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 D. 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 sponsered by Proje[illegible] AC and Honeywell Information Systems, Inc., and by the Adva[illegible]d Research Projects Agency of the Department of Defense.

An Implementation of Seal on Multics.

Table of Contents

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 typeball. The appendicies were printed on a Honeywell PRT-300 line printer.

An Implementation of Seal on Multics.

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 Sys ," by R. A. Freiburghouse, Honeywell, Inc., to be publis

(2) "Multics Programmers' Manual," MIT Proj MAC and Honeywell Information Systems, Inc., 1973.

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 require[illegible] hat every reference be validated, and that assumptions o[illegible] iorance or secrecy are inadequate to solve the problem. Rece[illegible] more emphasis has 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, e code genera-

---

(1) "A Hardware Architecture for Implementin otection Rings," by Michael D. Schroeder and Jerome H. Sa er, Comm ACM 15, 3 (March 1972), p157-170.

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 separately compiled procedures, each of which may contain nested procedures. 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. Complete 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 extensibility without loss of program readability.

The design objectives of Seal which influenced this thesis are:

1. It must be compilable into non-interpretive code, but it must allow some variables to possess values of unrestricted 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 secur [illegible] at no program, legal or illegal, can destroy its [illegible] or its supporting mechanisms.

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 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.

---

(1) "Software and Its Impact: A Quantitative Assessment," Barry W. Boehm, DATAMATION 19, 5 (May 1973), page .

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 capabilitites 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 followin[illegible] lisclaimer: "A program that violates the constraint[s listed [illegible] this manual] may or may not be compiled by the Multics PL/[illegible] mpiler. If compiled, it may or may not execute. If execut[illegible] t may or may not produce consistent results in the current or [illegible] ure versions of the implementation." Multics PL/I Langu[illegible], Document AG94, Honeywell Information Systems, Inc., 1972, [illegible] je 1-5.

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 data pe. Stated in PL/I terms, it is possible to have a pointe an integer, or a pointer to a character string, but they are interchangeable;

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 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 H6180 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 referring to the table entry for the language operator and reading a few lines of information. All code generating decisions are made by the table.

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" claues, symbolic variables, and a limited form of subroutine call. The compiler writer must be careful that his internal language does not become too unwieldly itself. Otherwise, too much effort will be spent debugging it, when the real task is to correctly compile the user's program.

Section II.4: Code Generator Implementation.

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 process 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 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 [illegible]
'ldq constant_address' is changed to 'ldq c[illegible]ant_value,dl'
'ldq constant_address' is changed to 'ldq c[illegible]tant_address-*,ic'
In the first example, the constant must be [illegible]ess than 2**18, in the second example it may be any value.

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

---

(1) Compiler Construction for Digital Computers, David Gries, p379.

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
```

is optimized into:

```
load    y
add     1
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 mechanism, 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.

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 existance of saved values at predetermined points, in coordination with the optimizer, which never uses a saved result across a transfer-of-control point.

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, "How to Write a Readable FORTRAN Program," by Daniel D. Mc[illegible]cken and Gerald M. Weinberg, DATAMATION 18, 10 (October 1972), [illegible]3-77.

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-m[illegible]down" programming management philosophy create a situation in [illegible]ch no subsystem is ever fully debugged. Each person who modi[illegible] the subsystem runs the risk of adding a bug along with his "f[illegible]." Every programmer 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 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 builtin 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 cations and con-

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," b without hardware assistance of some sort it is usually prohi vely expensive in 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 WATFOR 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.

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

---

(1) "Design and Implementation of a Diagnos c Compiler for PL/I," by Richard W. Conway and Thomas R. Wil , Comm ACM 16, 3 (March 1973), p176.

(2) "A Conversational Compiler for Full PL/ by R. N. Cuff, British Computer Society Journal 15, 2 (May /2), p99-104.

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.

---

(3) "On Implementation of Label Varial ," by Robert R. Fenichel, Comm ACM 14, 5 (May 1971), p349-

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.

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 able to be confused with 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 oposed solution defines a new instruction (1) which takes the upper and lower bound in a pair of 18-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 wo[illegible] be required to specify the 4 possible even/odd index registe[illegible]rs (0-1, 2-3, 4-5, and 6-7). This may still be thought of [illegible]one instruction, however.

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 +.

