td>
</tr>
<tr>
<td>II.4.1 Modularity.</td>
<td>17</td>
</tr>
<tr>
<td>II.4.2 Simplicity.</td>
<td>18</td>
</tr>
<tr>
<td>II.4.3 Limiting Optimizations.</td>
<td>19</td>
</tr>
<tr>
<td>II.4.4 Self-checking.</td>
<td>24</td>
</tr>
<tr>
<td>II.4.5 Programming Style.</td>
<td>25</td>
</tr>
<tr>
<td>II.5 Detection of User Program Errors.</td>
<td>26</td>
</tr>
<tr>
<td>II.5.1 Compile Time Checks.</td>
<td>27</td>
</tr>
<tr>
<td>II.5.2 Execution Time Checks.</td>
<td>28</td>
</tr>
<tr>
<td>III. Summary and Suggestions for Further Work.</td>
<td>33</td>
</tr>
<tr>
<td>III.1 Summary.</td>
<td>33</td>
</tr>
<tr>
<td>III.2 Problems and Suggestions.</td>
<td>34</td>
</tr>
<tr>
<td>III.3 Suggestions for Hardware Extensions.</td>
<td>34</td>
</tr>
<tr>
<td>III.4 Further Work.</td>
<td>37</td>
</tr>
<tr>
<td>Appendix A. Rules for Formatting PL/I Programs.</td>
<td>39</td>
</tr>
<tr>
<td>Appendix B. Multics Seal Code Generator Program.</td>
<td>42</td>
</tr>
<tr>
<td>Appendix C. Multics Seal Code Generator Table.</td>
<td>100</td>
</tr>
<tr>
<td>Bibliography.</td>
<td>147</td>
</tr>
</tbody>
</table>
Acknowledgements.

It is difficult to properly give credit to all of the people who have patiently taught me what I know. Those mentioned here stand out at the head of a very long list; to all of them I owe a sincere appreciation.

To Professor Michael D. Schroeder, my thesis advisor, for bringing this document from its initial rough stages.

To Mr. Robert A. Freiburghouse, for his perspicuity in the myriad technical areas of the Seal language.

To Professors F. J. Corbató and J. H. Saltzer, and the entire Computer Systems Research Division of Project MAC, for bringing together a lot of talented, dedicated people to work on Multics.

To all of the people at the Cambridge Information Systems Laboratory of Honeywell, with whom I learned a great deal about Multics and its PL/I compilers.

To my father, for suggesting that I get into this field.

This document was prepared on Multics using the QEDX text editor and the RUNOFF text formatter. It was printed on an IBM 2741 Selectric console, using an 015 typewriter. The appendices were printed on a Honeywell PRT-300 line printer.
Part I: Introduction.

Section I.1: Outline of the Thesis.

This thesis is concerned with the implementation of a code generator for a language which performs extensive error checking. The language implemented is Seal, (1) and the implementation was performed on the Multiplexed Information and Computing Service (Multics), (2) a prototype computer utility developed jointly by MIT and Honeywell.

Sections of the thesis discuss how the goal of controlling errors influenced the design and implementation of the Multics Seal compiler. The compiler is a simple, two-pass program, with an optional optimizing pass. While the first pass (the parse) and the internal representation were completely designed and implemented before this thesis was begun, they are compatible with this thesis. This thesis influenced the design and implementation of the Multics Seal code generator and runtime support routines.

(1) "A Language for Virtual Memory Systems," by R. A. Freiburghouse, Honeywell, Inc., to be published.

An Implementation of Seal on Multics.

Consideration is given to errors arising in the compiler itself, accidental errors in the user's Seal program, and deliberately malicious Seal programs. A compiler might accidentally generate incorrect code due to internal errors, or a user might accidentally use an undefined variable, and occasionally users try to tamper with the underlying support mechanisms. The thesis proposes error handling mechanisms which are simple extensions of the base protection mechanisms provided by the Multics operating system and associated hardware (the Honeywell 6180 processor). A major conclusion of the thesis is that a very small number of mechanisms are needed to effectively control errors, if they are used methodically.

The error handling mechanisms proposed are justified on the same basis as general access controlling facilities. The desire to shield the user from accidental programming errors, the desire to make the over-all system as well-defined as possible, and the desire to control malicious users are all relevant. Earlier attempts to provide mechanisms for access control often gave little more real protection than a smoke screen; knowledgeable programmers could always find a loophole. Today most system designers realize that effective protection requires that every reference be validated, and that assumptions of ignorance or secrecy are inadequate to solve the problem. Recent more emphasis has
An Implementation of Seal on Multics.

been placed on designing appropriate hardware mechanisms for implementing protection mechanisms. The Honeywell 6180 hardware is the direct result of research into such mechanisms. (1) Hardware assistance assures that every reference is validated, and it also minimizes the overhead associated with the validation process. This thesis proposes similar mechanisms for error control.

This thesis describes error controlling mechanisms which have been implemented with a combination of hardware and software support. Some of the mechanisms could benefit from additional hardware capabilities, but none of them were impossible or prohibitively expensive on the H6180 machine. A comparison of the Seal implementation to the Multics PL/I implementation shows that Seal is no slower than PL/I for execution of similar programs, when performing similar error checking. While obtaining execution speed comparable to PL/I was not a direct object of this thesis, several early error handling designs were discarded or modified due to their excessive cost.

The Seal code generator is a simple, table-driven program. It accepts the output of the Seal parser and generates Multics standard object programs. At the present time, the code genera-

An Implementation of Seal on Multics.

tor correctly implements a large subset of the Seal language.

This thesis concerns the development of:

1. A code generator for Seal on Multics.
2. Methods for controlling the design and implementation errors which would lead to compiler bugs.
3. Methods for catching all language violations, either at compile time or execution time.

This partial implementation has demonstrated that the methods chosen by this thesis are adequate. A final, complete implementation at a more leisurely schedule is planned.

Section I.2: A Brief Description of Seal.

Seal stands for "simple extensible algorithmic language." Seal is derived from Algol 68, Euler, and list processing languages. It contains very simple, but generalized facilities for constructing and manipulating data structures either as local values or as permanent, global values, residing in a hierarchically structured, segmented, virtual memory. Because the global data structures created by a Seal program are operated upon exactly as if they were the values of local variables, no complicated I/O language is needed.

The language is designed to facilitate construction of sound, well-structured programs built from collections of sepa-
An Implementation of Seal on Multics.

rately compiled procedures, each of which may contain nested pro-
cedures. All procedures are potentially recursive and use a
stack model of storage for their local variables.

Seal allows the programmer to build up an infinite set of
data types (modes) from a set of seven basic data types. Com-
plete data type checking is always performed, normally at compile
time, but dynamic type checking is provided for variables
declared to possess values of unrestricted mode. Procedures can
be invoked as functions or as infix or prefix operators. The
ability to create new data types and to define procedures which
can be invoked as operators provides limited language extensibili-
ity without loss of program readability.

The design objectives of Seal which influenced this thesis
are:

1. It must be compilable into non-interpreted code, but
   it must allow some variables to possess values of unre-
   stricted mode (called "any mode" by Seal).
2. It must provide for separate compilation of program
   modules and must have a uniform method of referencing
   global variables.
3. Its implementation must be so secure that no program,
   legal or illegal, can destroy itself or its supporting
   mechanisms.

Page 9.
An Implementation of SEAL on Multics.

Part II: Error Controlling Methods.

Section II.1: Why Control Errors?

Error control, in a general sense, pervades all aspects of computing. In hierarchically organized systems the correct operation of each level depends on the correct operation of all lower levels. Applications programs written in a compiled language depend on the correct operation of the compiler; the compiler depends on the correct operation of the operating system; the operating system depends on the hardware. As systems designers and programmers begin to get tired of fixing the same sorts of bugs over and over with each new system, they are beginning to search for ways to build correct systems from the outset.

Various error controlling schemes have been proposed thus far: structured programming, goto-less programming, "chief programmer," automatic verification, etc. That almost all such efforts have brought favorable results is evidence of the magnitude of the problem. But attempts to write correct programs are doomed from the start in many languages. Most of the computer languages in use today were designed before the problem was so great. If new solutions are not found, software costs will con-
An Implementation of Seal on Multics.

continue to rise. It is indeed ironic that hardware costs are fast becoming negligible compared to software costs. (1)

Seal, however, was designed from the start to aid error control. Seal provides a simple language for writing sound programs, and is able to catch and diagnose all illegal programs, either at compile time or execution time. This philosophy of complete error checking pervaded the entire project, and each phase of it will be discussed in the sections that follow.

Section II.2: Language Design.

The Seal language restricts the user to seven basic data types, and completely defines the operations allowed upon each type. It specifies that type-checking must be performed, either at compile time or execution time. Although the language allows pointer variables (in the PL/I sense) it restricts them sufficiently so that it can completely define their correct usage. The language does not incorporate any feature which enables the programmer to perform an undefined action.

The shortcomings of only partial error checking become obvious in large subsystems composed of separately compiled programs.

An Implementation of Seal on Multics.

Large subsystems are usually composed of many smaller programs, for reasons of modularity, maintainability, transportability (sensitive code is usually isolated), economy, and readability. Nearly all programming languages offer subroutine-type programming in some form.

Unfortunately, separate programs usually mean separate compilations, and separate compilations mean that the error checking capabilities of the compiler are greatly reduced. Only a few languages have attempted to define, for all cases, the semantics of separately compiled programs. While PL/I, for example, defines how separately compiled programs shall behave, it does not require an implementation to perform the checks. Multics PL/I has no provision for validating arguments on a call between separately compiled programs, and many subtle bugs have been difficult to track down by this omission. (1) By design, Seal requires than an implementation perform these checks. Seal has no undefined cases, or illegal programs which can "sneak by" the compiler.

---

(1) Multics PL/I is forced to give the following disclaimer: "A program that violates the constraint[s listed in this manual] may or may not be compiled by the Multics PL/I compiler. If compiled, it may or may not execute. If executed, it may or may not produce consistent results in the current or future versions of the implementation." Multics PL/I Language, Document AG94, Honeywell Information Systems, Inc., 1972, page 1-5.
An Implementation of Seal on Multics.

Another area of trouble in modern languages is the very general "address variable" capability known as "pointers" (PL/I) or "references" (Algol). Unfortunately, PL/I gives the programmer the ability to easily perform, within PL/I, actions which are undefined by the language. Hence, the compiler cannot diagnose, or even attempt to diagnose, constructs which have undefined actions, or would cause the runtime support programs to perform incorrectly. For example, a Multics PL/I program can easily destroy the data in the stack segment, either accidentally or on purpose. The programmer can just as easily write constructs that are syntactically correct, but whose semantics are undefined. Such constructs may execute, and may even yield some "meaningful" result. The compiler writer and language designer would very much like to catch all illegal uses of the language, and further, would like to provide a language which has no uncheckable constructs. Also, most users would like to be assured that they cannot fall into the trap of accidentally using such quirks.

Seal provides address variables, but instead of using the more general PL/I-type definition, it uses an Algol-like approach. A Seal "reference variable" not only specifies an address, but also completely specifies the datatype. Stated in PL/I terms, it is possible to have a pointer to an integer, or a pointer to a character string, but they are interchangeable;
An Implementation of Seal on Multics.

Pointers in Seal are specific, not general. Thus it isn't possible for a user to twiddle bits in his character strings or numbers by overlaying them with bit strings. And it is possible for the compiler to check for such illegal uses of reference variables and issue an error message. A consequence of this restriction is that it is not possible to write a garbage collector or storage allocator in Seal. The Seal programmer can only refer to Seal values; it is not possible to manage storage. This is not a severe limitation since storage management of the Seal values themselves is provided by the language and implementation.

Section II.3: Code Generator Design.

There is nothing radically new about the implementation itself. Rather, already proven methods were used to build the Seal code generator. This has left time to explore and develop methods for dealing with the other parts of the thesis.

The code generator is a one-pass, table-driven program. The output of the first pass of the compiler, the parser, is in the form of triples. Each triple consists of an opcode and two operands. The code generator expands each triple into Multics machine code by interpreting a small table. This table describes the code sequences and code generator state changes necessary for each triple. Some triples have several possible different inter-
An Implementation of Seal on Multics.

interpretations, depending on operand type. The table is also used to separate these sub-cases automatically, and at the same time, check for illegal operand types.

The lowest levels of the code generator were designed first. The Seal data-types and control structures were mapped onto the Multics base system. Then the Seal operators were mapped into sequences of H810 machine instructions. In this manner, the allowed operand types and the code sequences necessary to implement the operators were collected in a table. Then the table was augmented to include special commands describing the code generator state changes necessary to generate an object program. These commands are also used to request special actions to be performed by the code generator. Refer to Appendix B for a listing of the Multics Seal code generator, and to Appendix C for a listing of its table.

This table was almost completely designed and written before any code generator programs were started. It is the central force behind the code generator and the desire to control the design and implementation errors which would lead to bugs. The code sequence generated for any language construct may be determined (given the output of the parser) by going to the table entry for the language operator and reading a few lines of information. All code generating decisions are made by the table.
An Implementation of Seal on Multics.

Some of these decisions are expressed as direct H6180 machine code, some as state changes or requested actions in the code generator program. Nevertheless, by merely reading a few lines in the table, one may get most of the information necessary to determine the behavior of the code generator. This technique is in direct contrast to procedural-type code generators which tend to distribute such decisions over a large number of modules. Experience indicates that such an approach may be necessary for complicated languages such as PL/I, but it leads to rather esoteric bugs and manageability problems. The simplicity of the Seal language permitted the table approach, and hence, its easier control of errors.

The only major disadvantage of the table approach is that more complicated constructs cause the table to become a programming language of its own. The table used by the Seal code generator is only a small step above a macro-assembler language, yet it has "if-then-else" clauses, symbolic variables, and a limited form of subroutine call. The compiler writer must be careful that his internal language does not become too unwieldy itself. Otherwise, too much effort will be spent debugging it, when the real task is to correctly compile the user's program.
The design and implementation of the Seal code generator was guided by the desire to keep the number of compiler bugs to a minimum. Several techniques served as the primary means used to achieve this goal. They are:

1) Modularity,
2) Simplicity,
3) Limiting optimizations,
4) Self-checking,
5) Programming style.

Section II.4.1: Modularity.

The procedures which comprise the Seal code generator are quite modular. Once again, this design decision follows from the desire to localize the decision-making process in the hopes of controlling errors. Modularity also simplifies the whole compiler-building process, from design through coding to debugging. Experience with other compilers suggests that the major cause of compiler bugs is unanticipated side-effects between modules. Often they arise only under special circumstances, and thus are very hard to track down. Needless to say, much careful work is required to keep the compiler truly modular but the reward is that the thousands of decisions made during the compilation pro-
An Implementation of Seal on Multics.

cess are much better organized, and much more likely to be correct in all cases.

Section II.4.2: Simplicity.

The area of simplicity is often taken for granted, and is one of the first to get out of hand in any large system. It is often compromised in the name of efficiency. Unfortunately, the Seal code generator is no exception. However, the division of labor between a table and interpreting program has simplified matters. The table is designed to enumerate the many, many code patterns in a uniform, simple manner. The table is structured as a strictly diverging tree; cases are divided first by triple operator, then by operand type, then by predicates. Since each case is listed completely separately, and since the decision making "predicates" (if-then-else statements) are specified per case, it is not possible for decisions made by one case to affect another; each entry is physically and logically separate. See Appendix C for a listing of the table. The interpreting program, on the other hand, is designed to make only general decisions which are valid for all cases. Examples of decisions made by the program are putting literal constants into instructions via the H6180 "direct" modifiers, or using the H6180 instruction-counter
An Implementation of Seal on Multics.

modifier. (1) The division of the compiler into a table and driving program greatly simplifies the handling of special cases, thus reducing the likelihood of errors.

Other major design decisions helped to achieve simplicity. The use of triples as the internal representation forced a table representation that localized decisions, and kept the enumerating of cases uniform and simple. It also allowed the code generator to be non-recursive, which simplified its overall design and implementation.

Section II.4.3: Limiting Optimizations.

The decision to limit the optimizations performed by the code generator is the most radical departure from previous compilers. In the case of the Multics Seal implementation, only optimizations which could be done systematically and in a generalized fashion were permitted. This limited the implementation to three optimization methods, which are:

1) Optimize by combining redundant triples (which compute the same value),

2) Optimize by remembering the machine state during code

---

(1) In assembler notation, these decisions are: 'ldq constant_address' is changed to 'ldq constant_value, dl' and 'ldq constant_address' is changed to 'ldq constant_address*, ic' In the first example, the constant must be less than $2^{18}$, in the second example it may be any value.

Page 19.
An Implementation of Seal on Multics.

generation,

3) Optimize by sub-casing a given triple case.

No optimizations are performed via any look-ahead or look-behind, other than those implied by the machine state (i.e., register contents).

The sole task of the optional optimizer pass (between the parse and the code generator) is to perform the first optimization. This optimization is quite simple; Gries (1) estimates that a half-page of code is all that is required to implement it. An example of this optimization is:

Triple #: Triple operator. Operands.
#1. add i, j
#2. assign k, #1
#3. add i, j
#4. assign m, #3

into the following:
#1. add i, j
#2. assign k, #1
#3. assign m, #1

The second optimization is performed by the code generator as it expands each triple into machine code. It is perhaps the

---

An Implementation of Seal on Multics.

most error-prone optimization, due to the fact that commands to change the register state information must be explicitly entered in the table. An example of this optimization in the expression "i * j + k" is:

Machine Instruction.

load i
mult j
store tl
load tl
add k

into the following:

load i
mult j
add k

The third optimization is performed only by the table, and it is localized to a given triple-operand combination. The following diagram is an example of the optimization of the expression "x / (y + 1)", by using an inverted divide instruction.

load y
add 1
store tl
load x
divide tl

Page 21.
An Implementation of Seal on Multics.

is optimized into:

```
load   y
add    l
divinv x
```

The limiting of the optimizations performed is a another major method for controlling the implementation errors. Experience has shown that look-ahead and look-behind is very sensitive to special cases, and a frequent cause of esoteric bugs. It is the author's opinion that such look-ahead and look-behind (effectively combining triples) is symptomatic of poor communication between the parser and code generator of a compiler, and in a broader sense, between the programming language and the host computer. There seems to be no general method for optimizing adjacent triples in a foolproof manner. It would certainly not be possible to tabulate all of the possible triple pairs which might be optimized, much less higher numbers of triples. Requiring communication between triples to be via the registers may hinder code optimization, but it keeps the compiler well-defined in terms of its pre-specified tables. The real solution, when such look-ahead or look-behind appears to be necessary, is to change the parser (or language) to define a new construct having the desired properties of the old pair of constructs. This has the advantage of remaining within the existing code generator mecha-

Page 22.
An Implementation of Seal on Multics.

nism, and it states explicitly what the code generator was previ­ously trying to do implicitly. In this way, there can be no mis­understanding about the meaning of a given construct in any con­text, and the code generator can easily optimize it. Achieving such a design takes a lot of communication between the code gen­erator designer, language designer, and machine designer.

It is sometimes difficult to define new triples which are combinations of old ones because triples have only two operands. The solution has been to produce an invariant sequence of triples when a single triple is insufficient. Thus it is possible to express constructs which have more than two operands, but still use only two-operand triples. This produces a certain amount of complexity in the compiler (more code is required to recognize and handle the invariant sequences of triples as a unit), but it states exactly what is desired by a given construct.

Optimal code is achieved by picking triple operators which map well into H6180 machine code. Some iteration is necessary to achieve this (i.e., a change to the language or parser) but for the short term, or for infrequently used constructs, it is often easier to suffer with a few extra instructions.
An Implementation of Seal on Multics.

Section II.4.4: Self-checking.

Self-checking is an integral part of the compiler; it is not optional. The checks are designed to prevent the code generator from producing incorrect code due to internal errors which cause the code generator to enter an undefined state, or "mess up" its tables. Most of the tests are simple checks on the validity and consistency of its internal data bases. For example, reference counts are handled very carefully. A reference count is a number associated with a computed result, or triple output in Seal's case, which informs the code generator how many operands use this output. They are provided solely for optimization purposes. They enable the compiled code to evaluate an expression a minimum number of times, and then throw it away when it is no longer needed. If the reference count is too high, the evaluated result will never be discarded. If the reference count is too low, the result will be discarded too soon, and a compiler addressing error will occur. The Seal code generator checks the reference count each time it is decremented to ensure that it does not go too low, and it checks for the erroneous existence of saved values at predetermined points, in combination with the optimizer, which never uses a saved result across a transfer-of-control point.
An Implementation of Seal on Multics.

Other checks are for undefined addresses, undefined operands, unhandled triple cases, and failing to save a result whose reference count was non-zero. These checks, however, neither prevent mistakes nor guarantee perfection. Bugs not caught automatically include destroying a register without so indicating in the table, and incorrect code sequences.

Section II.4.5: Programming Style.

Programming style is an increasingly popular issue. The author is firmly convinced that consistency and readability are not only virtuous, they are absolutely necessary! Powerful languages such as Algol and PL/I offer very convenient control structures for writing readable, understandable programs. Newer languages are refining existing, overly general constructs like "goto" into less error-prone ones such as "case."

Four years of experience programming in PL/I has produced a short list of rules for producing consistent, readable programs. (1) Refer to Appendix A for a list of the rules and to Appendix B for an example of how they were used in this implementation.

(1) For a similar set of rules for FORTRAN, see "How to Write a Readable FORTRAN Program," by Daniel D. MacKenzie and Gerald M. Weinberg, DATAMATION 18, 10 (October 1972), 77-77.
An Implementation of Seal on Multics.

Section II.5: Detection of User-Program Errors.

Returning to the previous example of subsystems, it has been the author's experience that subsystems on Multics are very rarely "stable." In fact, the experience has been that changes can't be made fast enough; a subsystem may be neglected for a relatively long period of time only because there is no one to work on it. Such artificial stability is beyond the scope of this discussion.

Any subsystem which has a more-than-fleeting time span is likely to change hands several times. It may be designed, coded, debugged, enhanced and maintained by five different people. Until inter-programmer communication is improved, a great deal of information will be lost in each transfer, and will have to be regained, often at much expense of time and labor. Since the proper place for such communication is in the programs themselves, much attention must be given to writing programs which are as readable and understandable as possible.

The very need to change, and the "hand-me-down" programming management philosophy create a situation in which no subsystem is ever fully debugged. Each person who modifies the subsystem runs the risk of adding a bug along with his "fix." Every programmer

Page 26.
An Implementation of Seal on Multics.

has his own "war stories" about programs so sensitive that merely "looking at it" is likely to add a bug. Various schemes have been proposed in recent years to cure the evils of programming languages and programmers, but most of them have been quite restricted in their scope. (Prominent are the goto controversy, and the call for "structured programming"). Without dismissing the validity of such approaches, it is evident that Seal's solution is both non-trivial and global. Seal has "thrown in the towel" as far as believing the programmer when he claims that his program or subsystem is debugged. Seal has no separate debugging compiler, debugging interpreter, or debugging options.

Section II.5.1: Compile Time Checks.

Experience with the Multics PL/I compiler has proven the worth of comprehensive compile time error diagnosis. The Seal compiler has over 100 specific compile time error messages. (Perhaps the fact that PL/I has over four times as many messages is some indication of its added complexity). Compilation errors generally do not halt the compiler; every attempt is made to continue. Severe errors detected during the first pass (parse), however, will suppress the second pass (code generation). Compile time error messages print the source statement, its line number, and a short (10-20 word) explanation. No attempt is made

Page 27.
An Implementation of Seal on Multics.

to correct the error.

The following checks are performed at compile-time:
syntax errors,
undefined labels,
invalid selectors applied to user-defined modes (in
PL/I terms, invalid structure member names used in
a reference),
mode errors with built-in operators,
assignment to input-only parameters.

Section II.5.2: Execution Time Checks.

Execution time errors abort the faulty program and print a
diagnostic message giving the name of the program and, when fully
implemented, the source line number. In general, execution time
errors are unrecoverable, so the user is not allowed to restart
the program from the point of interruption.

The following errors are caught at execution time:
subscript range errors,
attempt to use an undefined value,
exponent underflow and overflow,
division by zero,
integer overflow during arithmetic

Page 28.
An Implementation of Seal on Multics.

versions,
mode violations of builtin operators by Any-mode operands,
mismatched modes between arguments and formal parameters,
improperly activated procedure values,
attempt to use null reference values,
range errors in arguments to builtin functions.

Some of the checks are automatically performed by the hardware (division by zero, underflow, overflow), some by adding a few in-line instructions (subscript errors, conversion errors), some by explicit operator calls (mode violation at run-time), and some are caught by the entry operator when the program is invoked (mismatched parameters). Undefined value is the most difficult error to catch. Ideally, this error would be caught by appropriate hardware assistance. This check must be performed at each reference to the value, and the easiest way to do it would be to provide a unique state for each machine word, distinguishable from a legal value, which would cause a hardware fault (trap) upon any attempt to load (read) the value, and which would be reset by any attempt to store into the value. Most simulators provide this capability "for free," but without hardware assistance of some sort it is usually prohibitively expensive in
An Implementation of Seal on Multics.

compiled code. Fortunately, even though the H6180 hardware does not provide the above, ideal solution, it does come close enough. The hardware takes a fault when it recognizes a special indirect word in an indirect chain. Thus, this implementation of Seal references all of its values through an indirect word (pointer) which is initialized to the faulting tag. Using a method suggested by D. A. Moon, (1) the compiled code changes the indirect word to a normal indirect tag at each assignment to a value. This prevents correct programs from faulting, and allows all further references to proceed normally. However, any attempt to use a value before it has been defined by an assignment will cause a fault, and an error-handling program will stop the program and print a message.

The undefined value check is the only one which causes any appreciable overhead. It requires an initialization operation at block entry, an extra instruction at every assignment, and an extra memory reference at each value reference. Other languages which check for undefined values use similar methods. One implementation of WATTOR actually uses the "ideal" method described above. It initializes the storage locations for every value to have a bad parity bit. Upon a read-type reference, the program

(1) Undergraduate member of Computer Systems Research Division at Project MAC.
An Implementation of Seal on Multics.

faults. WATFIV uses a less machine-dependent check; it initializes each value to a predefined constant, and compares the value to the constant at run-time. This has the major disadvantage that the programmer can accidentally run across this value in a legal program. PL/C, (1) checks for undefined values by initializing them to the most negative integer (which has no positive representation in two's complement), and then loading the absolute value into a register set aside for this sole purpose each time the value is referenced in a computation. If the value was undefined, a fault results; if not, execution proceeds. Their checking method could be used by the Multics Seal implementation, but the H6180 architecture severely limits the number of registers, and further, it seemed to the author that an extra instruction at each assignment was preferable to an extra instruction at each reference. The IBM PL/I Checkout compiler (2) checks for undefined values as it interpretively executes the program. It also checks for misusing pointer variables (c.f. section II.2 of this thesis).

Undefined procedure values are detected through a method


Page 31.
An Implementation of Seal on Multics.

described by Fenichel. (3) Basically, each activation record is labeled with a unique number (possibly the value of a hardware microsecond clock), as is each procedure value derived from that activation. This enables the implementation to diagnose the case where an internal procedure is activated (called) after its parent block has returned. This can happen in Seal because procedure values may be stored in static storage, but it is in error to use them unless they are either external (have no parent) or their parent is still active (i.e., on the activation record stack). At each usage of the procedure value (attempt to activate the procedure it represents) the unique number in the value is compared to the unique number in the activation to ensure that it represents the same activation.

An Implementation of Seal on Multics.

Part III: Summary and Suggestions for Further Work.

Section III.1: Summary.

The Multics Seal implementation currently compiles a large number of Seal constructs, and generates an executable Multics object program. The error controlling mechanisms described in this thesis were successful; the code generator and its table were debugged in about eight weeks of part-time work. The high modularity of the code generator enabled each section to be independently debugged, and the simple design kept the problems simple. The self-checking features of the compiler caught many oversights in the early phases of the compiler debugging, and later on they caught and clearly diagnosed problems which would have created obscure errors in the compiled programs.

Most of the compile time and execution time checks mentioned in this thesis have been implemented and tested; all of them have been designed. To the knowledge of the author, the Seal implementation is the only compiled Multics language to offer checking for undefined values.
Section III.2: Problems and Suggestions.

One restriction had to be placed on undefined values in Seal; this implementation does not allow them to be passed as arguments. This decision is primarily due to the implementation method, namely, not only is the value undefined, but so is its address. Since argument passing is implemented on Multics via passing the address of each argument, Seal can not pass undefined values. Fortunately, this decision also has a positive result; Seal programs may call a program in any other language on Multics which follows the standard Multics conventions (e.g. PL/I, FORTRAN, BCPL, etc.). Extending the Seal implementation to allow passing of undefined values would have required a non-standard mechanism.

Section III.3: Suggestions for Hardware Extensions.

The problems mentioned in the preceding section are evidence that more work is needed to define appropriate hardware mechanisms which would remove such restrictions and much of the present overhead. The following section describes H6180 hardware modifications which would benefit the Multics Seal implementation.
An Implementation of Seal on Multics.

Undefined Value.

The following mechanism is proposed for implementing undefined value checking in the hardware:

Each machine word (or byte, or character, or other basic addressing unit) has two states: "defined" and "undefined," distinguishable from any value.

Load-type instructions will fault if the state is undefined.

Store-type instructions will change the state of the word to defined, and complete the store instruction normally.

Instructions will be provided to force the state of a word (or block of words) to either defined or undefined.

Instructions will be provided to test the state of a word (or block of words) without faulting.

Since the address of each value is still perfectly well-defined, argument passing would be possible. Very little overhead would be required to initialize storage locations used for program variables to the undefined state, and no additional instructions would be required. Since the state is independent of the value, any data type (character, bit string, arithmetic) could be stored, and no value would be illegal or tend to be confused with...
An Implementation of Seal on Multics.

the undefined state. This could be implemented by adding one bit per word (since error-correcting codes already add several bits per word, the additional cost should not be too great).

Multiplication.

Seal integers occupy a single machine word on the H6180. The Seal multiply operator is defined to take two such single precision integers and return an integer. However, the H6180 has no single precision multiply instruction; all integer multiplications produce a double-precision result. Adding a single precision multiply would eliminate 4 instructions which now follow every integer multiplication in Seal.

Subscript Checking.

The instruction sequence to check a subscript to ensure that it lies between the lower and upper bounds of an array is very cumbersome, and has several undesired side effects. The bounds must be in the only two 36-bit arithmetic registers on the H6180, even though the final index can have no more than 18 bits of precision. The compiler must use these registers to compute all arithmetic results, and having them serve double duty causes excessive, unnecessary loading and storing. The proposed solution
An Implementation of Seal on Multics.

defines a new instruction (1) which takes the upper and lower bound in a pair of 19-bit index registers, compares them to a subscript in storage, and skips the next instruction if the subscript is within the bounds. In this manner, the compiler can leave the index registers loaded with the bounds, rather than continually loading the 36-bit arithmetic registers. This extension would save at least three instructions for each subscript usage, and possibly more due to the elimination of the dependence on the arithmetic registers.

Section III.4: Further Work.

This section presents some brief ideas for further work in the areas related to this thesis. These ideas assume that the implementation has been completed; once this is done it will be possible to investigate the viability of the new and unique features of the Seal language, such as: Does eliminating file I/O help programs which have used it in the past? What kind of improvements can be achieved by reprogramming some typical data-base management applications (heavy file I/O users) in Seal? Do the extensive checks provided by Seal prove valuable over the

(1) Due to the H6180 architecture, 4 opcodes would be required to specify the 4 possible even/odd index register pairs (0-1, 2-3, 4-5, and 6-7). This may still be thought of as one instruction, however.
An Implementation of Seal on Multics.

life of a subsystem or only at the beginning? Can these checks be implemented cheaply enough to satisfy the needs of the users? Are most of the errors found at compile time or execution time? (As many checks as possible are performed at compile time, for example, only "any mode" variables and parameters need to be type-checked at execution time).
Appendix A. Rules for Formatting PL/I Programs.

The following is a consistent set of rules for formatting PL/I programs to maximize readability and emphasize program structure and flow of control.

Syntactic Rules.

1. A statement group is defined as all of the statements within "procedure; ... end;", "do; ... end;", or "begin; ... end;". All statements within such a group shall be indented 5 spaces or 1 tab. The opening keyword (procedure, begin, do) shall be aligned in the same column as the closing keyword (end).

2. The "then" and "else" keywords shall each be on a separate line, and shall each begin in the same column as the "if" keyword.

3. No more than one statement shall be on the same line.

4. All labels will begin in column 1, preceded by a blank line, and on their own line.

5. One blank is used before and after the following operators:
   =, infix -, infix +, *, /, ||, |, &, "=" , < , >, < =, > =, "">, "<",

   One blank precedes, and none follow, these operators:
   -, prefix -, prefix +.

Page 39.
Appendix A. Rules for Formatting PL/I Programs.

One blank follows, and none precede, the comma.

No blanks precede or follow these delimiters:
"(", ")", "->", ".", ";".

Semantic Rules.

1. Variable names are as specific as possible and not abbreviated or "acronymed" (unless acceptable in normal English usage). Overly general names such as "i", or "switch" are restricted to a very local context (a few statements).

2. Goto statements are used only to construct a case statement (using a label vector and an indexed goto to enter a case, and another goto to leave the case), or to perform a non-local goto when aborting further processing.

3. Comments are encouraged, but should not be so trivial as to be merely "noise" (e.g. "i = i + 1; /* increment index */"). The real documentation is always the program; the comments should serve to give a global overview (of a procedure or a section of code), point out an obscure point, or give a simple explanation. Each procedure block has a global comment describing its function, calling sequence, side effects, etc. This comment comes immediately before the procedure statement, beginning in column 1, and surrounded by blank lines. Comments
Appendix A. Rules for Formatting PL/I Programs.

giving important information are placed on the line before the statement to which they refer, beginning in the same column. Comments giving trivial, yet useful information about a statement are placed on the same line in the right-hand margin, starting in column 50 or 60.

4. Blank lines are used to improve readability by grouping similar statements and separating dissimilar ones. Page ejects and vertical tabs provide the same service for internal procedures or large sections of code. If an internal procedure is referenced only in a small section of code, it may be placed nearby; otherwise it is placed at the end of the containing block.

5. All declarations come before any statements in a block. Each storage class uses a separate declare statement, in which the declarations are grouped alphabetically or logically (named constants might be sorted by initial value). Structures are indented 2 spaces per level, and the attributes of each member are aligned in the same column. Include files are used for declarations needed by more than one program.
Appendix B. Multics Seal Code Generator Program.

(subscriptirange):
seal_code_generator:
    procedure(work_seg_ptr, error_printer);
    acl subscriptirange condition;
on subscriptirange
    begin:
call loa("Subscriptirange in cg, debug");
call debug;
end;

/* Code generation program for the SEAL compiler.
Pau! Green, January 1973. */

/* Last Modified by PG on 5/17/73 */

/* parameters */
dcl   error_printer entry(ptr, fixed, fixed, fixed, fixed, fixed, fixed) returns(bit(1)) parameter,
      work_seg_ptr ptr parameter;

/* entries */
dcl   loa_entry, /* temporary */
      ioa_entry options(variable); /* temporary */

dcl   coo_err_entry options(variable),
establish_cleanup_proc_entry(entry),
hcs_terminate_noname_entry(ptr, fixed bin(35)),
hcs terminate_noname_entry(ptr, fixed bin(35)),
hcs_struncate_seg_entry(ptr, fixed, fixed bin(35)),
      seal display macro entry(ptr, fixed, bit(1));

/* automatic items */
dcl   (defs, jefs_rel, link, link_rel, name, object_seg, operand_info_ptr, operator_info_ptr,
      op, pattern_base, previous_definition, scratch(8),
      sp, stack_base, sym, sym_rel, text, text_rel, vp) pointer,
      (defs_ic, link_ic, object_ic, position, saved_symb_ic,
      stack_offset, symb_ic, text_ic, value_offset) fixed bin(18),
      (argument_index, b, bitlen, count, dils, element_size, l, lr, l, m,
      n, opcode, operand(10), output, pattern_ic, s, scratch_index, stack_end, temporary_count) fixed bin,
      (left_relocation, name_definition, operands, rel_code,
      signature_definition, zero_definition) bit(6) aligned,
      use fixed initial(0), /* temp */
code fixed bin(35),

Page 42.
Appendix B. Multics Seal Code Generator Program.

(befor_e_first_flowchange, done, found, on_heap) bit(1) aligned,
arg bit(2) aligned,
type1, type2 bit(9) aligned,
address bit(36) aligned,
1 value temp aligned like value

/* built-in functions */
dcl (addr, address, binary, bit, divide, fixed, hbound, index, lbound, length,
max, mod, null, pointer, rel, size, string, substr) builtin;

/* based items */
dcl 1 temporary_seal_name based aligned,
2 value_header bit(36),
2 make_offset_even unal bit,
2 runtime_allocate unal bit,
2 element_size unal fixed bin,
image based bit(bitten) aligned,
word_copy_image dim(n) fixed bin(35) aligned based,
based_string char(252144) aligned based,
1 unpacked_relocation aligned based,
2 half_word dim((itm-1) unaligned bit(16)),
1 word aligned based,
2 left_half unal bit(16),
2 right_half unal bit(16),
1 words aligned based,
2 first bit(36),
2 second bit(36),
2 third bit(36); 

do cl 1 operator_info aligned based(operator_info_ptr),
2 offset unal fixed bin(17),
2 type1 unal bit(5),
2 type2 unal bit(5),
2 length unal fixed bin(11),
1 operand_semantics dim(25) aligned based(operator_info_ptr),
2 operands unal,
3 opnd1 bit(9),
3 opnd2 bit(9),
2 pattern_offset unal fixed bin(17).

Page 43.
Appendix B. Multics Seal Code Generator Program.

1 pattern        dim(106) aligned based,
2 part1          unal bit(28),
2 pattern_flag   unal bit(1),
2 part2          unal bit(7),

1 instruction    aligned based,
2 ptr_reg        unal bit(3),
2 offset         unal bit(15),
2 opcode         unal bit(10),
2 inhibit        unal bit(1),
2 use_ptr_reg    unal bit(1),
2 tag            unal bit(6),

1 pattern_word   aligned based,
2 index_with_arg1 unal bit(1),
2 arg1           unal bit(3),
2 index_with_arg2 unal bit(1),
2 arg2           unal bit(8),
2 arg3           unal bit(10),
2 flag           unal bit(1),
2 pattern_op     unal bit(7),

1 Instruction_word aligned based,
2 ptr_reg        unal bit(3),
2 arg            unal bit(2),
2 offset         unal bit(13),
2 opcode         unal bit(10),
2 inhibit        unal bit(1),
2 use_ptr_reg    unal bit(1),
2 tag            unal bit(6),

/* text section references (unset internal static initial) */

objc(1) {   no_check_code  initial("00000000")  /* 0 */
    /* types 1-7 are for built-in modes */
    rel_code       initial("00000100")  /* 1-7 */
    list_code      initial("00000101")  /* 8 */
    constant_code  initial("00000110")  /* 10 */
    user_mode_code initial("00000111")  /* 11 */
    bit(3) aligned static,
    stack_frame_first_available_location fixed bin static initial(58),  /* must be even */
    even_offset_required(8) bit(1) aligned static
    initial(1'b, "0"b, "0"b, "0"b, "0"b, "1"b, "1"b, "1"b)
                objc(1) empty_register fixed static initial(0),

Page 44.
Appendix B. Multics Seal Code Generator Program.

/* NB: node_length(X) is the number of 36-bit words necessary to hold a SEAL value
   with mode X=1,7 (any_type to symbol_type). The number is derived from
   node_length(1) = size(any_model), etc. */

node_length(7) fixed static init(l1,l1,l1,0,2,64),
max_no_lims fixed init(l5) aligned static;

any_name char(4) aligned internal static init("seal"),
(activation_record_up_zero_indirect activation_record_up_zero
linkage_section_up_zero_indirect link_list_up_zero indirect
arg_list_up_zero indirect
lts_to_activation_record_f13
lts_to_temporary_storage_f13
init("10000000000000000000000000010000b", /* print */
activation_record_up_zero_init("1100000000000000000000000010000b", /* print */
linkage_section_up_zero_init("1100000000000000000000000010000b", /* print */
arg_list_up_zero_init("1100000000000000000000000010000b", /* print */
lts_to_activation_record_f13_init("1100000000000000000000000010000b", /* print */
lts_to_temporary_storage_f13_init("1100000000000000000000000010000b", /* print */
dct store_00 bit(13) dim(26) static aligned
initial( "1111101000b", /* sta 755(3) */
"1111011000b", /* sta 756(0) */
"1010111100b", /* dst 457(0) */
(10)"0"b,
"1011000000b", /* sxt 441(3) */
"1011000010b", /* sxt 441(0) */
"1011010010b", /* sxt 442(0) */
"1011001110b", /* sxt 443(0) */
"1011011110b", /* sxt 444(0) */
"1011011010b", /* sxt 445(0) */
"1011011100b", /* sxt 446(0) */
"1011011100b", /* sxt 447(0) */
"1011010000b", /* srl 295(0) */
"1011010010b", /* srl 295(1) */
"1011010100b", /* srl 296(0) */
"1011010110b", /* srl 296(1) */
"1011010010b", /* srl 297(0) */
"1011010110b", /* srl 297(1) */
"1111011000b", /* srl 293(0) */
"1111011100b", /* srl 293(1) */
"1111011100b", /* srl 295(0) */
"1111010010b", /* srl 295(1) */
"1111011010b", /* srl 296(0) */
"1111011110b", /* srl 296(1) */
"1111010010b", /* srl 297(0) */
"1111010110b", /* srl 297(1) */
/* These codes are for use with the register information entries. */

ccl ( A_reg initial(1),
 Q_reg initial(2),
 E&Q_reg initial(3),
 Any_reg initial(-1),
 X1_reg initial(5),
 X1_reg initial(6),
 X2_reg initial(7),
 X3_reg initial(8),

Page 45.
Appendix B. Multics Seal Code Generator Program.

X4_reg initial(9),
X5_reg initial(11),
X6_reg initial(11),
X7_reg initial(12),
AP_reg initial(13),
AB_reg initial(14),
BP_reg initial(15),
SB_reg initial(16),
LP_reg initial(17),
LB_reg initial(18),
SP_reg initial(19),
SB_reg initial(20),
constant initial(21)) fixed static:

/* external static */

dcl
  { seal_patterns_operator_table,
    seal_patterns_entry_control_word1_offset fixed bin,
    seal_patterns_undefined_label_instruction bit(36) aligned,
    seal_code_generator_symbol_table,
    seal_version_char character(256) varying aligned) external static;

/* include files */

/* Declarations for the Multics Standard Object Segment */

dcl
  obj_map_ptr,
  object_map_offset bit(18) unaligned based;

/* BEGIN INCLUDE SEGMENT ... obj_map.incl */
coded February 8, 1972 by Michael J. Spier
  /* last modified 14v, 1972 by H. Weaver */

declare
  1 map aligned based(map),
  2 declvers fixed bin,
  2 identifier char(8) aligned,
  2 text_offset bit(18) unaligned,
  2 text_length bit(18) unaligned,
  2 def_offset bit(18) unaligned,
  2 def_length bit(18) unaligned,
  2 link_offset bit(18) unaligned,
  2 link_length bit(18) unaligned,
  2 symb_offset bit(18) unaligned,
  2 symb_length bit(18) unaligned,
  2 bmap_offset bit(18) unaligned,
  2 bmap_length bit(18) unaligned,
  /* structure describing standard object map */
  /* must be the constant "obj_map" */
  /* offset rel to base of object segment of base of text section */
  /* length in words of text section */
  /* offset rel to base of object seg of base of definition section */
  /* length in words of definition section */
  /* offset rel to base of object seg of base of linkage section */
  /* length in words of linkage section */
  /* offset rel to base of object seg of base of symbol section */
  /* length in words of symbol section */
  /* offset rel to base of object seg of base of break map */
  /* length in words of break map */

Page 46.
Appendix B. Multics Seal Code Generator Program.

```plaintext
2 format aligned, /* word containing bit flags about object type */
3 bound bit(1) unaligned, /* on if segment is bound */
3 relocatable bit(1) unaligned, /* on if seg has relocation info in its first symbol block */
3 procedure bit(1) unaligned, /* on if segment is an executable object program */
3 standard bit(1) unaligned, /* on if seg is in standard format (more than just standard map) */
3 unused bit(1) unaligned; /* not currently used */

END INCLUDE SEGMENT ... obj_map.incl.pl1 */

dcl 1 std_symbol_header based aligned,
2 dcl_version fixed bin,
2 identifier char(8),
2 gen_number fixed bin,
2 gen_created fixed bin(71),
2 object_created fixed bin(71),
2 generator char(8),
2 gen_version unaligned,
3 offset bit(18),
3 size bit(18),
2 user1d unaligned,
3 offset bit(18),
3 size bit(18),
2 comment unaligned,
3 offset bit(18),
3 size bit(18),
2 text_boundary bit(18) unaligned,
2 stat_boundary bit(18) unaligned,
2 source_map bit(18) unaligned,
2 area_pointer bit(18) unaligned,
2 backpointer bit(18) unaligned,
2 block_size bit(18) unaligned,
2 next_block bit(18) unaligned,
2 rel_text bit(18) unaligned,
2 rel_cst bit(18) unaligned,
2 rel_link bit(18) unaligned,
2 rel_symbol bit(18) unaligned,
2 mini_truncate bit(18) unaligned,
2 maxi_truncate bit(18) unaligned;

dcl 1 seal_symbol_block aligned based,
2 symd_header like std_symbol_header;

dcl 1 definition_header aligned based,
2 definition_list unal bit(18),
2 unused unal bit(36),
2 flags unal,
3 new bit; /* always on */
```