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 futher 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

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.

```
(subscriptrange):
seal_code_generator_:
         procedure(work_seg_ptr, error_printer);

         dcl subscriptrange condition;
         on subscriptrange
         begin;
                  call ioa_("Subscriptrange in cg. debug:");
                  call debug;
         end;

/* Code generation program for the SEAL compiler.

         Paul 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      debug entry(),        /* temporary */
         ioa_ entry options(variable); /* temporary */

dcl      com_err_ entry options(variable),
         establish_cleanup_proc_ entry(entry),
         hcs_$make_seg entry(char(*) aligned, char(*) aligned, char(*) aligned, fixed bin, ptr, fixed bin(35)),
         hcs_$terminate_noname entry(ptr, fixed bin(35)),
         hcs_$truncate_seg entry(ptr, fixed, fixed bin(35)),
         seal_display_macro entry(ptr, fixed, bit(1));

/* automatic items */

dcl      (defs, defs_rel, link, link_rel, name, object_seg, operand_info_ptr, operator_info_ptr,
          p, pattern_base, previous_definition, scratch(8),
          sp, stack_base, symb, symb_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, dims, element_size, i, ir, j, 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,
          segname_definition, zero_definition) bit(18) aligned,
         use fixed initial(0),  /* temp */
         code fixed bin(35),
```

```
        (before_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;

/* builtin functions */

dcl     (addr, addrel, 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(bitlen) aligned,
        word_copy_image dim(n) fixed bin(35) aligned based,
        based_string char(262144) aligned based,

        1 unpacked_relocation           aligned based,
          2 half_word                   dim(0:m-1) unaligned bit(18),

        1 word                  aligned based,
          2 left_half           unal bit(18),
          2 right_half          unal bit(18),

        1 words                 aligned based,
          2 first               bit(36),
          2 second              bit(36),
          2 third               bit(36);

dcl 1 operator_info             aligned based(operator_info_ptr),
    2 offset                    unal fixed bin(17),
    2 type1                     unal bit(3),
    2 type2                     unal bit(3),
    2 length                    unal fixed bin(11),

    1 operand_semantics         dim(23) aligned based(operand_info_ptr),
    2 operands                  unal,
    3 opnd1                     bit(9),
    3 opnd2                     bit(9),
    2 pattern_offset            unal fixed bin(17).
```

```
    1 pattern                  dim(100) 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(8),
    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) */

dcl (     no_check_code        initial("000000000"b),        /* 0 */
          /* types 1-7 are for builtin modes */              /* 1-7 */
          ref_code             initial("000001000"b),        /* 8 */
          list_code            initial("000001001"b),        /* 9 */
          constant_code        initial("000001010"b),        /* 10 */
          user_mode_code       initial("000001011"b))        /* 11 */
          bit(9) aligned static,

          stack_frame_first_available_location fixed bin static initial(56),    /* must be even! */

          even_offset_required(8) bit(1) aligned static
          initial("1"b, "0"b, "0"b, "0"b, "1"b, "1"b, "0"b, "1"b);

dcl       empty_register fixed static initial(0),
```

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

        mode_length(7) fixed static init(1,1,1,1,6,2,64),
        max_no_dims fixed init(15) aligned static,
        my_name char(4) aligned internal static initial("seal"),
        (activation_record_up_zero_indirect     initial("110000000000000000000000000001010000"b), /* pr6|0,* */
        activation_record_up_zero               initial("110000000000000000000000000001000000"b), /* pr6|0   */
        linkage_section_up_zero_indirect        initial("100000000000000000000000000001010000"b), /* pr4|0,* */
        arg_list_up_zero_indirect               initial("001000000000000000000000000001010000"b), /* pr1|0,* */
        itp_to_activation_record_ft3            initial("110000000000000000000000000000100111"b),
        its_to_temporary_storage_ft3            initial("000000000000000000000000000000100111"b))
                  bit(36) aligned static;

dcl     store_op bit(10) dim(20) static aligned
                  initial(  "1111011010"b,      /* sta  755(0) */
                            "1111011100"b,      /* stq  756(0) */
                            "1001011110"b,      /* dfst 457(0) */
                            (10)"0"b,
                            "1001000000"b,      /* sxl0 440(0) */
                            "1001000010"b,      /* sxl1 441(0) */
                            "1001000100"b,      /* sxl2 442(0) */
                            "1001000110"b,      /* sxl3 443(0) */
                            "1001001000"b,      /* sxl4 444(0) */
                            "1001001010"b,      /* sxl5 445(0) */
                            "1001001100"b,      /* sxl6 446(0) */
                            "1001001110"b,      /* sxl7 447(0) */
                            "0101010000"b,      /* spri0 250(0) */
                            "0101010011"b,      /* spri1 251(1) */
                            "0101010100"b,      /* spri2 252(0) */
                            "0101010111"b,      /* spri3 253(1) */
                            "1101010000"b,      /* spri4 650(0) */
                            "1101010011"b,      /* spri5 651(1) */
                            "1100100100"b,      /* spri6 652(0) */
                            "1101010111"b);     /* spri7 653(1) */