Page 47.
Appendix B. Multics Seal Code Generator Program

```plaintext
3 ignore
3 unused
   bit;
   bit(16);

dcl 1 definition
   aligned base;
   forward  unal bit(18);  /* offset of next def */
   backward unal bit(18);  /* offset of previous def */
   value    unal bit(18);
   flags    unal;
   new      bit(1);
   ignore   bit(1);
   entry    bit(1);
   retain   bit(1);
   descriptor bit(1);
   unused   bit(10);
   class    unal bit(3);
   symbol   unal bit(18);  /* offset of ACC for symbol */
   segment  unal bit(18);  /* offset of segment def */

dcl 1 expression_word
   aligned base;
   2 type_pair     unal bit(18);
   2 expression    unal bit(18);

dcl 1 type_pair
   aligned base;
   2 type         unal bit(18);
   2 expr         unal bit(18);
   2 segname      unal bit(18);  
   2 entryname    unal bit(18);

dcl 1 source_map
   aligned base;
   2 version      fixed bin;
   2 number       fixed bin;
   2 name         (refer(source_map,number))   aligned;
   3 ptype        unaligned;
   4 offset       bit(18);
   4 size         bit(18);
   3 wid          bit(36);
   3 dtm          fixed bin(11);

dcl 1 relocation
   aligned base;
   2 left         unal bit(18);
   2 right        unal bit(18);

   1 packReloc     aligned base;
   2 dcl_version  fixed bin;
   2 string       bit(36*5553s) varying;

   (rel_absolute initial("20300")
```

Page 48.
Appendix B. Multics Seal Code Generator Program.

rel_text initial("10000"n),
rel_link18 initial("10010"n),
rel_negative_link18 initial("10011"n),
rel_link19 initial("11000"n),
rel_off initial("10101"n),
rel_symbol initial("10110"n),
rel_negative_symbol initial("10111"n),
rel_int_storage15 initial("11001"n),
rel_self initial("11100"n),
rel_exp_absolute initial("11110"n) bit(5) static aligned;

dcl 1 entry_sequence aligned based,
2 entry_definition unal bit(18);
2 unused; unal bit(18);

dcl 1 control_words aligned based,
2 stack_offset unal bit(18);
2 stack_size unal bit(18);
/* words which immediately follow entry instructions */
/* offset (ref to base of text) of template stack frame */
/* size of template stack frame in words */

dcl 1 linkage_header aligned based,
2 unused_1 bit(36),
2 definitions_offset unal bit(18),
2 first_reference_offset unal bit(18),
2 unused_2 pointer,
2 linkage_section pointer,
2 links_offset unal bit(18),
2 linkage_length unal bit(18),
2 object_segment unal bit(18),
2 unused_3 unal bit(18);

dcl ( du_mod init("00111"n), */ 03 */ /* Instruction Modifiers */
d1_mod init("00110"n), */ 01 */
d2_mod init("00100"n), */ 02 */
d3_mod init("00000"n), */ 00 */
d4_mod init("11111"n), */ 55 */
d5_mod init("11101"n), */ 54 */
d6_mod init("11010"n), */ 53 */
d7_mod init("10100"n), */ 52 */
d8_mod init("10010"n), */ 51 */
}
Appendix 9. Multics Seal Code Generator Program.

x6_mod_init("001110"b), /* 16 */

x7_mod_init("001111"b), /* 17 */

lts_mod_init("100110"b), /* 18 */ /* Pointer Modifiers */

lfp_mod_init("100111"b), /* 19 */

lfp_mod_init("100100"b), /* 20 */

flp2_mod_init("100110"b), /* 21 */

flp3_mod_init("100111"b), /* 22 */

bit(6) aligned internal static;

dcl ( r_tag_init("0u"b), /* First 2 bits of Modifier field */

rl_tag_init("01"b), /* du, dl not allowed */

lr_tag_init("11"b),

lt_tag_init("10"b))

bit(2) aligned static;

dcl ( ao_mod_init("001"b), /* 0 */ /* Pointer Registers */

ab_mod_init("001"b), /* 1 */

bb_mod_init("010"b), /* 2 */

bb_mod_init("111"b), /* 3 */

lb_mod_init("100"b), /* 4 */

lb_mod_init("110"b), /* 5 */

sb_mod_init("110"b), /* 6 */

sb_mod_init("111"b), /* 7 */

bit(3) aligned static;

dcl 1 lts_model aligned based,

2 (unused1 bit(3),

segment bit(25),

ring bit(3),

unused2 bit(9),

its bit(6),

offset bit(24),

unused3 bit(3),

bit bit(6),

unused4 bit(3)) unaligned;

dcl 1 packed_ptr_model aligned based,

2 (bit bit(6),

segment bit(12),

offset bit(12)) unaligned;

dcl 1 lts_model aligned based,

2 (str_reg bit(3),

unused1 bit(27),

lo bit(2),

offset bit(16),

unused2 bit(3),

unused3 bit(3)).
Appendix B. Multics Seal Code Generator Program.

bit bit(6),
unused3 bit(9)) unaligned;

/* Declarations for the SEAL compiler's internal storage */
/* This declaration must describe the same storage as is described by the declaration of opcodes. */
declare opcode_table dim(127) fixed based(adr(opcodes));

/* In the constant defining each operator, $i$ is an index to another macro or to a symbol node, $a$ is an index to a symbol node, $n$ is an integer constant, and $b$ is an index to a block. An index to another macro is always negative. An index to a symbol or block node is always positive. */
declare 1 opcodes int static, /* opcodes 1 thru 15 must be in the same order as their token codes */
2 assign fixed initial(1), /* assign i,1 */
2 add fixed initial(2), /* add i,1 */
2 sub fixed initial(3), /* sub i,1 */
2 divida fixed initial(4), /* divide i,1 */
2 mult fixed initial(5), /* mult i,1 */
2 and fixed initial(6), /* and i,1 */
2 or fixed initial(7), /* or i,1 */
2 or fixed initial(8), /* or i,1 */
2 cattenate fixed initial(9), /* cattenate i,1 */
2 cattenate_symbol fixed initial(10), /* cattenate_symbol i,1 */
2 less_than fixed initial(11), /* less_than i,1 */
2 greater_than fixed initial(12), /* greater_than i,1 */
2 less_or_equal fixed initial(13), /* less_or_equal i,1 */
2 greater_or_equal fixed initial(14), /* greater_or_equal i,1 */
2 equal fixed initial(15), /* equal i,1 */
2 not_equal fixed initial(16), /* not_equal i,1 */
2 minus fixed initial(17), /* minus i,1 */
2 exponentiate fixed initial(18), /* exponentiate i,1 */
2 complement fixed initial(19), /* complement i,1 */
2 deraf fixed initial(20), /* deraf i,1 */
2 negate fixed initial(21), /* negate i,1 */
2 lock fixed initial(22), /* lock i,1 */
2 unlock fixed initial(23), /* unlock i,1 */
2 test_lock fixed initial(24), /* test_lock i,s */
2 case_of fixed initial(25), /* case_of n,i */
2 case_limit fixed initial(26), /* case_limit n,i */
2 case_jump fixed initial(27), /* case_jump i,n */
2 branch fixed initial(28), /* branch s */
2 branch_true fixed initial(29), /* branch_true s,i */
2 branch_false fixed initial(30), /* branch_false s,i */
2 label
3 fixed initial(31), /* label s */
4 procedure
5 fixed initial(31), /* procedure s */
6 end
7 fixed initial(32), /* end */
8 link
9 fixed initial(33), /* link l */
10 unused_34
11 fixed initial(34), /* element l */
12 element
13 fixed initial(35), /* list n */
14 list
15 fixed initial(36), /* arg l */
16 arg
17 fixed initial(37), /* call l */
18 call
19 fixed initial(38), /* ret l */
20 ref
21 fixed initial(39), /* reduce s, l */
22 reduce
23 fixed initial(40), /* block b */
24 block
25 fixed initial(41), /* select s, l */
26 select
27 fixed initial(42), /* noo l */
28 nop
29 fixed initial(43), /* mode_select s, l */
30 mode_select
31 fixed initial(44), /* line_number n */
32 line_number
33 fixed initial(45), /* address 1 */
34 addr
35 fixed initial(46), /* encode_dims i, n */
36 encode_dims
37 fixed initial(47), /* encode_value i, l */
38 encode_value
39 fixed initial(48), /* arg_list n */
40 arg_list
41 fixed initial(49), /* encode_mode s, n */
42 encode_mode
43 fixed initial(50), /* split_prep i, l */
44 split_prep
45 fixed initial(51), /* built-in opcodes follow */
46 unused_35
47 fixed dim(52:53), /* current [1] */
48 current
49 fixed initial(61), /* errortrap [1] */
50 errortrap
51 fixed initial(62), /* incolumn [1] */
52 incolumn
53 fixed initial(63), /* infilemark [1] */
54 infilemark
55 fixed initial(64), /* item [1] */
56 item
57 fixed initial(65), /* itemmark [1] */
58 itemmark
59 fixed initial(66), /* linemark [1] */
60 linemark
61 fixed initial(67), /* inpagemark [1] */
62 inpagemark
63 fixed initial(68), /* instream [1] */
64 instream
65 fixed initial(69), /* linesize [1] */
66 linesize
67 fixed initial(70), /* outcolumn [1] */
68 outcolumn
69 fixed initial(71), /* outfilemark [1] */
70 outfilemark
71 fixed initial(72), /* outitem [1] */
72 outitem
73 fixed initial(73), /* outitemmark [1] */
74 outitemmark
75 fixed initial(74), /* outpagemark [1] */
76 outpagemark
77 fixed initial(75), /* outstream [1] */
78 outstream
79 fixed initial(76), /* pagesize [1] */
80 pagesize
81 fixed dim(73:79), /* single argument bif */
82 abs
83 fixed initial(80), /* abs l */
84 atan
85 fixed initial(81), /* atan l */
Appendix B. Multics Seal Code Generator Program.

2 boolean fixed initial(82); /* boolean */
2 cell fixed initial(83); /* cell */
2 cos fixed initial(84); /* cos */
2 delete fixed initial(85); /* delete */
2 delete dir fixed initial(86); /* delete dir */
2 dotach fixed initial(87); /* dotach */
2 exp fixed initial(88); /* exp */
2 find fixed initial(89); /* find */
2 floor fixed initial(90); /* floor */
2 integer fixed initial(91); /* integer */
2 isvoid fixed initial(92); /* isvoid */
2 length fixed initial(93); /* length */
2 log fixed initial(94); /* log */
2 log10 fixed initial(95); /* log10 */
2 rank fixed initial(96); /* rank */
2 real fixed initial(97); /* real */
2 sign fixed initial(98); /* sign */
2 sin fixed initial(99); /* sin */
2 size fixed initial(100); /* size */
2 sqrt fixed initial(101); /* sqrt */
2 symbol fixed initial(102); /* symbol */
2 tan fixed initial(103); /* tan */
2 trunc fixed initial(104); /* trunc */
2 unused bif fixed dim(105); /* multiple arg bifs */
2 create fixed initial(106); /* create */
2 is fixed initial(107); /* is */
2 get fixed initial(108); /* get */
2 put fixed initial(109); /* put */
2 void fixed initial(110); /* void */
2 split fixed initial(111); /* split */
2 unused bif fixed dim(1111); /* two argument bifs */
2 attach fixed initial(112); /* attach */
2 createdir fixed initial(113); /* createdir */
2 edit fixed initial(114); /* edit */
2 max fixed initial(115); /* max */
2 min fixed initial(116); /* min */
2 mod fixed initial(117); /* mod */
2 rename fixed initial(118); /* rename */
2 round fixed initial(119); /* round */

/* This variable defines the first optional argument bif. */
declare first bif fixed int static initial(60); /*
/* This variable defines the first single argument bif. */

Page 53.
Appendix B. Multics Seal Code Generator Program.

```c
declare first_blif fixed int static initial(50);
/* This variable defines the first two argument blif. */

declare first_blif2 fixed int static initial(120);
/* This declaration describes the working segment used to contain all tables as well as
the macro file produced by pass one of the compiler. */

declare (os: ws) pointer, k fixed bin;

declare 1 storage based(ws),
2 command_options, bit,
3 brief_option bit,
4 debug_option bit,
5 list_option bit,
6 setc_option bit,
7 parse_option bit,
8 stop_on_macro fixed,
9 stop_on_line fixed,
10 invocation fixed, /* number of activations of the compiler */
11 greatest_severity fixed, /* greatest severity error encountered */
12 severity_plateau fixed, /* cutoff for error printing */
13 constant_list dim(3) fixed, /* index to the chain of symbol nodes that represent literal constants. */
14 last_block fixed, /* The index of the last block */
15 last_symbol fixed, /* The index of the last symbol */
16 last_free fixed, /* The index of the last macro */
17 last_index fixed, /* Index in listing segment */
18 source_seg_limit fixed(24), /* length of source segment */
19 source_seg length fixed(24), /* length of object segment */
20 source_seg pointer, /* pointer to source segment */
21 list_seg pointer, /* pointer to listing segment */
22 output_seg pointer, /* pointer to output segment */
23 options pointer, /* options used in compilation */
24 user_id pointer, /* name of user compiling */
25 object_info aligned,
3 pathname pointer, /* pathname of source segment */
4 segname pointer, /* name of source segment */
5 clock_time fixed bin(71), /* clock reading for compilation */
6 seg fixed bin(71), /* name of segment being compiled */
7 uid bit(31), /* uid of segment being compiled */
2 free_space dim(1663) fixed, /* storage used to hold the internal representation of literal constants and the character-string representation of identifiers and keywords. */
2 block dim(01127),
```

Page 54.
### Appendix B. Multics Seal Code Generator Program

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifiers</td>
<td>Index to a chain of symbol nodes denoting identifiers of this block. /*</td>
</tr>
<tr>
<td>keywords</td>
<td>Index to a chain of symbol nodes denoting keywords of this block. */</td>
</tr>
<tr>
<td>level</td>
<td>Nesting level of this block. */</td>
</tr>
<tr>
<td>free_temps</td>
<td>List of available temporaries, ordered by number of words. */</td>
</tr>
<tr>
<td>stack_base</td>
<td>Base of template stack frame for this block */</td>
</tr>
<tr>
<td>stack_end</td>
<td>Size in words of template stack frame for this block */</td>
</tr>
<tr>
<td>stack_size</td>
<td>Size in words of stack frame for this block (includes temporaries) */</td>
</tr>
<tr>
<td>last_temporary_location</td>
<td>Last temporary location in use */</td>
</tr>
<tr>
<td>left_relocation</td>
<td>Offset in text of first executable instruction in this block */</td>
</tr>
<tr>
<td>symbol</td>
<td>Each element represents an identifier, keyword or literal. */</td>
</tr>
<tr>
<td>name</td>
<td>Pointer to the character-string representation of this symbol. */</td>
</tr>
<tr>
<td>value</td>
<td>PTR to the value of a literal constant. */</td>
</tr>
<tr>
<td>cross_refs</td>
<td>Index to the chain of cross-references to this node. */</td>
</tr>
<tr>
<td>cross_end</td>
<td>Index to the end of the cross-reference chain. */</td>
</tr>
<tr>
<td>def_line</td>
<td>Line on which this item was defined. */</td>
</tr>
<tr>
<td>mode</td>
<td>Index to the declaration of this item's mode. */</td>
</tr>
<tr>
<td>next</td>
<td>Index to the next item in this chain of symbol nodes. */</td>
</tr>
<tr>
<td>location</td>
<td>Position of a formal parameter or address of an item. */</td>
</tr>
<tr>
<td>count</td>
<td>For variables and mode components, this is the dimensionality. */</td>
</tr>
<tr>
<td>general</td>
<td>For external procedure constants, this is the index to the link macro. */</td>
</tr>
<tr>
<td></td>
<td>For named constants, this is the index to the macros produced by value_parser. */</td>
</tr>
<tr>
<td></td>
<td>For operator definitions, this is the index to the macros produced by procedure_body_parser. */</td>
</tr>
<tr>
<td></td>
<td>For mode definitions, this is the index to the chain of symbols representing the components. */</td>
</tr>
<tr>
<td></td>
<td>relocation for &quot;location&quot; field */</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ref</td>
<td>bit,</td>
</tr>
<tr>
<td>list_ref</td>
<td>bit,</td>
</tr>
<tr>
<td>label</td>
<td>bit,</td>
</tr>
<tr>
<td>variable</td>
<td>bit,</td>
</tr>
<tr>
<td>constant</td>
<td>bit,</td>
</tr>
<tr>
<td>component</td>
<td>bit,</td>
</tr>
<tr>
<td>mode_def</td>
<td>bit,</td>
</tr>
<tr>
<td>infix_def</td>
<td>bit,</td>
</tr>
<tr>
<td>prefix_def</td>
<td>bit,</td>
</tr>
<tr>
<td>external</td>
<td>bit,</td>
</tr>
<tr>
<td>input</td>
<td>bit,</td>
</tr>
<tr>
<td>output</td>
<td>bit,</td>
</tr>
<tr>
<td>defined</td>
<td>bit,</td>
</tr>
<tr>
<td>set</td>
<td>bit,</td>
</tr>
<tr>
<td>referenced</td>
<td>bit,</td>
</tr>
<tr>
<td>runtime_allocate</td>
<td>bit,</td>
</tr>
<tr>
<td>passed_as_arg</td>
<td>bit,</td>
</tr>
</tbody>
</table>
Appendix B. Multics Seal Code Generator Program.

```plaintext
2 register
3 mode  unal fixed, /* mode of value */
3 contents unal fixed, /* macro output<1>, empty=0, symbol>0 */
2 temporary
3 mode  unal fixed, /* mode of this temporary or next free temporary */
3 size   unal fixed, /* number of words in temporary */
3 location bit(36) aligned, /* address of temporary */

/* This is the internal representation output by pass one of the compiler. */

2 macro
3 count  unal fixed, /* reference count of macro output */
3 opcode unal fixed, /* operation to perform on operands */
3 op1    unal fixed, /* macro output<1>, empty=0, (symbol | block | constant) */
3 op2    unal fixed, /* same */
3 output unal fixed, /* register<0>, empty=0, temporary>0 */
3 unused unal fixed

/* Each cross_ref occupies one word of the free_space at the head of
the working segment. The variable is the index that identifies the
last used location in the free_space. */

declare 1 cross_ref
  dim(1663) based(addr(free_space)) aligned,
  2 line     fixed unaligned,
  2 next     fixed unaligned;

/* The token array is produced for each call to next_line. It occupies
the segment that eventually will contain the object code. */

declare 1 token
  dim(10030) based(os),
  2 name    pointer,
  2 type    fixed,
  2 size    fixed,
  2 keyword bit,
  2 constant bit;

delclare tstring
  char(token<1>.size) unaligned based;

delclare vstring
  char(256) aligned varying based;

/* Declarations for the Multics SEAL runtime data representations */

dcl 1 mode
  aligned based,
  2 ptr_reg unal bit(3),
  2 word   unal fixed dim(14),
  2 char   unal bit(2),
```

Page 56.
Appendix B. Multics Seal Code Generator Program.

2 bit    unal bit(4),
2 unused1 unal bit(6),
2 length_ap unal bit(6),
2 string   aligned varying bit(2359296);

dcl 1 value
 2 mode    unal bit(18),
 2 reference unal bit,
 2 list_reference unal bit,
 2 external  unal bit,
 2 constant  unal bit,
 2 used     unal bit,
 2 input    unal bit,
 2 output   unal bit,
 2 user_mode unal bit,
 2 restricted unal bit,
 2 unused   unal bit(5),
 2 dims     unal bit(4); /* number of dimensions in list */

dcl any_model
  integer_model based pointer aligned,
  real_model   based fixed binary(35,0),
  boolean_model based bit(1) aligned,
  gate_model   based bit(36) aligned,
  procedure_model based structure,
  text        pointer,
  stack       pointer,
  creation    fixed bin(71),
  symbol_model based character(256) varying aligned,

dcl list_model
  bound      unal fixed bin(17); /* current size in words */
 2 size      unal fixed bin(17); /* maximum size in words */
 2 element   unaligned ptr; dim(size refer(list_model.bound)),

dcl ext_list_model
  bound      unal fixed bin(17); /* current size in words */
 2 size      unal fixed bin(17); /* maximum size in words */
 2 ext_element dim(size refer(ext_list_model.bound)),
 3 offset    unal fixed bin(17),
 3 unused    unal bit(18),
  ref_model  based aligned ptr; /* internal reference */

dcl ext_ref_model based structure,
Appendix B. Multics Seal Code Generator Program.

```
2 unique_id bit(36),
2 offset unal fixed bin(17),
2 unused unal bit(18),

asize fixed bin; /* allocation size of structure elements */

dcl any_type
  any_type initial(1),
  boolean_type initial(2),
  gate_type initial(3),
  integer_type initial(4),
  proc_type initial(5),
  real_type initial(6),
  symbol_type initial(7); fixed static;

dcl seal_parser_
  entry(ptr, entry);
  entry(ptr, show);
  entry(ptr, table);
  entry(ptr, name,
    void quad (arg 1),
    void other (arg 2),
    void sym (arg 3),
    void bind (arg 4),
    void effect (arg 5)
  );

  entry(ptr, code,
    void name (arg 1),
    void number (arg 2),
    void decl (arg 3),
    void usage (arg 4),
    void call (arg 5)
  );

  entry(ptr, error);

  /* arg 1 (input)    work seg ptr */
  /* arg 2 (input)    error printing routine */

  /* arg 1 (input)    work seg ptr */
  /* arg 2 (input)    error printing routine */

  /* arg 1 (input)    work seg ptr */
  /* arg 2 (input)    string to be printed */

  /* arg 1 (input)    work seg ptr */
  /* arg 2 (input)    line number */
  /* arg 3 (input)    first source character index */
  /* arg 4 (input)    last source character index */
```
Appendix B. Multics Seal Code Generator Program.

/* program */

ws = workseg_ptr;
text = storage.output_seg;

/* create scratch segments */
scratch_index = 0;
if storage.debug_option
then call establish_cleanup_proc(clean_up);

defs = create_scratch_("defs");
defs_rel = create_scratch_("defs_rel");
link = create_scratch_("link");
link_rel = create_scratch_("link_rel");
symb = create_scratch_("symb");
symb_rel = create_scratch_("symb_rel");
text_rel = create_scratch_("text_rel");
block(1).stack_base = create_scratch_("stack_template");

/* Initialize text section */
text_ic = 1;

/* Initialize definitions section */
defs_ic = 0;
defs = pointer(defs, defs_ic);
defs_rel = pointer(defs_rel, defs_ic);
defs_ic = size(defs->definition_header);

zero_definition = bit(defs_ic, 18); /* all-zero word for list termination */
defs_ic = defs_ic + 1;
defs->definition_header.definition_list = bit(defs_ic, 18); /* N3: knows that first def is next */
defs_rel->definition_header.definition_list = rel_defs;
defs->definition_header.next,
defs->definition_header.ignore = "1"bit;

/* generate segname definition */
previous_definition = null;
call allocate_definition;
segname_definition = rel(defs);
name_definition = store_def(storage.segname->vstring);
Appendix B. Multics Seal Code Generator Program.

```c
defs->definition.value = zero_definition;
defs_rel->definition.value = rel_defs;
defs->definition.class = "011"b;
/* class 3 is a segname definition */
defs->definition.symbol = name_definition;
defs_rel->definition.symbol = rel_defs;
defs->definition.segname = bit(defs_lc, 18); /* for class 3 this is the segname thread */
defs_rel->definition.segname = rel_defs;
/* for class 3 segname field is offset of first non-class-3 definition */

/* Initialize symbol section */
sym_lc = 0;
/* Initialize linkage section */
link_lc = size(link->linkage_header);
/* Initialize variables */
temporary_count = 0;
do i = L to bbound(operand[*], 1);
   operand(i) = 0;
end;
```

Page 60.
Appendix B. Multics Seal Code Generator Program.

create_scratch_1 proc(name) returns(pointer);

    dcl      name char(*) aligned,
             code fixed bin(35),
             p pointer;

call hcs_make_seagate("", "Seal","\$subsr("123456789",storage,invocation,1)","\$name!",", ",";100","p,code);
    if p = null
        then do:
            call con.err(code, my_name, "Trying to create scratch segment in process directory!");
            call print(112);  /* fatal error -- abort & unwind */
        end:

call hcs_truncate_seagate(p,0,code);

scratch_index = scratch_index + 1;
scratch(scratch_index) = p;
return(s);
end create_scratch_1;


clean_up: procedure;

    dcl      i fixed bin;

    do i = 1 to scratch_index;
        call hcs_truncate_seagate(scratch(i), 0, code);  
        call hcs_terminate_noname(scratch(i), code);
    end;
    return;
end clean_up;

Page 61.
Appendix B. Multics Seal Code Generator Program.

/* Allocate all constants in the text section */

   do i = 3 to 1 by -1;
      do s = constant_list() repeat symbol(s).next while(s != 1);
         vp = symbol(s).value;

         if symbol(s).passed_as_arg
            then do:
               string(value_temp) = "g"b;
               value_temp.mode = allocate_mode(symbol(s).mode);
               value_temp.constant = "1"b;
               value_temp.inout = "1"b;

               if i = 2    /* even value offset required on this list */
                  then if text_ic = 2 * divide(text_ic, 2, 13, 0)
                     then text_ic = text_ic + 1;

                  text = pointer(text, text_ic);
                  text_rel = pointer(text_rel, text_ic);
                  text_ic = text_ic + 1;
                  string(text->value) = string(value_temp);
                  text_rel->relocation.left = rel_text;

               end;

         if i = 2
            then text_ic = 2 * divide(text_ic + 1, 2, 13, 1);

         symbol(s).location = bit(text_ic, 13);
         symbol(s).left_relocation = rel_text;

      if symbol(s).mode = symbol_type
         then do:
            i = 2;
            n, i = length(vp->vstring) - 2;
            text = pointer(text, text_ic);
            text_ic = text_ic + 1;

            do while(i < n):
               n = index(substr(vp->vstring, i, n), "");

               if n = "
                  then n = n;

               text->vstring = text->vstring ! substr(vp->vstring, i, n);
               i = i + n + 1;
               i = i - n - 1;
               n = i - n;
Appendix B. Multics Seal Code Generator Program.

```plaintext
end:
text_ic = text_ic + divide(length(text->vstring) + 3, 4, 17, 0);
end:
else do:
  n = mode_length(symbols.mode);
p=inter(text, text_ic)->word_copy_image = vp->word_copy_image;
text_ic = text_ic + n;
end:
end:
```

Page 63.
Appendix B. Multics Seal Code Generator Program.

/* The SEAL stack frame is organized into the following units, in order:
   standard Multics stack frame header,
   seal_operators_workspace,
   an array of ITP/ITS pointers, one per allocated name,
   space for the value associated with each ITP pointer,
   space for compiler temporaries.

   The format of the stack template (stored at the base of the text section) is, in order:
   template ITP/ITS pointer (initialized with fault_tag_3's),
   value headers

   The pointers & value headers are in 1 to 1 correspondence. The template pointers are copied
   as a block into the runtime frame, and then the value headers are individually
   copied into the runtime frame.

   The storage allocator must make two passes—one to compute the offsets of the pointer
   array, and one to compute the offsets of the values. The pointers are kept separate only to improve
   the runtime copy operation.

   ITP pointers are used for values on the stack (this lets us compute the offset at runtime and still
   fault on undefined values), while ITS pointers are used for values in Temporary storage
   (since the offset must be computed at runtime anyway). An operator call is generated to allocate
   values in Temporary storage. */

   do b = 1 to last_block;
      /* Begin pass one. */
      stack_offset = stack_frame_first_available_location;
      stack_base = block(b).stack_base;
      do i = 1 to bound(block(b)).identifiers(*) do;
         do s = block(b).identifiers(i) repeat symbol(s).next while(s~2);
            string(value_temp) = "$\"b;
            if symbol(s).referenced
               then if symbol(s).variable
               then do;
                  /* Allocate a parameter */
                  symbol(s).defined = "1"b;
                  /* suppress sxil/tsx; instruction */
                  position = binary(substr(symbol(s).location, 19, 18), 18) * 2;
                  symbol(s).location = bit(position, 18) \ arg_list_op_zero_indirect
                  end;
               else do;
                  /* Allocate a variable */
                  if "symbol(s).ref"
Appendix B. Multics Seal Code Generator Program.

then if "symbol(s).list_ref
  then if symbol(s).mode <= symbol_type
    then element_size = mode_length(symbol(s).mode);
  else do:
    value_temp.user_mode = "1";b;
    element_size = compute_size(symbol(s).mode);
  end;
  else do:
    value_temp.list_reference = "1";b;
    element_size = size(p->packed_ptr_model);
  end;
  else do:
    value_temp.reference = "1";b;
    element_size = size(p->ref_model);
  end;
if symbol(s).count >= 0
  then do:
    dims = symbol(s).count;
    if dims > max_no_dms
      then do:
        call print(i30); /* too many dimensions */
        dims = max_no_dms;
        symbol(s).count = max_no_dms;
      end;
      value_temp.dims = bit(binary(dims, 4), 4);
      symbol(s).runtime_allocate = "1";b;
      /* symbol(s).element_size = element_size; */
    end;
    value_temp.mode = allocate_mode(symbol(s).mode);
    /* Allocate name pointer on stack */
    symbol(s).location = bit(stack_offset, 16) +
      activation_record_up_zero_indirect:
    p = address(stack_base, stack_offset - stack_frame_first_available_location);
    p->temporary_name.value_header = string(value_temp);
    p->temporary_name.element_size = element_size;
    if symbol(s).mode > symbol_type
      then k = hound(even_offset_required(*), 1);
    else k = symbol(s).mode;
    p->temporary_name.make_offset_even = even_offset_required(*);
    p->temporary_name.runtime_allocate = symbol(s).runtime_allocate;
    stack_offset = stack_offset + size(p->temporary_name);
end;
else if symbol(s).constant & symbol(s).mode /= proc_type
  then do:
    /* Allocate a named constant */
  end;
)}
Appendix B. Multics Seal Code Generator Program.

\[
\begin{align*}
J &= \text{symbol(s).general;} \\
\text{symbol(s).location} &= \text{symbol(j).location;} \\
\text{symbol(s).left_relocation} &= \text{symbol(j).left_relocation;}
\end{align*}
\]

end;

end;

\text{block(b).stack_end} = \text{stack_offset} - \text{stack_frame_first_available_location;}

/* Begin pass two. */

\text{value_offset, i, stack_end} = \text{stack_offset;}

\text{stack_offset} = \text{stack_frame_first_available_location;}

do while(\text{stack_offset} < \text{stack_end};

\text{name} = \text{addrel(stack_base, stack_offset - stack_frame_first_available_location);}

\text{p} = \text{addrel(stack_base, i - stack_frame_first_available_location);}

\text{string(p->\text{value})} = \text{name->temporary_seal_name.value_header;}

\text{element_size} = \text{name->temporary_seal_name.element_size;}

\text{if name->temporary_seal_name.runtime_allocate then do:}

\text{string(name->its_model)} = \text{its_to_temporary_storage_ft3 || \text{bit(binary(1, 18), 18);}}

\text{name->its_model.segment} = \text{bit(binaryrel(p, 15), 15);}

end;

\text{else do:}

\text{if name->temporary_seal_name.make_offset_even then if value_offset = 2 * divide(value_offset, 2, 18, 0) then do:}

/* this value must begin on an even boundary, but the value header for it must be in the preceding word, which is odd. Since the offset is now even, add 1 to get the desired odd boundary for the value header. */

\text{value_offset} = \text{value_offset + 1;}

end;

\text{value_offset} = \text{value_offset + 1;}

/* space for value header */

\text{string(name->itp_model)} = \text{its_to_activation_record_ft3 || \text{bit(value_offset, 16);}}

\text{value_offset} = \text{value_offset + element_size;}

end;

\text{stack_offset} = \text{stack_offset + size(name->itp_model);}

\text{i} = \text{i + size(p->\text{value});}

end;

\text{block(b).temporary_end} = \text{value_offset;}

\text{block(b).stack_size} = \text{value_offset;}

\text{block(b).stack_base} = \text{addrel(stack_base, 2 * divide(i-stack_frame_first_available_location+1, 2, 18, 0));}

end;

s = last_symbol;

Page 66.
Appendix B. Multics Seal Code Generator Program

/* This function encodes the mode of a variable into a (hopefully) unique bit string. It calls itself recursively to process components of user-defined modes. */

allocate_mode:
  procedure(mode_index) recursive returns(bit(18) unaligned):

dcl (c, cm, cl, i, m, mode_index) fixed,
  (co, a) dfr,
  mode_offset bit(18):

  /* Codes 1 through 7 are reserved for builtin modes */

dcl (left_parn init("1000"b),
       right_parn init("1001"b)) bit(4) static aligned;

  /* The list, list_ref, and # of dims must still be encoded. This will take 6 more
   bits for each element (one for list, one for list_ref, and four for # dims (15 is maximum # dim)

  m = mode_index:

  /* If this Mode has already been encoded, return its offset. */

  if symbol(m).location == ""b
    then return(substr(symbol(m).location,1,18)):

  /* To encode this mode, first encode each of its components */

  do c = symbol(m).general repeat symbol(c).general while(c<0):
    cm = symbol(c).mode:
    if symbol(cm).location == ""b
      then mode_offset = allocate_mode(cm):
    end;

  /* If this mode has no sub-components it must be a builtin mode. Encode it now. */

  if symbol(m).general == 0
    then if symbol(m).location == ""b
        then if cm:
          then text = pointer(text, text_in):

          text_rel = pointer(text_rel, text_ic):

          text_ic = text_ic + 2:

          text_rel->relocation.left = rel_text:

          text->mode_word = text_ic:

          text->mode.length_reg = 3l_mod:

          text->mode.string = bit64aray(m,4),4):

          text_ic = text_ic + 1:

          symbol(m).location = rel(p):

          text_rel->relocation.left = rel_text:
          text->mode_word = text_ic:
          text->mode.length_reg = 3l_mod:
          text->mode.string = bit64aray(m,4),4):

          text_ic = text_ic + 1:

          symbol(m).location = rel(p):
Appendix B. Multics Seal Code Generator Program.

symbol(m).left_relocation = rel_text;
end;
else:
    /* already encoded */
else do
/* This node has sub-components. Concatenate their mode strings together to form
the mode string of the parent. */
    p = pointer(text, text_ic);
text_rel = pointer(text_rel, text_ic);
text_ic = text_ic + 2;
text_rel->relocation.left = rel_text;
p->mode.word = text_ic;
p->mode.length_reg = ql_mod;
    /* really EIS 1 register code */
p->mode.string = left_parn;
bitlen = length(symbol(m).name->vstring)*9;
p->mode.string = p->mode.string ++ addrel(symbol(m).name1)->image;
do c = symbol(m).general repeat symbol(c).general while(c="=3");
    cm = symbol(c).mode;
    cp = pointer(text, substr(symbol(cm).location1,18));
p->mode.string = p->mode.string ++ cp->mode.string;
    bitlen = length(symbol(c).name1->vstring) * 9;
p->mode.string = p->mode.string ++ addrel(symbol(c).name1)->image;
end;
p->mode.string = cm->mode.string ++ right_parn;
text_ic = text_ic + divide(length(p->mode.string)+35,36,17,1);
symbol(m).location = rel(p);
symbol(m).left_relocation = rel_text;
end;
return(rel(p));
end allocate_node;

Page 59.
Appendix B. Multics Seal Code Generator Program.

/* compile the program */
do ir = 1 to storage.last_ir:
    opcode = macro(ir).opcode;
    operand(1) = macro(ir).operand1;
    operand(2) = macro(ir).operand2;
    output = macro(ir).output;
    count, macro(ir).count = macro(ir).count + 1;
    /* Normalize the reference counts. */
    if opcode = use
        then do:
            /* use just causes the output of its argument (another macro) to be "used" as the output
             of this macro. Just share its temporary/register location, and decrement reference count. */
            i = operand(1);
            output = macro(ir).output;
            count, macro(ir).count = macro(ir).count - 1;
        end;
    else if macro(ir).opcode = line_number
        then if stop_on_line = operand(1)
            then do:
                call loa_("stop on line ", stop_on_line);
                call debug;
            end;
        else:
            else do:
                /* Get information about this triple operator */
                operator_info_ptr = address(address_patterns$operator_table), opcode);
                if stop_on_macro = ir
                    then do:
                        call loa_("stop on macro ", ir);
                        call debug;
                    end;
                /* Get the nodes of each operand and type check them in the table. */
                type1 = get_type(operand(1), operator_info.type1);
                type2 = get_type(operand(2), operator_info.type2);
                operands = type1 ! type2;
                operand_info_ptr = pointer(operator_info_ptr, operator_info.offset);
                done, found = "0"b;
                do i = 1 to operator_info.length while("found");
                    if string(operand_semantics, operands(i)) = operands
                        then found = "1"b;
            end;
Appendix B. Multics Seal Code Generator Program.

end;

if "found
then do:

/* One or more of the operands are of the wrong mode for this operator */

if type1 = user_mode_code ! type2 = user_mode_code
then call print(102);
else call print(101);
done = "1"b
/* Don't try to generate code */

end;

/* Get pointer to the code pattern for this operator-operand combination.

pattern_base = pointer(operator_info_ptr, operand_semantics.pattern_offset);

do pattern_ic = 1 by 1 while ("done"):

p = addr(pattern_base->pattern(pattern_ic));
if p->pattern_word.flag
then call interpret_pattern;
else do:

arg = p->instruction_word.arg;
if arg = "00"b

then do:

if p->pattern_word.flag
then do:

else do:

/* Until a better way is found, assume "00"b is OK. */

call get_address(operand(binary(arg, 2)), "1"b);

end;

end;

end;

end;

end;

end; /* And do the next trial */
Appendix B. Multics Seal Code Generator Program.

/* This procedure encodes the macro's operands so that they table of allowable operand types may be searched. */

get_type proc (macro_operand, type) returns (bit9 aligned);

dcl (macro_operand, mop) fixed bin,
    type bit (3) unaligned;

dcl (index_ initial ("001"b),
    constant_ initial ("010"b),
    symbol_ initial ("011"b)) bit (3) aligned static;

/* program */

if type = constant_
    then return (constant_code);

if type = symbol_
    then return (no_check_code);

if type = index_
    then return (no_check_code);

mop = macro_operand;

if mop = 0
    then do;
        /* no operand */
        return (no_check_code);

    end;

if mop > 0
    then do;
        /* operand is a symbol node */
        if symbol (mop).count = 0
            then return (list_code);

        if symbol (mop).ref
            then return (ref_code);

        if symbol (mop).node = symbol_type
            then return (user_node_code);

        endif;

    end;

Page 72.
Appendix B. Multics Seal Code Generator Program.

```plaintext
return(bit(binary(symbol(mop).mode, 9), 9));
end;
else do:
    /* operand is the output of another macro */
    mop = -mop;
    if macro(mop).count <= 0 then call print(133); /* reference count too low */
    mop = macro(mop).output;
    if mop = 0 then call print(104); /* no output available */
    if mop > 0 then do:
        /* macro output is in a register */
        return(bit(binary(register(-mop).mode, 9), 9));
    end;
    /* macro output is in a temporary */
    return(bit(binary(temporary(mop).mode, 9), 9));
end;
end get_type;
```
Appendix B. Multics Seal Code Generator Program.

/* This subroutine is called every time the code generator wishes to address a macro operand (which may be a constant, symbol, temporary, or register). */

get_address: proc(macro_operand, direct_modifier_allowed);

/*
(input) macro_operand macro operand to be addressed.
(output) address address of the operand, in a form suitable for a 6180 instruction.
(output) left_relocation relocation code for this address */

dcl (i, macro_operand, opnd, tmp) fixed bin,
    direct_modifier_allowed bit(1) aligned parameter,
    p pointer;

/* program */

opnd = macro_operand;

if opnd = 0
    then do:
        /* no operand available to address */
        call print(125);
        address = (18)"1":
        left_relocation = "0":
        return;
    end;

if opnd > 0
    then do:
        /* operand is a symbol node */

        if symbol(opnd).label = "symbol(opnd).defined"
            then do:
                /* push this address onto the usage chain of the label */
                address = symbol(opnd).location
                symbol(opnd).location = bit(text_ic, 16);

                /* use special relocation code so that undefined labels can be diagnosed later. */
                left_relocation = (6)"1" bit(binary(opnd, 12), 12);
            return;
        end;

        else if symbol(opnd).constant & direct_modifier_allowed
            then if symbol(opnd).mode = boolean_type & symbol(opnd).mode = integer_type
                then do:
Appendix A. Multics Seal Code Generator Program.

/* try to use a du or dl modifier */
p = symbol(opnd).value;
if p->word.left_half == "j" then do:
  address = p->word.right_half (12)"0b // dl_mod;
  left_relocation = rel_absolute;
  return;
end:
else if p->word.right_half == "0"b then do:
  address = p->word.left_half (12)"0b // dl_mod;
  left_relocation = rel_absolute;
  return;
end;

/* normal case */
address = symbol(opnd).location;
left_relocation = symbol(opnd).left_relocation;
if address == "j"b then call print(110) /* undefined address */
return;
else do:
/* operand is the output of another macro */
l = opnd;
if macro(l).count > 0 then macro(l).count = macro(l).count - 1;
else do:
  /* reference count <= 0 */
call print(15);
  address = (18)"1b;
  left_relocation = "0"b;
  return;
end;
cond = macro(l).output;
if cond == 0 then do:
  /* macro has no output at this point */
  Page 75.
Appendix B. Multics Seal Code Generator Program.

call print(107);
address = (18)"1"b;
left_relocation = "0"b;
end;

if oopd > 0
then do:
/* output is in a temporary in storage */
address = temporary(opnd).location:
left_relocation = rel_absolute;
if macro(i).count = 0
then do:
/* this is the last reference to the temporary */
call free_temporary(opnd):
macro(i).output = 0;
end;
return;
end;
else do:
/* output is in a register: in order to provide an address,
the register will have to be stored into a temporary. */
temp = register(-opnd).contents;
call save_register(-opnd);
if temp > 0
then do:
/* register contained a symbol node */
address = symbol(temp).location;
left_relocation = symbol(temp).left_relocation;
end;
else do:
/* register contained a macro output */

oopd = macro(-opnd).output:
address = temporary(opnd).location:
left_relocation = rel_absolute;
end;
return;
end:
}

end get_address;

Page 76.
Appendix B. Multics Seal Code Generator Program.

/* This procedure is called to force a value out of a register. */
If the value is simply a loaded symbol, it is not stored, merely thrown away.
If the value is a macro output, it is stored in a temporary, and the macro changed to reflect this. */

save_register: proc(register_index);

/* program */

value = register(register_index).contents;

if value = empty_register
then return;

if value > 0
then do:
    register(register_index).mode = 0;
    register(register_index).contents = empty_register;
    return;
end;

if macro(-value).count > 0
then do:
    temporary_index = get_temporary(mode_length(register(register_index).mode));
    temporary(temporary_index).mode = register(register_index).mode;
    macro(-value).output = temporary_index;
    text = pointer(text, text+1);
    text+1 = string(text+1); temporary(temporary_index).location;
    text->instruction.opcode = store_op(register_index);
end;

register(register_index).mode = 0;
register(register_index).contents = empty_register;
return;

end save_register;
Appendix B. Multics Seal Code Generator Program.

interpret_pattern: proc;

/* automatic items */

dcl 1 fixed bin,
  a pointer,
  offset fixed hint(item),
  (arg1, arg2, arg3, p_op) fixed bin,
  (load_complement, not_flag) bit(1) aligned;

* constant items (unset : internal static initial) */

dcl load_complement_op dim(2) static bit(10) aligned
  initial( "0110111110"b, /* lca 335(0) */
        "0110111111"b), /* lca 336(0) */

load_op dim(20) static bit(10) aligned
initial( "0100111110"b, /* lda 235(0) */
        "0100111111"b, /* lda 236(0) */
        "1000111110"b, /* dfld 433(0) */
        "1000111111"b, /* dfld 434(0) */
        "1110110000"b, /* lxl0 720(0) */
        "1110110010"b, /* lxl1 721(0) */
        "1110110001"b, /* lxl0 722(0) */
        "1110110011"b, /* lxl1 723(0) */
        "1110110002"b, /* lxl0 724(0) */
        "1110110012"b, /* lxl1 725(0) */
        "1110110003"b, /* lxl0 726(0) */
        "1110110013"b, /* lxl1 727(0) */
        "1110110004"b, /* eop3 350(0) */
        "3110110010"b, /* eop1 351(1) */
        "3110110011"b, /* eop2 352(0) */
        "3110110012"b, /* eop3 353(1) */
        "3110110013"b, /* eop4 354(0) */
        "3110110014"b, /* eop5 355(1) */
        "3110110015"b, /* eop6 372(0) */
        "3110110016"b, /* eop7 373(1) */

(tro  initial("1110010000"b, /* 71(0) */
  ora  initial("110111100"b, /* 275(0) */
  rls  initial("100111110"b, /* 431(0) */
  stag  initial("111011011"b, /* 757(0) */
  sx1  initial("100110001"b) /* 441(0) */
bit(36) static aligned,

nop_instruction bit(36) aligned static initial("00000000000000000000061001000000111"b), /* 300000 311307 */
  is_zero fixed bin initial(0) static;

Page 78.
Appendix B. Multics Seal Code Generator Program.

/* program */
if p->pattern_word.index_with_arg1
then arg1 = operand(binary(p->pattern_word.arg1, 8));
else arg1 = binary(p->pattern_word.arg1, 8);

if p->pattern_word.index_with_arg2
then arg2 = operand(binary(p->pattern_word.arg2, 8));
else arg2 = binary(p->pattern_word.arg2, 8);

arg3 = binary(p->pattern_word.arg3);
p_op = binary(p->pattern_word.pattern_op);
go to operator(p_op);

operator(1):
  /* LOAD - Register, Value */
operator(2):
  /* LOADC - Register, Value */
if p_op = 1
then load_complement = "0";
else load_complement = "1";

/* LOAD places values in registers. As far as the machine state is concerned, it is a copying
operation -- the machine state does not indicate that a symbol node has been LOADED; the
assumption is that all patterns change the value in the register once it is LOADED.

Triple outputs (and associated temporaries) are kept track of by the machine state however.
So before LOADING, save the old register contents, if necessary. */
if register(arg1).contents = arg2
then do:
  /* value to be loaded is already in the register */
  1 = register(arg1).contents;
  if 1 > 0 /* decrement reference count because we have */
then macro().count = macro().count - 1; /* made arg1 addressable */
  return;
end;
else if arg1 = A_reg
then call save_register((EAQ_reg));
else if arg1 = Q_reg
then call save_register((EAQ_reg));
else if arg1 = EAQ_reg
then do:
call save_register((A_reg));
call save_register((Q_reg));
end;
call save_register(arg1);

Page 79.
Appendix B. Multics Seal Code Generator Program.

call get_address(arg2, (arg1 < AP_reg));
if arg2 > 0 & arg1 < AP_reg
then if symbol[arg2].constant
then if symbol[arg2].mode = integer_type
then do:
  /* try to generate Ic Q N, di for label N */
  q = symbol[arg2].value;
  l: q->word.le:_.half = (18)"1"b
  then do:
    address = bit(binary(~ q->integer_model, 18), 18) "12"b
    left_relocation = rel_absolute;
    load_complement = "load_complement:
  end;
end:
text = pointer(text, text_ic);
text_rel = pointer(text_rel, text_ic);
text_ic = text_ic + 1;
string(text->instruction) = address;
if load_complement
then text->instruction.opcode = load_complement_op(arg1);
else text->instruction.opcode = load_op(arg1);
text_rel->relocation.left = left_relocation;
return;
operator(2): /* INREG - Register, Value, failure label */
operator(1): /* NOT_INREG - Register, Value */
if op == 2
  then not_flag = "1"b;
else not_flag = "0"b;
if (register(arg1).contents = arg2) = not_flag
then pattern_ic = pattern_ic + arg3 - 1;
  /* increment interpretation loop index to failure label - 1 */
  /* (because do loop will add 1 back) */
else do:
  j = -register(arg1).contents;
  if j > 0
then macro(j).count = macro(j).count - 1;
  /* made arg1 addressable */
end;
return;
Appendix B. Multics Seal Code Generator Program.

operator(3):
    j = register(arg1).contents;
    if j < 3
        then do
            if macro(-j).count > 3
                then call print(139);
                macro(-j).output = 0;
            end;
            if macro(ir).count > 0
                then do
                    register(arg1).mode = arg2;
                    register(arg1).contents = -ir;
                    macro(ir).output = -arg1;
                end;
        end;
    return;

operator(4):
    d = arg1;
    return;

operator(5):
    call get_address(arg2, "C"o);
    text = pointer(text, text_ic);
    text_rel = pointer(text_rel, text_ic);
    text_ic = text_ic + 1;
    string(text->instruction) = address;
    text->instruction.opcode = load_os(BP_reg1);
    text_rel->relocation.left = left_relocation;
    return;

operator(7):
    call save_register(arg1);
    return;

operator(9):
    l = macro(-arg2).output;
    offset = binary(substr(symbol(1).location, 1, 18)) + arg1;
    text = pointer(text, offset);
    text_rel = pointer(text_rel, offset);
    string(text->instruction) = bit(text_ic, 18);
    text->instruction.opcode = tra;
    text_rel->relocation.left = symbol(1).left_relocation;
    return;

operator(10):
    s = s + 1;
    symbol(s).location = bit(text_ic, 18);
Appendix B. Multics Seal Code Generator Program.

```c
symbol(s).left_relocation = rel_text;
macro(ir).output = s;
text = pointer(text, text_ic);
Text_rel = pointer(text_rel, text_ic);
string(text->instruction) = bit(text_ic, 18);  // generate tra *,qi */
text_ic = text_ic + 1;
text->instruction.opcode = tra;
text->instruction.tag = qi_mod;
text_rel->relocation.left = rel_text;
text_ic = text_ic + argl;
return;

operator(11):  /* FILL_USAGE - index of label */
do i = 1 to hbound/register(*), 1;  
call save_register();
end;
offset = binary(substr(symbol(arg1).location, i, 16));
if offset ~= 0 /* has the label been used yet? */
then do:
text = pointer(text, offset);
text_rel = pointer(text_rel, offset);
do offset = binary(text->instruction.offset, 15) repeat binary(text->instruction.offset, 15)
while(offset ~= 0):
text->instruction.offset = bit(binary(text_ic, 15), 15);
text_rel->relocation.left = rel_text;
text = pointer(text, offset);
text_rel = pointer(text_rel, offset);
end:
text->instruction.offset = bit(binary(text_ic, 15), 15);
text_rel->relocation.left = rel_text;
end:
symbol(arg1).defined = "1b;  
symbol(arg1).location = bit(text_ic, 16);  
symbol(arg1).left_relocation = :if_text;
if "1b then call flush_all_temporaries;  /* TURNED OFF */
return;

operator(12):  /* NOP - no arguments */
text = pointer(text, text_ic);
text_ic = text_ic + 1;
string(text->instruction) = nop_instruction;
return;

operator(13):  /* GEN_LINK - index of symbol node */
symbol(arg1).location = bit(link_ic, 16);  
linkage_section.up_zero_indirect;
symbol(arg1).left_relocation = rel_link15;
link = pointer(link, link_ic);
```

Page 82.
Appendix B. Multics Seal Code Generator Program.

link_rel = pointer(link_rel, link_ic);

/**************** Form the negative offset of this link in the linkage section. */
/**************** in two's complement form, naturally. */
substring(link->its_model, 1, 18) = bit(binary(link_ic - 1, 18), 18);
link->relocation.left = rel_negative_link18;
link_ic = link_ic + size(link->its_model);
link->its_model.its = ft2_mod;
link->its_model.offset = bit(defs_ic, 18);    /* offset of expression_word */
link->its_model.offset = rel_defs;
defs = pointer(defs, defs_ic);
defs_rel = pointer(defs_rel, defsc_ic);
defs_ic = defsc_ic + size(defs->expression_word);
defs->expression_word.type_pair = bit(defs_ic, 18);    /* offset of type_pair */
defs_rel->expression_word.type_pair = rel_defs;
defs = pointer(defs, defsc_ic);
defs_rel = pointer(defs_rel, defsc_ic);
defs_ic = defsc_ic + size(defs->type_pair);
defs->type_pair.type = "3003000300000010"
;    /* type 4 is segname$entryname */
defs->type_pair.segname,
defs->type_pair.entryname = store_def(symbol(arg1).name->vstring);
defs->type_pair.segname,
defs_rel->type_pair.entryname = rel_defs;
return;

operator(14)    /* GEN_DEF - index of symbol node */

   text_ic = text_ic + 1;    /* leave space for entry_sequence structure */
symbol(arg1).location = bit(text_ic, 18);
symbol(arg1).left_relocation = rel_text;
block().entry_location = text_ic;

   call allocate_definition;
   name_definition = store_def(symbol(arg1).name->vstring);
defs->definition.value = bit(text_ic, 18);
defs_rel->definition.value = rel_text;

   text = pointer(text, text_ic - 1);    /* points to entry_sequence structure */
text->entry_sequence.entry_definition = rel_defs;
Appendix D. Multics Seal Code Generator Program.

```
if symbol(arg1).external
    thendefs->definition.entry = "1"b;
defs->definition.retain = "1"b;
defs->definition.class = "303"b;
/* class 0 is a value relative to the text section */
defs->definition.symbol = name_definition;
defs_rel->definition.symbol = rel_defs;
defs->definition.segname = segname_definition;
defs_rel->definition.segname = rel_sets;
return;

operator(15):
    /* FILL_LIST - index of expression to put on list */
    return;

operator(16):
    /* ALLOCATE_LIST */
    return;

operator(17):
    /* ARG_PTR - Register, Index of arg_list macro */
    text = pointer(text, text_ic);
    text_ic = text_ic + 1;

call get_address(arg2, "3"b);
position = binary(substr(address, 4, 15), 15) + 2 * argument_index;
address = bit(position, 16) + activation_record_up_zero;
string(text->instruction) = address;
text->instruction.opcode = store_op(arg1);
argument_index = argument_index + 1;
return;

operator(18):
    /* ALLOCATE_ARG_LIST - number of arguments */
    macroin.output = get_temporary(2 * (arg1 + 1));
    argument_index = 1;
    text = pointer(text, text_ic);
    text_ic = text_ic + 1;
    /* construct an 8-bit number which is 2 * number of args. */
    /* The fld instruction will left-shift this number 3 places into the au, which is where we want it. */
    substr(string(text->instruction), 4, 14) = bit(binary[2 * arg1, 9], 8);
    text->instruction.opcode = fld;
    text->instruction.tag = dl_mod;
    text = pointer(text, text_ic);
    text_ic = text_ic + 1;
    text->instruction.offset = "00000000000100”b; /* 4 */
```
Appendix B. Multics Seal Code Generator Program.

```c
text->instruction.opcode = orn;
text->instruction.tag = al_mod;
text = pointer(text, text_ic);
text_ic = text_ic + 1;
string(text->instruction) = temporary(macro(ir), output).location | activation_record_up_zero;
text->instruction.opcode = start;
operator(19):
   /* CALL */
operator(20):
   /* TYPE_CHECK */
return;
operator(21):
   done = "1"b;
return;
operator(22):
   /* MODE_SELECT */
return;
operator(23):
   pattern_ic = pattern_ic + arg3 - 1;
   /* GOTO */
   /* Increment loop index (do loop will add 1 back... */
return;
operator(24):
   /* SET_STACK_SIZE */
   text = pointer(text, block(b).entry_location);
   offset = block(b).stack.size;
   offset = 16 * divide(offset + 15, 16, 16, 0); /* make stack size mod 16 */
   text->instruction.offset = bit(binary(offset, 15), 15);
return;
operator(25):
   /* DEFINE - Value */
   /* try to eliminate sxl1 instruction */
if arg1 > 0
   then if "symbol(arg1).defined
   then if before_first_flowchange
   then "symbol(arg1).defined = "1"b;
   else return;
call get_address(arg1, "0"b);
substr(address, 31, 8) = no_mod;
text = pointer(text, text_ic);
text_ic = text_ic + 1;
string(text->instruction) = address;
text->instruction.cocode = sxl1;
text_rel->relocation.left = left_relocation;
/* this register always contains an ITP tag */
```

Page 35.
Appendix B. Multics Seal Code Generator Program.

```c
return;
operator(26): /* ADD_REFERENCE - value */
    if arg1 = 0
        /* macro output */
        then if macro(-arg1).count <= 0
            then call print(111);
            else macro(-arg1).count = macro(-arg1).count + 1;
            return;
    operator(27): /* NOTE_FLOWCHANGE */
        before_first_flowchange = "O";
        return;
operator(28): /* IF_OPERAND = Value, check code */
    if arg2 = is_zero
        then do:
            if arg1 = 0
                then pattern lc = pattern lc + arg2 - 1;
                /* take failure label */
            end:
            return;
end interpret_pattern;
```
Appendix B. Multics Seal Code Generator Program.

/* This function is called to get an n-word temporary on the stack frame */

generate_temporary proc(n) returns(fixed);

dcl n fixed parameter,
       (even_offset, l, 1) fixed;

   if n > 2
      then l = 3;
   else l = n;

   if block(b).free_temps(l) == 0
      then do
          i = block(b).free_temps(l);
          block(b).free_temps(l) = temporary(i).mode; /* remove temporary(i) from list */
          return(i);
       end;
   else do
       i = block(b).temporary_end;
       if n = 2 * divide(i + 1, 2, 18, 0) /* if n is an even number */
          then do
              /* all temporaries which are an even length must be on an even boundary */
              even_offset = 2 * divide(i + 1, 2, 18, 0);
              if i = even_offset
                 then do
                     /* save extra word as a temporary */
                     temporary_count = temporary_count + 1;
                     if temporary_count > hbound(temporary(*)), 1
                        then call print(114); temporary(temporary_count).location = bitbinary(i, 18), 18): activation_record_uo_zero;
                     temporary(temporary_count).size = 1;
                     block(b).temporary_end = i + 1;
                     temporary(temporary_count).mode = block(b).free_temps(1);
                     block(b).free_temps(1) = temporary_count;
                  end;
      end;

   temporary_count = temporary_count + 1;
   if temporary_count > hbound(temporary(*)), 1
      then call print(114); temporary(temporary_count).location = bit(binary(b), temporary_end, 18): activation_record_uo_zero;
   temporary(temporary_count).size = n;
   block(b).temporary_end = block(b).temporary_end + n;
   block(b).stack_size = max(block(b).stack_size, block(b).temporary_end);
Appenlix B. Multics Seal Code Generator Program.

/* This subroutine is called to place a no-longer-needed stack temporary on the free list. */
free_temporary: proc(ft);
  dcl  ft fixed parameter;
      (l,n) fixed;
    n = temporary(ft).size;
    if n > 2
      then l = 3;
    else l = r;
    temporary(ft).node = block(b).free_temps(l);
    block(b).free_temps(l) = ft;
    return;
end free_temporary;

/* This subroutine is called to check that all temporaries have been freed at this point.
   If they haven't, then someone's reference count was too high */
flush_all_temporaries: proc;
/* Check to see that all temporaries have been freed, then reset temporary allocation offset + lists.
   Must also check to see that any values in the registers have ref count = 0 */
  temporary_count = 0;
  block(b).free_temps(1) = 0;
  block(b).free_temps(2) = 0;
  block(b).free_temps(3) = 0;
  block(b).temporary_end = block(b).stack_end;
end flush_all_temporaries;

Page 89.
Appendix B. Multics Seal Code Generator Program.

/* Copy the template stack frame for each block into the text section, and
   set the control word for each block to contain the offset & size of its template stack frame */

    do b = 1 to last_block;
   text_lc = 2 * divide(text_lc + 1, 2, 18, 0);
   text = pointer(text, text_lc);
   h = block(b).stack_end;
   n = h * 1.5;  /* copy both pointers and value headers */
   stack_base = block(b).stack_base;
   text->word_copy_image = stack_base->word_copy_image;
   do j = 1 to n - 1;
      /* fill in relocation for value headers */
      text_rel = pointer(text_rel, text_lc + j);
      text_rel->relocation.left = rel_text;
   end:

   j = block(b).entry_location $entry_control_word_offset;
   text = pointer(text, j);
   text_rel = pointer(text_rel, j);

   text->control_words.stack_offset = bit(text_lc, 18);
   text_rel->control_words.stack_offset = rel_text;

text->control_words.stack_size = bit(divide(1, 2, 18, 0), 13);
   text_lc = text_lc + n;

end:
/* The text section is finished, make it an even number of words long. Reference MPM Part III, Section 11.2 */
    text_ic = 2 * divide(text_ic + 1, 2, 18, 0);
/* Finish the definitions section. */
/* Generate symbol_table definition */
call allocate_definition;
    name_definition = store_def("symbol_table");
defs->definition.value = "C"b;
defs->definition.class = "C10"b; /* symbol block starts at symbol 0 */
defs->definition.symbol = name_definition;
defs->definition.segname = segname_definition;
defs->definition.segname = rel_defs;
defs->definition.symbol = rel_defs;
*/

Page 91.
Appendix B. Multics Seal Code Generator Program.

/* Procedure to store a character string in the definitions section and return an offset to it */

store_def: proc(def_string) returns(bit(16) unaligned);

dcl
  def_string char(*) varying aligned,
  d pointer,
  1 name aligned based,
  2 size aligned fixed bin(8),
  2 string unal character(n refer(name.size));

if def_string = ""
  then return(zero_definition);

  d = pointer(defs, defs_ic);
  defs_ic = defs_ic + divide(length(def_string) + 3 + 1, 4, 17, 1); /* length + roundup + 
    d->name.size = length(def_string);
  d->name.string = def_string;
  return(rel(d));

end store_def;

/* Procedure to allocate a definition block and thread it into the chain. */

allocate_definition: proc;

  if previous_definition = null
    then previous_definition->definition.forward = bit(defs_ic, 18);

  defs = pointer(defs, defs_ic);
  defs_rel = pointer(defs_rel, defc_ic);
  defs_ic = defs_ic + size(defs->definition);

  if previous_definition = null
    then defs->definition.forward = zero_definition;
  else defs->definition.forward = rel_defs;

  if previous_definition = null
    then defs->definition.backward = rel(previous_definition);
  else defs->definition.backward = zero_definition;
  defs_rel->definition.backward = reldefs;

  if previous_definition = null
    then previous_definition = def;

end allocate_definition;

Page 92.
Appendix B. Multics Seal Code Generator Program.

/* Fill in the linkage header. Reference MPM Part III, Chapter 11.4 */
/* The first few words of the linkage section have been reserved for the linkage_header */
/* SEAL does not use either the internal storage area, or the first-reference traps. */

link = pointer(link, 0);
link_rel = pointer(link_rel, 0);

link->linkage_header.declarations_offset = bit(text_ic, 18);  /* since defs follow text section in object */
link_rel->linkage_header.declarations_offset = rel_text;

link->linkage_header.links_offset = bit(binary(size(link->linkage_header), 18), 18);
link_rel->linkage_header.links_offset = rel_link18;

link->linkage_header.linkage_length = bit(link_ic, 18);
link_rel->linkage_header.linkage_length = rel_link18;

/* Create the symbol section. Reference MPM Part III, Chapter 11.5 */
/* N.B. Because all items in the symbol_block are relative to the base of the symbol_block, and not the symbol section, they have absolute relocation codes. The one exception is symbol_header.backpointer, which is relative to the base of the symbol section, and therefore has negative symbol relocation. */

symb_ic = 0;
symb = pointer(symb, symb_ic);
symb_rel = pointer(symb_rel, symb_ic);
symb_ic = symb_ic + sizesymbol_block);
symb->seal_symbol_block.act_version = 1;
symb->seal_symbol_block.identifier = "Symbtree";
symb->seal_symbol_block.gen_number = 1;  /* temp, for now */
symb->seal_symbol_block.gen_created = ador(seal_code_generator_symbol_table->std_symbol_header.gen_created;
symb->seal_symbol_block.object_created = storage_clock_time;
symb->seal_symbol_block.generator = "seal";

string(symb->seal_symbol_block.gen_version) = store_symbol(seal_version, 8);
string(symb->seal_symbol_block.user_id) = store_symbol(storage.user_id->vstring);
if storage.options == null
then string(symb->seal_symbol_block.comment) = store_symbol(storage.options->vstring);

symb->seal_symbol_block.text_boundary = "0000000000000000b";
symb->seal_symbol_block.stat_boundary = "0000000000000000b";
symb_ic = 2 * divide(symb_ic + 1, 2, 18, 0);  /* source_map must start on an even boundary */
symb->seal_symbol_block.source_map = bit(symb_ic, 18);  /* source_map is logically part of this symbol block */
symb_rel->seal_symbol_block.backpointer = rel_negative_symbol;
symb->seal_symbol_block.block_size = bit(binary(size(symb->seal_symbol_block), 18), 13);
/* Procedure to store a character string in the symbol section and return a "string pointer" to it. */
store_symbol: proc(symbol_string) returns(bit(36) unaligned);
dc1 symbol_string char(*) varying,
    s pointer,
    based_string char(length(symbol_string)) based aligned,
    1 string_pointer structure,
    2 offset   unal bit(18),
    2 size     unal bit(18);  
s = pointer(symb, symb lc);
s->based_string = symbol_string;
string_pointer.offset = bit(symb lc, 18);
string_pointer.size = bit(binary(length(symbol_string), 18), 18);
symb lc = symb lc + divide(length(symbol_string) + 3, 4, 17, 0);
return(string(string_pointer));
end store_symbol;

Appendix B. Multics Seal Code Generator Program.
Appendix B. Multics Seal Code Generator Program.

/* Create the source map. */

symb = pointer(symb, symb_lc);
symb->source_map.version = 1;
*/ the order of the next 3 statements is critical! */
n, symb->source_map.number = 1;
symb_lc = symb_lc + size(symb->source_map);
*/ this one sets n to be the number of source files */
string(symb->source_map.map(1), pathname) = store_symbol(storage, pathname->vstring);
symb->source_map.map(1).uid = storage.uid;
symb->source_map.map(1).dtm = storage.dtm;

*/ Make all Sections an even number of words long. Reference MPH Part III, Sections 11.3 and 11.4 */

defs_lc = 2 * divide(defs_lc + 1, 2, 18, 0);
link_lc = 2 * divide(link_lc + 1, 2, 18, 0);
symb_lc = 2 * divide(symb_lc + 1, 2, 18, 0);

*/ Pack the relocation bits. All code which sets the relocation bits must come before this point. */

sp, symb = pointer(symb, 0);
symb->seal_symbol_block.mini_truncate,
symb->seal_symbol_block.maxi_truncate = bit(symb_lc, 18); /* save block & source_map, but not relocation bits */
saved_symb_lc = symb_lc;
*/ don't generate relocation bits for the relocation bits! */
l = 1;
du p = text_rel, defs_rel, link_rel, symb_rel;
p = pointer(p, 0);
go to section(1);

section(1):

m = text_ic;
sp->seal_symbol_block.rel_text = bit(symb_ic, 16);
go to begin_packing;

section(2):

m = defs_ic;
sp->seal_symbol_block.rel_def = bit(symb_ic, 18);
go to begin_packing;

section(3):

m = link_ic;
sp->seal_symbol_block.rel_link = bit(symb_ic, 18);
go to begin_packing;

section(4):

m = saved_symb_ic;

Page 95.
Appendix B. Multics Seal Code Generator Program.

```
begin_packing:
  k = 0;
  m = m + 2;
  symb = pointer(symb, symb_ic);
  symb->packed.relocation.dcl_version = 2;
  do j = 0 to m-1;
    rel_code = p->unpacked_relocation.half_word();
    if rel_code == "mb"
      then do:
        if k == 0
          then do:
            /* have "k" consecutive absolute half-words, try to use expanded abs...*/
            call expanded_absolute;
          k = 0;
        end:
      if substr(rel_code, 1, 6) = (6)"1b"
      then do:
        /* Address is on a usage chain which was never filled...undefined label. */
        /* Temporarily call loa_until seal error routine can take symbolic args */
        n = binary(substr(rel_code, 7, 12));
        call loa_("seal_code_generator": The label "---a" is undefined!",
                  symbol(n).name->vstring);
        call printf(113);
        text = pointer(text, divide(), 2, 18, 0));  /* 4 is even */
        string(text->instruction) = seal_patterns$undefined_label_instruction;
        rel_code = rel_absolute;
        end;
      symb->packed_relocation.string = symb->packed_relocation.string || substr(rel_code, 1, 5);
      end;
    else k = k + 1;
  end;
  if k == 0
    then call expanded_absolute;
  l = l + 1;
  symb_ic = symb_ic + divide(length(symb->packed_relocation.string) + 35, 36, 24, 0) + 1 + 1;
end;
  symb = pointer(symb, 0);
  symb->seal_symbol_block.block_size = bit(symb_ic, 16);
```

Page 96.
Appendix B. Multics Seal Code Generator Program.

/* This procedure uses the expanded absolute code when it is more efficient */

expanded_absolute: proc

    if k < 16
        then symb->packed_relocation.string = symb->packed_relocation.string || substr("0000000000000000", 1, k);
    else do
        do while(k > 1023);
            symb->packed_relocation.string = symb->packed_relocation.string || rel_exp_absolute || "11111111"b;
            k = k - 1023;
        end;
        symb->packed_relocation.string = symb->packed_relocation.string || rel_exp_absolute || blt(binary(k, 10), 10);
    end;

    return

end expanded_absolute:
Appendix B. Multics Seal Code Generator Program.

/* Create the object segment by concatenating the 4 sections. */
Reference MPM Part III, Section 11 */

/* The text section has been constructed in place in the object segment.
   Append the definitions, linkage and symbol sections to it.
   Finally, create the object map, and the object map pointer. */

The definitions section must start on an even boundary, and must be an even number of words long. (MPM III, 11.3) */

object_seg = pointer(text, text_ic);
defs = pointer(defs, 0);

n = defs_ic;
object_seg->word_copy_image = defs->word_copy_image;
object_ic = text_ic + defs_ic;

/* The linkage section must be an even number of words long. It automatically starts on an even boundary,
   because the definitions ends on an even boundary ... (MPM III, 11.4) */

object_seg = pointer(object_seg, object_ic);
link = pointer(link, 0);

n = link_ic;
object_seg->word_copy_image = link->word_copy_image;
object_ic = object_ic + link_ic;

/* The symbol section begins on an even boundary, and has been made an even number of words long,
   although the MPM does not require it. (MPM III, 11.5) */

object_seg = pointer(object_seg, object_ic);
symb = pointer(symb, 0);

n = symb_ic;
object_seg->word_copy_image = symb->word_copy_image;
object_ic = object_ic + symb_ic;

/* Now fill in the object map in place in the object segment. */
Reference MPM Part III, Section 11.6 */

map = pointer(object_seg, object_ic);
object_ic = object_ic + size(map->map);
map.obj_cl = 1;
map.identifier = "obj_map";
map.text_offset = "0";

Page 98.
Appendix B. Multics Seal Code Generator Program.

```c
map.text_length = bit(text_ic, 18);
map.def_offset = bit(text_ic, 13);
map.def_length = bit(defs_ic, 13);
map.link_offset = bit(binary(text_ic + defs_ic, 18), 18);
map.link_length = bit(link_ic, 18);
map.symo_offset = bit(binary(text_ic + defs_ic + link_ic, 18), 18);
map.symo_length = bit(symb_ic, 18);
map.format.relocatable
map.format.procedure
map.format.standard = "1"b;

/* Fill in the object map pointer in the very last word of the object segment. */
object_seg = pointer(object_seg, object_ic);
object_seg->object_map_offset = rel(map);
object_ic = object_ic + 1;
storage.output_seg_length = object_ic * 36;
storage.last_symbol = s;

/* That's all, folks! */
if storage.debug_option
then call clean_up;
return;

/* Temporary procedure to print error message and call debug. Will be replaced by final version later. */
print: proc(error_number);
dcl error_number fixed bin,
character builtin,
(debug, print) external entry options(variable);
    call print("error_messages", character(error_number), character(error_number));
call log("error","1r = "3"debug","1r);
call debug;
end print;
end seal_code_generator;
```
Appendix C. Multics Seal Code Generator Table.

" Interpretive table for Seal code generator. "
" Paul A. Green  
" 15 November 1972.  
" Last Modified by PS on 5/9/73  
"  
followon
name    seal_patterns_
segdef  operator_table

" This is a list of the possible operand types. 

| equ | 4,0 | No check |
| equ | 4,1 | Any       |
| equ | 8,2 | Boolean   |
| equ | 3,3 | Gate      |
| equ | 1,4 | Integer   |
| equ | 3,5 | Proc      |
| equ | 2,6 | Real      |
| equ | 5,7 | Symbol    |
| equ | 2,8 | Ref       |
| equ | i,9 | List      |
| equ | 3,10| Constant  |
| equ | J,11| User-defined mode |

" Constants taken from assembly of seal_operators_  

| bool | lt_to_bool,0 |
| bool | gt_to_bool,3 |
| bool | lt_or_eq_to_bool,6 |
| bool | gt_or_eq_to_bool,11 |
| bool | eq_to_bool,14 |
| bool | ne_to_bool,17 |
| bool | null_arg_list,24 |
| bool | null_pointer,26 |
| bool | vec,30 |
| equ | add_any_operator,vec+1 |
| equ | subtract any_operator,vec+2 |
| equ | divide any_operator,vec+3 |
| equ | multiply any_operator,vec+4 |
| equ | and any_operator,vec+5 |
| equ | or any_operator,vec+6 |
| equ | subscript_error_operator, vec+7 |
| equ | entry_operator,vec+8 |
| equ | complement any_operator,vec+9 |

Page 100.
Appendix C. Multics Seal Code Generator Table.

<table>
<thead>
<tr>
<th>Equ</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equ</td>
<td>negate_any_operator, vec+10</td>
</tr>
<tr>
<td>equ</td>
<td>return_operator, vec+11</td>
</tr>
<tr>
<td>equ</td>
<td>check_mode_operator, vec+12</td>
</tr>
<tr>
<td>equ</td>
<td>less_than_any_operator, vec+13</td>
</tr>
<tr>
<td>equ</td>
<td>integer_to_real, vec+14</td>
</tr>
<tr>
<td>equ</td>
<td>real_to_integer, vec+15</td>
</tr>
<tr>
<td>equ</td>
<td>call_operator, vec+16</td>
</tr>
<tr>
<td>equ</td>
<td>get_operator, vec+17</td>
</tr>
<tr>
<td>equ</td>
<td>out_operator, vec+18</td>
</tr>
<tr>
<td>equ</td>
<td>fixedoverflow_operator, vec+19</td>
</tr>
<tr>
<td>equ</td>
<td>return_registers_operator, vec+20</td>
</tr>
<tr>
<td>equ</td>
<td>undefined_label_operator, vec+21</td>
</tr>
</tbody>
</table>

"This list defines all of the pattern operators. It must correspond to a similar list in the program.

<table>
<thead>
<tr>
<th>Equ</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>equ</td>
<td>load_op, 1</td>
</tr>
<tr>
<td>equ</td>
<td>inreg_op, 2</td>
</tr>
<tr>
<td>equ</td>
<td>result_op, 3</td>
</tr>
<tr>
<td>equ</td>
<td>set_block_op, 4</td>
</tr>
<tr>
<td>equ</td>
<td>setup_op, 5</td>
</tr>
<tr>
<td>equ</td>
<td>loadc_op, 6</td>
</tr>
<tr>
<td>equ</td>
<td>erase_op, 7</td>
</tr>
<tr>
<td>equ</td>
<td>not_inreg_op, 8</td>
</tr>
<tr>
<td>equ</td>
<td>case_usage_op, 9</td>
</tr>
<tr>
<td>equ</td>
<td>allocate_case_op, 10</td>
</tr>
<tr>
<td>equ</td>
<td>fill_usage_op, 11</td>
</tr>
<tr>
<td>equ</td>
<td>nop_op, 12</td>
</tr>
<tr>
<td>equ</td>
<td>gen_link_op, 13</td>
</tr>
<tr>
<td>equ</td>
<td>gen_def_op, 14</td>
</tr>
<tr>
<td>equ</td>
<td>fill_list_op, 15</td>
</tr>
<tr>
<td>equ</td>
<td>allocate_list_op, 16</td>
</tr>
<tr>
<td>equ</td>
<td>arg_ptr_op, 17</td>
</tr>
<tr>
<td>equ</td>
<td>allocate_arg_list_op, 18</td>
</tr>
<tr>
<td>equ</td>
<td>call_op, 19</td>
</tr>
<tr>
<td>equ</td>
<td>type_check_op, 20</td>
</tr>
<tr>
<td>equ</td>
<td>return_op, 21</td>
</tr>
<tr>
<td>equ</td>
<td>mode_select_op, 22</td>
</tr>
<tr>
<td>equ</td>
<td>goto_op, 23</td>
</tr>
<tr>
<td>equ</td>
<td>set_stack_size_op, 24</td>
</tr>
<tr>
<td>equ</td>
<td>define_op, 25</td>
</tr>
<tr>
<td>equ</td>
<td>add_reference_op, 26</td>
</tr>
<tr>
<td>equ</td>
<td>note_flowchange_op, 27</td>
</tr>
<tr>
<td>equ</td>
<td>jperand_op, 28</td>
</tr>
</tbody>
</table>

"These symbols are used by the operand pattern.
Appendix C. Multics Seal Code Generator Table.

```
equ    is_zero, 0

" This is a list of "registers used by the code patterns.

equ    A_reg, 1
nequ   2, 2
nequ   AQ, 3
nequ   Any, 4
nequ   K0, 5
nequ   K1, 6
nequ   K2, 7
nequ   K3, 8
nequ   K4, 9
nequ   K5, 10
nequ   K6, 11
nequ   K7, 12
nequ   AP, 13
nequ   AB, 14
nequ   AP, 15
nequ   B, 16
nequ   P, 17
nequ   .B, 18
nequ   SP, 19
nequ   SB, 20

" This list defines the possible operand types of a
" triple opcode. They are derived from a similar
" list given in seal_operators.incl.pl1

" index type: implies mode check must be made on operand.
equ    i, 1

" optional index: implies operand may not be present.
equ  j, 1

" index, without mode check.
equ    l, 3

" constant: no mode check, operand type must be "C".
equ    1, 2

" symbol: no mode check, operand is symbol index.
equ    5, 3

" block: no mode check, operand is block index.
equ    5, 3
```

Page 102.
Appendix C. Multics Seal Code Generator Table.

These names are used to cause references to be made to triple operands at interpretation time. The upper 2 bits of the 15-bit address field of an instruction are reserved for this purpose. Therefore these names may be used in either a 15-bit or 18-bit address context.

```
bool arg1,020000
bool arg2,040000
```

These names are the 9-bit encodings of "arg1" and "arg2". They are used as arguments to the pattern operators.

```
bool arg1,401
bool arg2,402
```

The following instruction is inserted by the code generator into the compiled program whenever an attempt is made to reference an undefined label. It will cause a transfer to a diagnostic routine at execution time.

```
segdef undefined_label_instruction
undefined_label_instruction:
  tsx0 aplundefined_label_operator
```

```
BEGIN INCLUDE FILE ... stack_frame.incl.alm
```

```
equ stack_frame.prev_sp,16
equ stack_frame.condition_word,16
bool stack_frame.condition_bit,000100  (DL)
bool stack_frame.cross_ring_bit,001000  (DL)
equ stack_frame.next_sp,18
equ stack_frame.signaller_word,18
bool stack_frame.signaller_bit,001000  (DL)
equ stack_frame.return_ptr,20
equ stack_frame.entry_ptr,22
equ stack_frame.operator_ptr,24
equ stack_frame.lp_ptr,24
equ stack_frame.arg_ptr,26
equ stack_frame.on_unit_re1_ptr,20
equ stack_frame.operator_ret_ptr,31
equ stack_frame.translator_id
equ stack_frame.regs,32
equ stack_frame.min_length,40
```
Appendix C. Multics Seal Code Generator Table.

```
END INCLUDI$ FILE ... stack_frame.inc1a1m

This list defines offsets in a Seal program's stack frame.

equ seal_frame.saved_lb, 10

equ seal_frame.text_base_ptr, 40

equ seal_frame.linkage_ptr, 42

equ seal_frame.runtime_arglist, 44

equ seal_frame.runtime_arg1, 46

equ seal_frame.stack_ptr, 48

equ seal_frame.arg1, 50

equ seal_frame.arg2, 52

Offset of first allocated name pointer.

equ seal_frame.first_free, 56
```
### Appendix C. Multics Seal Code Generator Table.

This is a list of all of the triple operators produced by the parse, in the same order as in `seal_operators.incl.pl1`

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Type1</th>
<th>Type2</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused_0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>assign</td>
<td>i</td>
<td>i</td>
<td>10</td>
</tr>
<tr>
<td>add</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>sub</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>divide</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>mult</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>and</td>
<td>i</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>or</td>
<td>i</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>concatenate</td>
<td>i</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>concatenate_symbol</td>
<td>i</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>less_than</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>greater_than</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>less_or_equal</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>greater_or_equal</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>equal</td>
<td>i</td>
<td>i</td>
<td>8</td>
</tr>
<tr>
<td>not_equal</td>
<td>i</td>
<td>i</td>
<td>8</td>
</tr>
<tr>
<td>shape</td>
<td>i</td>
<td>i</td>
<td>8</td>
</tr>
<tr>
<td>exponentiate</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>complement</td>
<td>i</td>
<td>i</td>
<td>5</td>
</tr>
<tr>
<td>deref</td>
<td>i</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>negate</td>
<td>i</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>lock</td>
<td>i</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>unlock</td>
<td>i</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>test_lock</td>
<td>i</td>
<td>s</td>
<td>2</td>
</tr>
<tr>
<td>case_of</td>
<td>n</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>case_limit</td>
<td>n</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>case_jump</td>
<td>n</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>branch</td>
<td>s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>branch_true</td>
<td>s</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>branch_false</td>
<td>s</td>
<td>i</td>
<td>2</td>
</tr>
<tr>
<td>label</td>
<td>s</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>procedure</td>
<td>s</td>
<td>b</td>
<td>1</td>
</tr>
<tr>
<td>and</td>
<td>b</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>link</td>
<td>i</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>unused_34</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>element</td>
<td>i</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>list</td>
<td>n</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>arg</td>
<td>n</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>call</td>
<td>i</td>
<td>i</td>
<td>1</td>
</tr>
<tr>
<td>ret</td>
<td>i</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>reduce</td>
<td>s</td>
<td>i</td>
<td>1</td>
</tr>
</tbody>
</table>
Appendix C. Multics Seal Code Generator Table.

<table>
<thead>
<tr>
<th>Define</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>block</code>, 0, 0, 1</td>
<td></td>
</tr>
<tr>
<td><code>select</code>, s, i, 2</td>
<td></td>
</tr>
<tr>
<td><code>rop</code>, i, 0, 1</td>
<td></td>
</tr>
<tr>
<td><code>mode_select</code>, s, i, 1</td>
<td></td>
</tr>
<tr>
<td><code>line_number</code>, n, 0, 1</td>
<td></td>
</tr>
<tr>
<td><code>addr</code>, i, 0, 1</td>
<td></td>
</tr>
<tr>
<td><code>encode_dims</code>, i, n, 1</td>
<td></td>
</tr>
<tr>
<td><code>encode_value</code>, i, i, 1</td>
<td></td>
</tr>
<tr>
<td><code>arg_list</code>, n, 0, 1</td>
<td></td>
</tr>
<tr>
<td><code>encode_mode</code>, s, n, 1</td>
<td></td>
</tr>
<tr>
<td><code>split_prep</code>, i, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_52</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_53</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_54</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_55</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_56</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_57</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_58</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_59</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>current</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>errortrap</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>incolumn</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>infilemark</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>init</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>inititem</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>inlinemark</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>inpagemark</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>instream</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>linesize</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>outcolumn</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>outfilemark</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>outitem</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>outlinemark</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>outpagemark</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>outstream</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>pagesize</code>, o, i, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_78</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>unused_79</code>, 0, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>abs</code>, i, 0, 3</td>
<td></td>
</tr>
<tr>
<td><code>atan</code>, i, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>boolean</code>, i, 0, 3</td>
<td></td>
</tr>
<tr>
<td><code>ceil</code>, i, 0, 3</td>
<td></td>
</tr>
<tr>
<td><code>cos</code>, i, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>delete</code>, i, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>delete_dir</code>, i, 0, 0</td>
<td></td>
</tr>
<tr>
<td><code>detach</code>, i, 0, 0</td>
<td></td>
</tr>
</tbody>
</table>
Appendix C. Multics Seal Code Generator Table.

```
define       axp, i, 0, 0
#define        find, i, 0, 2
#define       floor, i, 0, 3
#define      integer, i, 0, 5
#define    isvoid, i, 0, 0
#define    length, i, 0, 1
#define       log, i, 0, 0
#define    log10, i, 0, 0
#define   rank, i, 0, 0
#define     real, i, 0, 5
#define       sign, i, 0, 3
#define        sin, i, 0, 0
#define      size, i, 0, 2
#define       sqrt, i, 0, 0
#define  symbol, i, 0, 7
#define       tan, i, 0, 0
#define       trunc, i, 0, 1
#define    unused_105, i, 0, 0
#define    unused_106, i, 0, 0
#define    unused_107, i, 0, 0
#define    unused_108, i, 0, 0
#define    unused_109, i, 0, 0
#define      create, i, 2
#define       is, i, 1
#define       get, i, 0, 1
#define      put, i, 0, 1
#define      void, i, 0, 1
#define       split, i, i, 0
#define    unused_116, i, 0, 0
#define    unused_117, i, 0, 0
#define    unused_118, i, 0, 0
#define    unused_119, i, 0, 0
#define      attach, i, i, 0
#define    createdir, i, i, 0
#define       edit, i, i, 3
#define       max, i, 1, 5
#define       min, i, 1, 5
#define       mod, i, 1, 5
#define     ~ename, i, i, 0
#define     ~ound, i, i, 2
```
Appendix C. Multics Seal Code Generator Table.

The following entries are the operand checking definitions. They are scanned top down until the first match is found on both operands. Then the associated pattern is interpreted by the code generator program.

<table>
<thead>
<tr>
<th>OPERAND_1, OPERAND_2, PATTERN_OFFSET</th>
</tr>
</thead>
<tbody>
<tr>
<td>unused_0: def 0</td>
</tr>
<tr>
<td>assign: def I,I, assign II</td>
</tr>
<tr>
<td>def I,R, assign IR</td>
</tr>
<tr>
<td>def R,I, assign RI</td>
</tr>
<tr>
<td>def R,R, assign RR</td>
</tr>
<tr>
<td>def 3,B, assign BB</td>
</tr>
<tr>
<td>def P,P, assign PP</td>
</tr>
<tr>
<td>def S,S, assign SS</td>
</tr>
<tr>
<td>def R,e, Re, assign Re Re</td>
</tr>
<tr>
<td>def _i Li, assign _i Li</td>
</tr>
<tr>
<td>def A,A, assign AA</td>
</tr>
<tr>
<td>add: def I,I, add II</td>
</tr>
<tr>
<td>def I,R, add IR</td>
</tr>
<tr>
<td>def R,I, add RI</td>
</tr>
<tr>
<td>def R,R, add RR</td>
</tr>
<tr>
<td>def A,A, add AA</td>
</tr>
<tr>
<td>sub: def I,I, sub II</td>
</tr>
<tr>
<td>def I,R, sub IR</td>
</tr>
<tr>
<td>def R,I, sub RI</td>
</tr>
<tr>
<td>def R,R, sub RR</td>
</tr>
<tr>
<td>def A,A, sub AA</td>
</tr>
<tr>
<td>divide: def I,I, divide II</td>
</tr>
<tr>
<td>def I,R, divide IR</td>
</tr>
<tr>
<td>def R,I, divide RI</td>
</tr>
<tr>
<td>def R,R, divide RR</td>
</tr>
<tr>
<td>def A, A, divide AA</td>
</tr>
<tr>
<td>multi: def I,I, multiply II</td>
</tr>
<tr>
<td>def I,R, multiply IR</td>
</tr>
<tr>
<td>def R,I, multiply RI</td>
</tr>
<tr>
<td>def R,R, multiply RR</td>
</tr>
<tr>
<td>def A, A, multiply AA</td>
</tr>
<tr>
<td>and: def 3,B, and BB</td>
</tr>
<tr>
<td>def A,A, and AA</td>
</tr>
</tbody>
</table>
Appendix C. Multics Seal Code Generator Table.

    ori    def    3,B,or_BB
    def    4,A,or_AA

    catenate: def    1,i,catenate_LiLi

    catenate_symbol: def    3,S,catenate_symbol_SS
    def    4,A,catenate_symbol_AA

"These triples perform the indicated relational test, and give a
"p olean output = "I" if the test was true.
"In all cases, the comparison is operand1 <= operand2.

    less_than: def    1,i,less_than_II
    def    1,R,less_than_IR
    def    2,i,less_than_RI
    def    2,R,less_than_RR
    def    4,A,less_than_AA

    greater_than: def    1,i,greater_than_II
    def    1,R,greater_than_IR
    def    2,i,greater_than_RI
    def    2,R,greater_than_RR
    def    4,A,greater_than_AA

    less_or_equal: def    1,i,less_or_equal_II
    def    1,R,less_or_equal_IR
    def    2,i,less_or_equal_RI
    def    2,R,less_or_equal_RR
    def    4,A,less_or_equal_AA

    greater_or_equal: def    1,i,greater_or_equal_II
    def    1,R,greater_or_equal_IR
    def    2,i,greater_or_equal_RI
    def    2,R,greater_or_equal_RR
    def    4,A,greater_or_equal_AA

    equal: def    1,i,equal_II
    def    1,R,equal_IR
    def    2,i,equal_RI
    def    2,R,equal_RR
    def    3,B,equal_BB
    def    4,P,P,equal_PP
Appendix C. Multics Seal Code Generator Table.

```
def S, S, equal_SS

def A, A, equal_AA

not_equal:
def I, I, not_equal_II

def I, R, not_equal_IR

def R, I, not_equal_RI

def R, R, not_equal_RR

def B, B, not_equal_BB

def P, P, not_equal_PP

def S, S, not_equal_SS

def A, A, not_equal_AA

shape:
def L, L, shape_LL

def L, I, shape_LI

def L, R, shape_LR

def I, B, shape_IB

def L, P, shape_LP

def I, S, shape_LS

def I, G, shape_LG

def I, A, shape.LA

exponentiate:
def I, I, exponentiate_II

def I, R, exponentiate_IR

def R, I, exponentiate_RI

def R, R, exponentiate_RR

def A, A, exponentiate_AA

complement:
def B, complement_B

def A, complement_A

def A, complement_A

def Re, deref_Re

negate:
def I, negate_I

def R, negate_R

def A, negate_A

lock:
def G, lock_G

def A, lock_A

unlock:
def G, unlock_G

def A, unlock_A

test_lock:
def G, N, test_lock_GN
```
### Appendix C. Multics Seal Code Generator Table.

<table>
<thead>
<tr>
<th>Definition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>def A,N,test_lock_AN</code></td>
<td>Fill in Nth slot in transfer vector I.</td>
</tr>
<tr>
<td><code>def C,N,case_of_CN</code></td>
<td>This triple generates code to test expression.</td>
</tr>
<tr>
<td><code>def C,I,caselimit_CI</code></td>
<td>This triple actually performs the case transfer.</td>
</tr>
<tr>
<td><code>def I,C,casejump_IC</code></td>
<td>This triple generates a transfer.</td>
</tr>
<tr>
<td><code>def V,branch_N</code></td>
<td>This triple generates a conditional transfer.</td>
</tr>
<tr>
<td><code>def V,B,branch_true_NB</code></td>
<td>Operand 1 is the label.</td>
</tr>
<tr>
<td><code>def V,A,branch_true_NA</code></td>
<td>Operand 2 is the boolean expression.</td>
</tr>
<tr>
<td><code>def V,B,branch_false_NB</code></td>
<td>This triple defines a label's address.</td>
</tr>
<tr>
<td><code>def V,A,branch_false_NA</code></td>
<td>Operand1 is the symbol node of the label.</td>
</tr>
<tr>
<td><code>def V,label_N</code></td>
<td></td>
</tr>
</tbody>
</table>

Page 111.
Appendix G. Multics Seal Code Generator Table.

"This triple generates a SEAL entry sequence for a procedure.  
operand 1 is the symbol node of the procedure.  
output is an entry sequence.

procedure:  
def v,N,procedure_NN

"This triple marks the end of a procedure.  
operand 1 is the block index of this block.

def v,end_N

"This triple generates a Multics link to an external name.  
operand 1 is the symbol node of the name.

def P,link_P

unused_341  
def 0

"The next two triples are used to create lists.  
USAGE: LIST(n), ELEMENT(exp)...
"This triple generates code to fill in an element of the list.  
operand 1 is the expression representing the element.

def v,element_N

"This triple defines the size of the list to be created.  
operand 1 is the number of element triples to follow.

def c,list_C

"form is: ADDR... CALL ARG...
"operand 1 is output of addr triple (value in storage  
output is arg list pointer(i)

def v,arg_N

"This triple generates a call.   
operand 1 is entry to call.  
operand 2 is arg_list triple.

def P,N,call_PN

def v,N,call_AN

" This triple generates a return.
" operand 1 is return from call.  
" operand 2 is return pointer.

def c,ret_I

def 2,ret_R
Appendix C. Multics Seal Code Generator Table.

\[
\begin{align*}
def1 & \quad 3, \text{ret}_B \\
def1 & \quad \?, \text{ret}_P \\
def1 & \quad S, \text{ret}_S \\
def1 & \quad \text{Red}, \text{ret}_\text{Red} \\
def1 & \quad .1, \text{ret}_\text{Li} \\
def1 & \quad A, \text{ret}_A \\
def1 & \quad N, \text{ret}_N \\
\end{align*}
\]

"if \( \text{operand} \ 1 < 0 \) it is the index of a user-supplied operator, otherwise, \( \text{operand} \ 1 \) is the opcode of the language built-in infix operator.

\[
\begin{align*}
\text{reduce}: & \quad \text{def} \quad \?, \text{Li}, \text{reduce}_\text{PLi} \\
& \quad \text{def} \quad C, \text{Li}, \text{reduce}_\text{CLi} \\
\end{align*}
\]

"\( \text{operand} \ 1 \) is the index of the current block.

\[
\begin{align*}
\text{block}: & \quad \text{def} \quad N, \text{block}_N \\
\end{align*}
\]

"This triple performs subscript checking and then subscript evaluation. \( \text{operand} \ 1 \) is the symbol node of the list. \( \text{operand} \ 2 \) is the subscript expression. \( \text{output} \) is the "select"ed element.

\[
\begin{align*}
\text{select}: & \quad \text{def} \quad N, \text{I}, \text{select}_\text{NI} \\
& \quad \text{def} \quad A, \text{I}, \text{select}_\text{AI} \\
\end{align*}
\]

"The output of this triple is just its input.

\[
\begin{align*}
\text{nop}: & \quad \text{def} \quad N, \text{nop}_N \\
\end{align*}
\]

"This triple computes the index of a selector on a mode name. \( \text{operand} \ 1 \) is the symbol node of the mode name. \( \text{operand} \ 2 \) is the selector. \( \text{output} \) is not yet determined.

\[
\begin{align*}
\text{mode_select}: & \quad \text{def} \quad N, \text{N}, \text{mode_select}_\text{NN} \\
\end{align*}
\]

"\( \text{operand} \ 1 \) is line number of source program. \( \text{no output} \).

\[
\begin{align*}
\text{line_number}: & \quad \text{def} \quad C, \text{line_number}_C \\
\end{align*}
\]

"This triple is currently unused.

Page 113.
Appendix C. Multics Seal Code Generator Table.

```
addr: def v,addr_N

" operand 1 is the encode_mode triple.
" operand 2 is the number of dimensions.

encode_dims:
def v,c,encode_dims_NC

" operand 1 is the encode_dims triple.
" operand 2 is the expression to be encoded.

encode_value:
def v,c,encode_value_NC

" This triple allocates space for an argument list.
" operand 1 is the number of arguments.
" output is the address of the arg list.

arg_list:
def c,arg_list_C

" operand 1 is the symbol node which identifies this mode.
" operand 2 is a literal constant describing the
" Reference and List combinations.

encode_mode:
def v,c,encode_mode_NC

split_prept:
def 0

unused_52:
unused_53:
unused_54:
unused_55:
unused_56:
unused_57:
unused_58:
unused_59:
def 0
```

Page 114.
Appendix C. Multics Seal Code Generator Table.

" Zero or one-argument builtin functions.

current:
errortrap:
inchumnt:
infilemark:
initem:
initemmark:
linemark:
inpagemark:
istream:
linesize:
outcolumn:
outfilemark:
outitem:
outitemmark:
outlinemark:
outpagemark:
ostream:
pagesize:
unused_78:
unused_79:
def0
}
Appendix C. Multics Seal Code Generator Tables.

" Single argument builtin functions."

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
<th>Type</th>
<th>Arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>abs</td>
<td>def1</td>
<td>I</td>
<td>abs_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R</td>
<td>abs_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A</td>
<td>abs_A</td>
</tr>
<tr>
<td>atan</td>
<td>def1</td>
<td>I</td>
<td>atan_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R</td>
<td>atan_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A</td>
<td>atan_A</td>
</tr>
<tr>
<td>boolean</td>
<td>def1</td>
<td>I</td>
<td>boolean_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R</td>
<td>boolean_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A</td>
<td>boolean_A</td>
</tr>
<tr>
<td>ceil</td>
<td>def1</td>
<td>R</td>
<td>ceil_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>I</td>
<td>ceil_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A</td>
<td>ceil_A</td>
</tr>
<tr>
<td>cos</td>
<td>def1</td>
<td>I</td>
<td>cos_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R</td>
<td>cos_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A</td>
<td>cos_A</td>
</tr>
<tr>
<td>delete</td>
<td>def1</td>
<td>S</td>
<td>delete_S</td>
</tr>
<tr>
<td>deletedir</td>
<td>def1</td>
<td>S</td>
<td>deletedir_S</td>
</tr>
<tr>
<td>detach</td>
<td>def1</td>
<td>S</td>
<td>detach_S</td>
</tr>
<tr>
<td>exp</td>
<td>def1</td>
<td>I</td>
<td>exp_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R</td>
<td>exp_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A</td>
<td>exp_A</td>
</tr>
<tr>
<td>find</td>
<td>def1</td>
<td>S</td>
<td>find_S</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A</td>
<td>find_A</td>
</tr>
<tr>
<td>floor</td>
<td>def1</td>
<td>R</td>
<td>floor_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>I</td>
<td>floor_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A</td>
<td>floor_A</td>
</tr>
<tr>
<td>integer</td>
<td>def1</td>
<td>R</td>
<td>integer_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>I</td>
<td>integer_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>J</td>
<td>integer_J</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>S</td>
<td>integer_S</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A</td>
<td>integer_A</td>
</tr>
<tr>
<td>isvoid</td>
<td>def0</td>
<td>J</td>
<td></td>
</tr>
</tbody>
</table>

Page 116.
### Appendix C. Multics Seal Code Generator Table

<table>
<thead>
<tr>
<th>Function</th>
<th>Oefi</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>length</td>
<td>def1</td>
<td>_I, length_Li</td>
</tr>
<tr>
<td>log</td>
<td>def1</td>
<td>I, log_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R, log_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, log_A</td>
</tr>
<tr>
<td>log10</td>
<td>def1</td>
<td>I, log10_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R, log10_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, log10_A</td>
</tr>
<tr>
<td>rank</td>
<td>def0</td>
<td>)</td>
</tr>
<tr>
<td>real</td>
<td>def1</td>
<td>I, real_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R, real_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>B, real_B</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>S, real_S</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, real_A</td>
</tr>
<tr>
<td>sign</td>
<td>def1</td>
<td>I, sign_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R, sign_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, sign_A</td>
</tr>
<tr>
<td>sin</td>
<td>def1</td>
<td>I, sin_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R, sin_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, sin_A</td>
</tr>
<tr>
<td>size</td>
<td>def1</td>
<td>S, size_S</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, size_A</td>
</tr>
<tr>
<td>sqrt</td>
<td>def1</td>
<td>I, sqrt_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R, sqrt_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, sqrt_A</td>
</tr>
<tr>
<td>symbol</td>
<td>def1</td>
<td>I, symbol_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R, symbol_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>B, symbol_B</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>S, symbol_S</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>Re, symbol_Re</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>Li, symbol_Li</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, symbol_A</td>
</tr>
<tr>
<td>tan</td>
<td>def1</td>
<td>I, tan_I</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>R, tan_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, tan_A</td>
</tr>
<tr>
<td>trunc</td>
<td>def1</td>
<td>R, trunc_R</td>
</tr>
<tr>
<td></td>
<td>def1</td>
<td>A, trunc_A</td>
</tr>
</tbody>
</table>

Page 117.
Appendix C. Multics Seal Code Generator Table.

unused_105:
unused_106:
unused_107:
unused_108:
unused_109:
def0   }
Appendix C. Multics Seal Code Generator Table.

"Multi-argument argument builtin functions."
"operand 1 is the encode_value triple.
"operand 2 is 0 or is the pathname.
create: def N,0,create_NO
def N,S,create_NS

"operand 1 is the expression to be tested.
"operand 2 is the encode_dims triple.
isset def N,N,is_NN

"operand 1 is the target to assign to.
get: def1 N,get_N

"operand 1 is the expression to print out.
put: def1 N,put_N

"operand 1 is the value.
"operand 2 is ?
void: def1 N,N,void_NN

split: def0 ()

unused_116:
unused_117:
unused_118:
unused_119:
def0 0

Page 119.
Appendix C. Multics Seal Code Generator Table.

"Two argument builtin functions.

attacn:  def  $,S,attachn_SS

createdir: def  $,S,createdir_SS

edit:    def  I,S,edit_IS
         def  $,S,edit_RS
         def  A,A,edit_AA

max:     def  I,I,max_II
         def  I,R,max_IR
         def  $,I,max_RI
         def  $,R,max_RR
         def  A,A,max_AA

min:     def  I,I,min_II
         def  I,R,min_IR
         def  $,I,min_RI
         def  $,R,min_RR
         def  A,A,min_AA

mod:     def  I,I,mod_II
         def  I,R,mod_IR
         def  $,I,mod_RI
         def  $,R,mod_RR
         def  A,A,mod_AA

rename:  def  $,S(rename_SS

rounds:  def  $,C,round_RC
         def  A,C,round_AC
Appendix C. Multics Seal Code Generator Table.

" PATTERNS TO BE INTERPRETED BY THE CODE GENERATOR."

flatten:

" How reference counts are handled:
"
" The parser sets the reference count to the number of
" triple operands which reference the output of a triple.
" The code generator decrements the reference count each
" time it makes the triple output addressable.
" The parser assumes that each triple operand is exactly one
" reference, so if in reality, some pattern makes an
" operand addressable more than once, it must first add one
" to the reference count.
"
" The reference count is also decremented if the
" INREG pattern-op succeeds (or NOT_INREG fails),
" since by doing so it has effectively made its argument
" addressable.
"
" N.B. Patterns consisting of only NOP and RETURN
" have not yet been written.

assign_II:

```
ADD_REFERENCE arg1
LOAD 1,arg2
DEFINE arg1
stq arg1
RESULT 1,I
RETURN
```

assign_IR:

```
ADD_REFERENCE arg1
LOAD EQQ,arg2
t$X0 apiReal_to_Integer
DEFINE arg1
stq arg1
RESULT 1,I
RETURN
```

assign_RII:

```
ERASE A_reg
ADD_REFERENCE arg1
LOAD 1,arg2
t$X0 apiInteger_to_Real
UEDEFINE arg1
dfst arg1
```

Page 121.
Appendix C. Multics Seal Code Generator Table.

RESULT EAQ,R
RETURN

assign_RR:
ADD_REFERENCE arg1
LOAD EAQ, arg2
DEFINE arg1
dist arg1
RESULT EAQ,R
RETURN

assign_RB:
LOAD A_reg, arg2
DEFINE arg1
sta arg1
RESULT A_reg, B
RETURN

assign_PP
assign_SS
assign_ReRe
assign_LiLi
assign_AA:
NOP
RETURN

add_IL:
IF_INREG 1, arg1
then { adq arg2 }
else { IF_INREG Q, arg2
then { adq arg1 }
else { LOAD Q, arg1
adq arg2 }}
RESULT 1,1
RETURN

add_IR:
IF_INREG 1, arg1
then { ERASE A_reg
fsxO aplInteger_to_Real
ifad arg2 }
else { LOAD Q, arg1
ERASE A_reg
fsxO aplInteger_to_R
ifad arg2 }
RESULT EAQ,R
RETURN
Appendix C. Multics Seal Code Generator Table.

add_RI:
\[ \text{IF} _{\text{INREG}} \quad l, \text{arg2} \]
\[ \text{then} \{ \]
\[ \text{ERASE} \quad A_\text{reg} \]
\[ \text{tsx0} \quad \text{aplInteger_to_Real} \]
\[ \text{dfad} \quad \text{arg1} \} \]
\[ \text{else} \{ \]
\[ \text{LOAD} \quad Q, \text{arg2} \]
\[ \text{ERASE} \quad A_\text{reg} \]
\[ \text{tsx0} \quad \text{aplInteger_to_Real} \]
\[ \text{dfad} \quad \text{arg1} \} \]
\[ \text{RESULT} \quad \text{EAQ}, R \]
\[ \text{RETURN} \]

add_RR:
\[ \text{IF} _{\text{INREG}} \quad \text{EAQ}, \text{arg1} \]
\[ \text{then} \{ \]
\[ \text{dfad} \quad \text{arg2} \} \]
\[ \text{else} \{ \]
\[ \text{IF} _{\text{INREG}} \quad \text{EAQ}, \text{arg2} \]
\[ \text{then} \{ \]
\[ \text{dfad} \quad \text{arg1} \} \]
\[ \text{else} \{ \]
\[ \text{LOAD} \quad \text{EAQ}, \text{arg1} \]
\[ \text{dfad} \quad \text{arg2} \} \}
\[ \text{RESULT} \quad \text{EAQ}, R \]
\[ \text{RETURN} \]

add_AA:
\[ \text{IF} _{\text{INREG}} \quad \text{Any}, \text{arg1} \]
\[ \text{then} \{ \]
\[ \text{SETUP} \quad \text{Any}, \text{arg2} \]
\[ \text{tsx0} \quad \text{apladd_any_operator} \}
\[ \text{else} \{ \]
\[ \text{IF} _{\text{INREG}} \quad \text{Any}, \text{arg2} \]
\[ \text{then} \{ \]
\[ \text{SETUP} \quad \text{Any}, \text{arg1} \]
\[ \text{tsx0} \quad \text{apladd_any_operator} \}
\[ \text{else} \{ \]
\[ \text{LOAD} \quad \text{Any}, \text{arg1} \]
\[ \text{SERIP} \quad \text{Any}, \text{arg2} \]
\[ \text{tsx0} \quad \text{apladd_any_operator} \} \}
\[ \text{RESULT} \quad \text{Any}, \text{Any} \]
\[ \text{RETURN} \]

sub_II:
\[ \text{LOAD} \quad l, \text{arg1} \]
\[ \text{sbq} \quad \text{arg2} \]
\[ \text{RESULT} \quad l, 1 \]
\[ \text{RETURN} \]

sub_IR:
\[ \text{ERASE} \quad A_\text{reg} \]
\[ \text{LOAD} \quad l, \text{arg1} \]
\[ \text{tsx0} \quad \text{aplInteger_to_Real} \]
\[ \text{dfsd} \quad \text{arg2} \]
\[ \text{RESULT} \quad \text{EAQ}, R \]
Appendix C. Multics Seal Code Generator Table.

RETURN

sub_RI:
IF_INREG 1, arg2
then {
  ERASE A_reg
  fsx0 ap!Integer_to_Real
  dfsb arg1
  fneg 0
}
else {
  LOADQ Q, arg2
  ERASE A_reg
  fsx0 ap!Integer_to_Real
  dfad arg1
}
RESULT EAQ,R
RETURN

sub_RR:
IF_INREG EAQ, arg1
then {
  dfsb arg2
}
else {
  IF_INREG EAQ, arg2
  then {
    dfsb arg1
    fneg 0
  }
  else {
    LOAD EAQ, arg1
    dfsb arg2
  }
}
RESULT EAQ,R
RETURN

sub_AA:
LOAD Any, arg2
SETUP Any, arg1
fsx0 ap!Subtract_any_operator
RESULT Any, Any
RETURN

divide_HI:
ERASE A_reg
LOAD 1, arg1
div arg2
RESULT 2, I
RETURN

divide_IR:
ERASE A_reg
LOAD 2, arg1
tsx0 ap!Integer_to_Real
dfdv arg2
RESULT EAQ,R
Appendix C. Multics Seal Code Generator Table.

RETURN

divide_RI:
ERASE A_reg
LOAD l, arg2
tsx0 apiInteger_to_Real
dfdi arg1
RESULT EAQ,R
RETURN

divide_R2:
IF_INREG EAQ, arg1
then ( dfdv arg2 )
else ( IF_INREG EAQ, arg2
then ( dfdi arg1 )
else ( LOAD EAQ, arg1
dfdv arg2 ) )
RESULT EAQ,R
RETURN

divide_AA:
LOAD Any, arg2
SETUP Any, arg1
tsx0 apidivide_any_operator
RESULT Any, Any
RETURN

multiply_IL:
ERASE A_reg
IF_INREG 1, arg1
then ( mpy arg2 )
else ( IF_INREG Q, arg2
then ( mpy arg1 )
else ( LOAD Q, arg1
mpy arg2 ) )
" next instruction sets the carry bit = 1 if bit 0 ever
" changes during shift, else sets it = 0.
11s 36
" and the next instruction transfers if carry = 0,
" meaning that all of the bits in the A register,
" and the sign bit of the Q were equal, meaning t it
" the number in the AQ was single precision.
1nc 2, ic
tsx0 apifixedoverflow_operator
lrs 36
RESULT l, I
RETURN
Appendix C. Multics Seal Code Generator Table.

\[\text{multiply}_\text{IR}\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERASE</td>
<td>A_reg</td>
</tr>
<tr>
<td>LOAD</td>
<td>arg1</td>
</tr>
<tr>
<td>\text{tsx0}</td>
<td>\text{apInteger}_to_\text{Real}</td>
</tr>
<tr>
<td>\text{dfmp}</td>
<td>arg2</td>
</tr>
<tr>
<td>\text{RESULT}</td>
<td>EAQ,R</td>
</tr>
<tr>
<td>\text{RETURN}</td>
<td></td>
</tr>
</tbody>
</table>

\[\text{multiply}_\text{RR}\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{IF}_\text{INREG}</td>
<td>\text{EAQ},arg1</td>
</tr>
<tr>
<td></td>
<td>\text{dfmp} arg2</td>
</tr>
<tr>
<td></td>
<td>\text{ELSE}</td>
</tr>
<tr>
<td></td>
<td>\text{IF}_\text{INREG} EAQ,arg2</td>
</tr>
<tr>
<td></td>
<td>\text{dfmp} arg1</td>
</tr>
<tr>
<td></td>
<td>\text{ELSE}</td>
</tr>
<tr>
<td></td>
<td>\text{LOAD} EAQ,arg1</td>
</tr>
<tr>
<td></td>
<td>\text{dfmp} arg2</td>
</tr>
<tr>
<td>\text{RESULT}</td>
<td>EAQ,R</td>
</tr>
<tr>
<td>\text{RETURN}</td>
<td></td>
</tr>
</tbody>
</table>

\[\text{multiply}_\text{AA}\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{IF}_\text{INREG}</td>
<td>Any,arg1</td>
</tr>
<tr>
<td></td>
<td>\text{SETUP} Any,arg2</td>
</tr>
<tr>
<td></td>
<td>\text{tsx0} \text{apImultiply}_any_operator</td>
</tr>
<tr>
<td></td>
<td>\text{ELSE}</td>
</tr>
<tr>
<td></td>
<td>\text{IF}_\text{INREG} Any,arg2</td>
</tr>
<tr>
<td></td>
<td>\text{SETUP} Any,arg1</td>
</tr>
<tr>
<td></td>
<td>\text{tsx0} \text{apImultiply}_any_operator</td>
</tr>
<tr>
<td></td>
<td>\text{ELSE}</td>
</tr>
<tr>
<td></td>
<td>\text{LOAD} Any,arg1</td>
</tr>
<tr>
<td></td>
<td>\text{SETUP} Any,arg2</td>
</tr>
<tr>
<td></td>
<td>\text{tsx0} \text{apImultiply}_any_operator</td>
</tr>
<tr>
<td>\text{RESULT}</td>
<td>Any,Any</td>
</tr>
<tr>
<td>\text{RETURN}</td>
<td></td>
</tr>
</tbody>
</table>

\[\text{and}_\text{BB}\]

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{IF}_\text{INREG}</td>
<td>A_reg,arg1</td>
</tr>
<tr>
<td></td>
<td>ana arg2</td>
</tr>
<tr>
<td></td>
<td>\text{ELSE}</td>
</tr>
<tr>
<td></td>
<td>\text{IF}_\text{INREG} A_reg,arg2</td>
</tr>
<tr>
<td></td>
<td>ana arg1</td>
</tr>
<tr>
<td></td>
<td>\text{ELSE}</td>
</tr>
<tr>
<td></td>
<td>\text{LOAD} A_reg,\text{arg}</td>
</tr>
<tr>
<td></td>
<td>ana \text{arg}</td>
</tr>
<tr>
<td>\text{RESULT}</td>
<td>A_reg,B</td>
</tr>
</tbody>
</table>
and_AA:

```
IF_INREG Any, arg1
then {
  SETUP Any, arg2
  
  tsx0 aplaind_any_operator }
else {
  IF_INREG Any, arg2
  then {
    SETUP Any, arg1
    tsx0 aplaind_any_operator }
  else {
    LOAD Any, arg1
    SETUP Any, arg2
    tsx0 aplaind_any_operator }}
RESULT Any, arg1
RETURN
```

or_BB:

```
IF_INREG A_reg, arg1
then {
  ora arg2 }
else {
  IF_INREG A_reg, arg2
  then {
    ora arg1 }
  else {
    LOAD A, arg1
    ora arg2 }
RESULT A_reg, B
RETURN
```

or_AA:

```
IF_INREG Any, arg1
then {
  SETUP Any, arg2
  tsx0 aplor_any_operator }
else {
  IF_INREG Any, arg2
  then {
    SETUP Any, arg1
    tsx0 aplor_any_operator }
  else {
    LOAD Any, arg1
    SETUP Any, arg2
    tsx0 aplor_any_operator }}
RESULT Any, B
RETURN
```

catenate_Lili:
catenate_symbol_SS:
catenate_symbol_AA:
NOP
RETURN
less_than_II:
Appendix C. Multics Seal Code Generator Table.

ERASE A_reg
LOAD q, arg1
cmpq arg2
tsx0 appllt_to_bool
RESULT A_reg,B
RETURN

less_than_IRI:
LOAD 1, arg1
ERASE A_reg
tsx0 applnteger_to_Real
dfcmp arg2
tsx0 appllt_to_bool
RESULT A_reg,B
RETURN

less_than_RRI:
LOAD 1, arg2
ERASE A_reg
tsx0 applnteger_to_Real
dfcmp arg1
tsx0 appllt_to_bool
RESULT A_reg,B
RETURN

less_than_RRI:
IF_INREG EAQ, arg1
then {  dfcmp arg2
        tsx0 appllt_to_bool
    }
else { IF_INREG EAQ, arg2
        then {  dfcmp arg1
tsx0 applgt_to_bool
        }
        else { LOAD EAQ, arg1
dfcmp arg2
tsx0 appllt_to_bool
        }
    }
RESULT A_reg,B
RETURN

less_than_AAI:
ERASE A_reg
LOAD Any, arg1
SETUP Any, arg2
tsx0 applless_than_any_operator
RESULT A_reg,B
RETURN

greater_than_IRI:

Page 128.
Appendix C: Multics Seal Code Generator Table.

ERASE  A_reg
LOAD  arg1
cmpq  arg2
tso  aplgt_to_bool
RESULT  A_reg,B
RETURN

greater_than_RI:
LOAD  arg1
ERASE  A_reg
tso  aplgt_to_Real
dfcmp  arg2
tso  aplgt_to_bool
RESULT  A_reg,B
RETURN

greater_than_RII:
LOAD  arg2
ERASE  A_reg
tso  aplgt_to_Real
dfcmp  arg1
tso  apllt_to_bool
RESULT  A_reg,B
RETURN

greater_than_RIII:
IF_INREG  EQA, arg1
ten {  dfcmp  arg2
tso  aplgt_to_bool  }
else {  IF_INREG  EQA, arg2
ten {  dfcmp  arg1
tso  apllt_to_bool  }
else {  LOAD  EQA, arg1
dfcmp  arg2
tso  aplgt_to_bool  }
RESULT  A_reg,B
RETURN

less_or_equal_RII:
ERASE  A_reg
NOP
RETURN

Page 129.
Appendix G. Multics Seal Code Generator Table.

```
IF_INREG \arg1 \arg2
then {
  \texttt{cmpq arg2}
  \texttt{tsx0 \texttt{apilt_or_eq_to_bool}}
}
else {  \texttt{IF_INREG Q, arg2}
  \texttt{then { \texttt{cmpq arg1}}
  \texttt{tsx0 \texttt{apilt_or_eq_to_bool}}
  \texttt{else { \texttt{LOAD Q, arg1}}
  \texttt{cmpq arg2}
  \texttt{tsx0 \texttt{apilt_or_eq_to_bool}}
  \texttt{RESULT A_reg, B}
  \texttt{RETURN}
}
}

less_or_equal_IR:
\texttt{LOAD \arg1}
\texttt{ERASE A_reg}
\texttt{tsx0 \texttt{aplinteger_to_Real}}
\texttt{dfcmp \arg2}
\texttt{tsx0 \texttt{apilt_or_eq_to_bool}}
\texttt{RESULT A_reg, B}
\texttt{RETURN}

less_or_equal_RIR:
\texttt{LOAD \arg2}
\texttt{ERASE A_reg}
\texttt{tsx0 \texttt{aplinteger_to_Real}}
\texttt{dfcmp \arg1}
\texttt{tsx0 \texttt{apilt_or_eq_to_bool}}
\texttt{RESULT A_reg, B}
\texttt{RETURN}

less_or_equal_RIR:
\texttt{IF_INREG EAQ, arg1}
\texttt{then { \texttt{dfcmp \arg2}}
\texttt{tsx0 \texttt{apilt_to_bool}}
}
else { \texttt{IF_INREG EAQ, arg2}
\texttt{then { \texttt{dfcmp arg1}}
\texttt{tsx0 \texttt{apilt_to_bool}}
}
else { \texttt{LOAD EAQ, arg1}}
\texttt{dfcmp \arg2}
\texttt{tsx0 \texttt{apilt_to_bool}}
\texttt{RESULT A_reg, B}
\texttt{RETURN}
}

less_or_equal_AAA:
\texttt{ERASE A_reg}
\texttt{NOP}
\texttt{RETURN}
```

Page 130.
Appendix C. Multics Seal Code Generator Table.

greater_or_equal_IIt:
ERASE A_reg
IF_INREG Q, arg1
then { cmpq arg2
tsx0 ap1gt_or_eq_to_bool } 
else { IF_INREG Q, arg2
then { cmpq arg1
tsx0 ap1gt_or_eq_to_bool }
else { LOAD Q, arg1
cmpq arg2
tsx0 ap1gt_or_eq_to_bool } }
RESULT A_reg, B 
RETURN

greater_or_equal_IRI:
LOAD Q, arg1
ERASE A_reg
tsx0 ap1Integer_to_Real
dfcmp arg2
tsx0 ap1gt_or_eq_to_bool
RESULT A_reg, B 
RETURN

greater_or_equal_RRI:
LOAD Q, arg2
ERASE A_reg
tsx0 ap1Integer_to_Real!
dfcmp arg1
tsx0 ap1gt_or_eq_to_bool
RESULT A_reg, B 
RETURN

greater_or_equal_RRI:
IF_INREG EAQ, arg1
then { dfcmp arg2
tsx0 ap1gt_or_eq_to_bool }
else { IF_INREG EAQ, arg2
then { dfcmp arg1
tsx0 ap1gt_or_eq_to_bool }
else { LOAD EAQ, arg1
dfcmp arg2
tsx0 ap1gt_or_eq_to_bool } }
RESULT A_reg, B 
RETURN

greater_or_equal_AAI:
ERASE A_reg
Appendix C: Multics Seal Code Generator Table.

NOP
RETURN

**equal_II:**

ERASE A_reg
IF_INREG Q, arg1
then { cmpq arg2 }
else { IF_INREG Q, arg2
then { cmpq arg1 }
else { LOAD Q, arg1
cmpq arg2 }
}
tsx0 apieq_to_bool
RESULT A_reg, B
RETURN

**equal_IR:**

ERASE A_reg
LOAD Q, arg1
tsx0 apilnteger_to_Real
dfcmp arg2
tsx0 apieq_to_bool
RESULT A_reg, B
RETURN

**equal_RI:**

ERASE A_reg
LOAD Q, arg2
tsx0 apilnteger_to_Real
dfcmp arg1
tsx0 apieq_to_bool
RESULT A_reg, B
RETURN

**equal_RR:**

IF_INREG EAQ, arg1
then { dfcmp arg2 }
else { IF_INREG EAQ, arg2
then { dfcmp arg1 }
else { LOAD EAQ, arg1
dfcmp arg2 }
}
tsx0 apieq_to_bool
RESULT A_reg, B
RETURN

**equal_3d:**

IF_INREG A_reg, arg1
then { cmpq arg2 }

Page 132
Appendix C. Multics Seal Code Generator Table.

else { IF_INREG A_reg, arg2
then { cmpa arg1 }
else { LOAD A_reg, arg1
cmpa arg2 }
}

" Tight fit here: eq_to_bool will destroy A_reg,
" but ERASE operation won't destroy machine indicators.
" Might be safer to always ERASE then LOAD arg1 into A_reg.

ERASE A_reg
tsx0 apIeq_to_bool
RESULT A_reg, B
RETURN

equal_PP:
equal_SS:
equal_AA:

NOP
RETURN

not_equal_RI1:
LOAD l, arg1
ERASE A_reg
tsx0 apIInteger_to_Real
dfcmp arg2
tsx0 apine_to_bool
RESULT A_reg, B
RETURN

not_equal_RI2:
LOAD l, arg2
ERASE A_reg
tsx0 apIInteger_to_Real
dfcmp arg1
tsx0 apine_to_bool
RESULT A_reg, B
RETURN
Appendix C. Multics Seal Code Generator Table.

not_equal_RR:
  IF_INREG EAQ, arg1
  then { ifcmp arg2 }
  else { IF_INREG EAQ, arg2
          then { dfcmp arg1 }
          else { LOAD EAQ, arg1
                   dfcmp arg2 }
          } }
  
  tsx0 apline_to_bool
  RESULT A_reg, B
  RETURN

not_equal_BB:
  IF_INREG A_reg, arg1
  then { cmpa arg2 }
  else { IF_INREG A_reg, arg2
          then { cmpa arg1 }
          else { LOAD A_reg, arg1
                   cmpa arg2 }
          } }
  ERASE A_reg
  tsx0 apline_to_bool
  RESULT A_reg, B
  RETURN

not_equal_PP:
not_equal_SS:
not_equal_AA:
  NOP
  RETURN

shape_LiLi:
shape_LiI:
shape_LiR:
shape_LiB:
shape_LiP:
shape_LiS:
shape_LiG:
shape_LiA:
  NOP
  RETURN

exponentiate_Il:
exponentiate_Ir:
exponentiate_II:
exponentiate_RR:
exponentiate_AA:
  NOP
  RETURN
Appendix C. Multics Seal Code Generator Table.

complement_B:
LOAD 1_reg, arg1
ERA =00000000, du
RESULT A_reg, B
RETURN

complement_A:
LOAD 1_reg, arg1
tsx0 apicomplement_any_operator
RESULT Any, B
RETURN

deref_Ra:
NOP
RETURN

negate_I:
LOADC 1, arg1
RESULT 1, I
RETURN

negate_R:
IF_INREG EAQ, arg1
then { fneg 0 }
else { LOADC EAQ, arg1 }
RESULT EAQ, R
RETURN

negate_A:
LOAD Any, arg1
tsXO apinegatc_any_operator
RESULT Any, Any
RETURN

lock_G:
lock_A:
unlock_G:
unlock_A:
test_lock_G:
test_lock_A:
NOP
RETURN

case_of_G:
CASE_USAGE arg1, arg2
RETURN
Appendix C. Multics Seal Code Generator Table.

casematch_CI:
   ERASE   Q
   lda     1,d1
   LOAD    F_arg1
   cwi     arg2
   tsx0    aplec_to_poo1
   RESULT  A_reg,B
   RETURN

casematch_IC:
   LOAD     Q, arg1
   ALLOCATE_CASE arg2
   " no RESULT pattern: output will be set directly
   " by the ALLOCATE_CASE pattern operator.
   RETURN

branch_N:
   NOTE_FLOWCHANGE
   tpt arg1
   RETURN

branch_true_NB:
   NOTE_FLOWCHANGE
   LOAD    A_reg, arg2
   tpt arg1
   RETURN

branch_true_NA:
   NOTE_FLOWCHANGE
   ADD_REFERENCE arg2
   TYPE_CHECK   B, arg2
   LOAD    A_reg, arg2
   tpt arg1
   RETURN

branch_false_NB:
   NOTE_FLOWCHANGE
   LOAD    A_reg, arg2
   tpt arg1
   RETURN

branch_false_NA:
   NOTE_FLOWCHANGE
   ADD_REFERENCE arg2
   TYPE_CHECK   B, arg2
   LOAD    A_reg, arg2

Page 130.
Appendix C. Multics Seal Code Generator Table.

```
tpl arg1
RETURN

label_N:
NOTE_FLOWCHANGE
FILL_USAGE arg1
RETURN

procedure_NN:
PROCEDURE DEF arg1
ENTRY_LOCATION
eax7
epbb
sp140,*
"seal_operators&operator_table
tspab Opentry_operator
ENTRY_CONTROL_WORD1
oct
"
"will be stack size,
"stack template offset & size,
RETURN

segdef ENTRY_CONTROL_WORD1_OFFSET
ENTRY_CONTROL_WORD1_OFFSET
zero 0,ENTRY_CONTROL_WORD1-ENTRY_LOCATION

end_N:
SET_STACK_SIZE arg1
RETURN

link_P:
PROCEDURE LINK arg1
RETURN

element_N:
FILL_LIST arg1
RETURN

list_C:
ALLOCATE_LIST arg1
RETURN

arg_list_C:
ALLOCATE_ARG_LIST arg1
RETURN

arg_N:
LOAD 3P, arg1
ARG_PTR 3P, arg2
RETURN
```

Page 137.
Appendix C. Multics Seal Code Generator Table.

call_ANI:

ADD_REFERENCE arg1
TYPE_CHECK P, arg1
" fall through into CALL_PN code.

call_PN:

ERASE 38
ERASE 53
ERASE _P
IF_OPERAND arg2, is_zero
then { appab apnnull_arg_list }
else { LOAD AB, arg2 }
epp1p sp!seal_frame.linkage_ptr,*
LOAD 3P, arg1
tsx0 ap!call_operator

" This pattern has no RESULT pattern-op because there is
" currently no way to talk about a return value.
" Perhaps CALL can do it in the future.
" CALL arg1, arg2 unused.
tsx0 ap!reload_registers_operator
RETURN

ret_I:
ret_R:
ret_B:
ret_P:
ret_S:
ret_Re:
ret_Li:
ret_A:

NOP
RETURN

ret_N:
tra ap!return_operator
RETURN

reduce_PLi:
reduce_CLi:
NOP
RETURN

block_N:
SET_BLOCK arg1
RETURN

select_A:

Page 138.
Appendix C. Multics Seal Code Generator Table.

TYPE_CHECK Li, arg1

"fall into select NI code.

select NI:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ERASE</td>
<td>4_reg</td>
</tr>
<tr>
<td>lda</td>
<td>1, dl</td>
</tr>
<tr>
<td>load</td>
<td>2, arg1</td>
</tr>
<tr>
<td>qr1</td>
<td>16</td>
</tr>
<tr>
<td>cwi</td>
<td>arg2</td>
</tr>
<tr>
<td>tze</td>
<td>2, ic</td>
</tr>
<tr>
<td>tso</td>
<td>api:: subscript_error_operator</td>
</tr>
<tr>
<td>load</td>
<td>2, arg2</td>
</tr>
<tr>
<td>epnp</td>
<td>arg1, *qi</td>
</tr>
<tr>
<td>result</td>
<td>3P, Re</td>
</tr>
<tr>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

nop NI:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nop</td>
<td>) , dl</td>
</tr>
</tbody>
</table>

return

mode select NN:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode select</td>
<td>arg1, arg2</td>
</tr>
</tbody>
</table>

return

line number C:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

addr N:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td></td>
</tr>
</tbody>
</table>

encode dims NC:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>encode value NC</td>
<td></td>
</tr>
<tr>
<td>encode mode NC</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td></td>
</tr>
</tbody>
</table>

return

aos I:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>erase</td>
<td>2</td>
</tr>
<tr>
<td>load</td>
<td>4_reg, arg1</td>
</tr>
<tr>
<td>cmpa</td>
<td>) , dl</td>
</tr>
<tr>
<td>tpi</td>
<td>2, ic</td>
</tr>
<tr>
<td>neg</td>
<td>3</td>
</tr>
<tr>
<td>lrs</td>
<td>36</td>
</tr>
</tbody>
</table>
Appendix C. Multics Seal Code Generator Table.

RESULT r,i
RETURN

abs_R1
LOAD EAQ, arg1
cmpa 1,dl
tcp 2,ic
fneg 0
RESULT EAQ,R
RETURN

abs_A1
NOP
RETURN

atan_I1
atan_R1
atan_A1
NOP
RETURN

boolean_I1
ERASE A_reg
LOAD a,arg1
cmpq 0,dl
tze 2,ic
ldq =0400000,du
ldr 3b
RESULT A_reg,B
RETURN

boolean_R1
LOAD EAQ, arg1
dcmp =0.0e0,du
" Next instruction assumes that a floating point zero
" leaves the A_reg = 0.
tze 2,ic
lda =0400000,du
RESULT A_reg,B
RETURN

boolean_A1

ceil_R1
ceil_I1
ceil_A1

Page 140.
Appendix C. Multics Seal Code Generator Table.

```plaintext
cos_It
cos_Rt
cos_At
delete_S:
deletedir_S:
detach_S:
exp_I:
exp_R:
exp_A:
find_SI:
find_A:
floor_R:
floor_I:
floor_A:

integer_I:
  LOAD       1, arg1
  RESULT     1, I
  RETURN

integer_R:
  LOAD       EQ, arg1
  tso0 apiReal_to_Integer
  RESULT     1, I
  RETURN

integer_B:
  LOAD       A_reg, arg1
  IRAI
  RESULT     1, I
  RETURN

integer_S:
integer_A:

length_Li:
  NOO
  RETURN

log_I:
log_R:
log_A:
```

Page 141.
Appendix C. Multics Seal Code Generator Table.

log10_I:
log10_R:
log10_A:

NOP
RETURN

real_I:

ERASE a_reg
LOAD q, arg1
tsx0 ap!Integer_to_Real
RESULT EQ, R
RETURN

real_R:

LOAD EQ, arg1
RESULT EQ, R
RETURN

real_B:

ERASE q
LOAD a_reg, arg1
lrl 71
tsx0 ap!Integer_to_Real
RESULT EQ, R
RETURN

real_S:

real_A:

NOP
RETURN

sign_I:

ERASE 1
ldq 1, dl
szn arg1
tze 3, ic " =0 0"
tmi 3, ic " >0 1"
ldq 1, dl
tra 2, ic
lcq 1, dl
RESULT 2, I
RETURN

sign_R:

ERASE EQ
ldq 0, dl
" The next instruction will correctly test a double precision
Appendix C. Multics Seal Code Generator Table.

" number for being <0, =0, or >0, which is all we need.
" (there is no ofszn instruction).
  fszn      arg1
  tze       3,ic
  tmi       3,ic
  ldq       1,dl
  tra       2,ic
  lcaq      1,dl
  RESULT    2,I
  RETURN

sign_A:

sin_I:
sin_R:
sin_A:

size_S:
size_A:

sqrt_I:
sqrt_R:
sqrt_A:

symbol_I:
symbol_Re:
symbol_Li:
symbol_R:
symbol_B:
symbol_S:
symbol_A:

tan_I:
tan_R:
tan_A:

trunc_R:
trunc_A:

create_NO:
create_NS:

is_NN:
  NOP
  RETURN

Page 143.
Appendix C. Multics Seal Code Generator Table.

get_N:

ERASE _p
ERASE _sb
IF_OPERAND arg1, is_zero
then { appbp apinull_pointer, * }
else { load BP, arg1 }
tsx0 apiget_operator
return

put_N:

ERASE _p
ERASE _sb
IF_OPERAND arg1, is_zero
then { appbp apinull_pointer, * }
else { load BP, arg1 }
tsx0 aplitput_operator
return

void_N:
attach_SS:
createdir_SS:
edit_IS:
edit_RS:
edit_AA:
nop
return

max II:

ADD_REFERENCE arg2
load l, arg1
cmpeq arg2
tpl 2, l
load l, arg2
result l, i
return

max IR:

ADD_REFERENCE arg2
load l, arg1
erase A_reg
tsx0 apltInteger_to_Real
dfcmp arg2
tpl 2, l
load l, arg2
return

Page 144.
Appendix C. Multics Seal Code Generator Table.

RESULT EAQ,R
RETURN

max_RI:
ADD_REFERENCE arg1
ERASE A_reg
LOAD 2,arg2
tsx0 aplInteger_to_Real
cfcmp arg1
fmi 2,ic
LOAD EAQ,arg1
RESULT EAQ,R
RETURN

max_RR:
ADD_REFERENCE arg2
LOAD EAQ,arg1
cfcmp arg2
fpl 2,ic
LOAD EAQ,arg2
RESULT EAQ,R
RETURN

max_AA:
NOP
RETURN

min_RI:
ADD_REFERENCE arg2
LOAD 2,arg1
cmpq arg2
fmi 2,ic
LOAD 2,arg2
RESULT 2,I
RETURN

min_RR:
ADD_REFERENCE arg2
ERASE A_reg
LOAD 2,arg1
tsx0 aplInteger_to_Real
cfcmp arg2
fmi 2,ic
LOAD EAQ,arg2
RESULT EAQ,R
RETURN
Appendix C. Multics Seal Code Generator Table.

<table>
<thead>
<tr>
<th>Label</th>
<th>Instruction</th>
<th>Args</th>
</tr>
</thead>
<tbody>
<tr>
<td>min_RI</td>
<td>ADD_REFERENCE</td>
<td>arg1</td>
</tr>
<tr>
<td></td>
<td>ERASE</td>
<td>A_reg</td>
</tr>
<tr>
<td></td>
<td>LOAD</td>
<td>l,arg2</td>
</tr>
<tr>
<td></td>
<td>Tsto</td>
<td>apiInteger_to_Real</td>
</tr>
<tr>
<td></td>
<td>afcmp</td>
<td>arg1</td>
</tr>
<tr>
<td></td>
<td>tpi</td>
<td>2,ic</td>
</tr>
<tr>
<td></td>
<td>LOAD</td>
<td>EAQ,arg1</td>
</tr>
<tr>
<td></td>
<td>RESULT</td>
<td>EAQ,R</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>min_RR</td>
<td>ADD_REFERENCE</td>
<td>arg2</td>
</tr>
<tr>
<td></td>
<td>LOAD</td>
<td>EAQ,arg1</td>
</tr>
<tr>
<td></td>
<td>afcmp</td>
<td>arg2</td>
</tr>
<tr>
<td></td>
<td>tpi</td>
<td>2,ic</td>
</tr>
<tr>
<td></td>
<td>LOAD</td>
<td>EAQ,arg2</td>
</tr>
<tr>
<td></td>
<td>RESULT</td>
<td>EAQ,R</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>min_AA</td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>mod_II</td>
<td>ERASE</td>
<td>A_reg</td>
</tr>
<tr>
<td></td>
<td>LOAD</td>
<td>l,arg1</td>
</tr>
<tr>
<td></td>
<td>div</td>
<td>arg2</td>
</tr>
<tr>
<td></td>
<td>lrl</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>RESULT</td>
<td>l,l</td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td>mod_IR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_RI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_RR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod_AA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rename_SS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>round_RG</td>
<td></td>
<td></td>
</tr>
<tr>
<td>round_LA</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NOP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RETURN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
An Implementation of Seal on Multics.

BIBLIOGRAPHY


McCracken, D. D., and Weinberg, G. M., "How to Write a Readable FORTRAN Program," DATAMATION 18, 10 (October 1972), 73-77.


Page 147.