/* These codes are for use with the register information entries. */

dcl (    A_reg     initial(1),
         Q_reg     initial(2),
         EAQ_reg   initial(3),
         Any_reg   initial(4),
         X0_reg    initial(5),
         X1_reg    initial(6),
         X2_reg    initial(7),
         X3_reg    initial(8),
```

```
        X4_reg    initial(9),
        X5_reg    initial(10),
        X6_reg    initial(11),
        X7_reg    initial(12),
        AP_reg    initial(13),
        AB_reg    initial(14),
        BP_reg    initial(15),
        BB_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_$ character(256) varying aligned) external static;

/* include files */

/* Declarations for the Multics Standard Object Segment */
/* Reference Multics Programmer's Manual, Part III */

dcl     mapp ptr,
        object_map_offset bit(18) unaligned based;

/* BEGIN INCLUDE SEGMENT ... obj_map.incl.pl1
coded February 8, 1972 by Michael J. Spier        */
/* last modified May, 1972 by M. Weaver */

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

```
          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(14) 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 userid            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_def           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 symb_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 */
```

```
            3 ignore            bit,                      /* always on in header */
            3 unused            bit(16);

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

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

dcl       1 type_pair           aligned based,
            2 type              unal bit(18),
            2 trap              unal bit(18),
            2 segname           unal bit(18),
            2 entryname         unal bit(18);

dcl       1 source_map          aligned based,
          2 version             fixed bin,
          2 number              fixed bin,
          2 map(n refer(source_map.number))        aligned,
            3 pathname          unaligned,
              4 offset          bit(18),
              4 size            bit(18),
            3 uid               bit(36),
            3 dtm               fixed bin(71);

dcl        1 relocation         aligned based,
             2 left             unal bit(18),
             2 right            unal bit(18),

           1 packed_relocation aligned based,
             2 dcl_version      fixed bin,
             2 string           bit(36*65536) varying,

          (rel_absolute         initial("00000"b),
```

```
             rel_text             initial("10000"b),
             rel_link18           initial("10010"b),
             rel_negative_link18  initial("10011"b),
             rel_link15           initial("10100"b),
             rel_defs             initial("10101"b),
             rel_symbol           initial("10110"b),
             rel_negative_symbol  initial("10111"b),
             rel_int_storage18    initial("11000"b),
             rel_int_storage15    initial("11001"b),
             rel_self             initial("11010"b),
             rel_exp_absolute     initial("11110"b)) 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,              /* words which immediately follow entry instructions */
             2 stack_offset       unal bit(18),               /* offset (rel to base of text) of template stack frame */
             2 stack_size         unal bit(18);               /* 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("000011"b),     /* 03 */  /* Instruction Modifiers */
              dl_mod     init("000111"b),     /* 07 */
              ic_mod     init("000100"b),     /* 04 */
              in_mod     init("010000"b),     /* 20 */
              au_mod     init("000001"b),     /* 01 */
              al_mod     init("000101"b),     /* 05 */
              qu_mod     init("000010"b),     /* 02 */
              ql_mod     init("000110"b),     /* 06 */

              no_mod     init("000000"b),     /* 00 */
              x0_mod     init("001000"b),     /* 10 */
              x1_mod     init("001001"b),     /* 11 */
              x2_mod     init("001010"b),     /* 12 */
              x3_mod     init("001011"b),     /* 13 */
              x4_mod     init("001100"b),     /* 14 */
              x5_mod     init("001101"b),     /* 15 */
```

```
        x6_mod     init("001110"b),    /* 16 */
        x7_mod     init("001111"b),    /* 17 */

        its_mod    init("100011"b),    /* 43 */  /* Pointer Modifiers */
        itp_mod    init("100001"b),    /* 41 */
        ft2_mod    init("100110"b),    /* 46 */
        ft3_mod    init("100111"b))    /* 47 */
                   bit(6) aligned internal static;

dcl (   r_tag      init("00"b),                  /* First 2 bits of Modifier field */
        ri_tag     init("01"b),        /* du, dl not allowed */
        ir_tag     init("11"b),
        it_tag     init("10"b))
                   bit(2) aligned static;

dcl (   ap_mod     init("000"b),       /* 0 */   /* Pointer Registers */
        ab_mod     init("001"b),       /* 1 */
        bp_mod     init("010"b),       /* 2 */
        bb_mod     init("011"b),       /* 3 */
        lp_mod     init("100"b),       /* 4 */
        lb_mod     init("101"b),       /* 5 */
        sp_mod     init("110"b),       /* 6 */
        sb_mod     init("111"b))       /* 7 */
                   bit(3) aligned static;

dcl  1 its_model   aligned based,
       2 (unused1  bit(3),
         segment   bit(15),
         ring      bit(3),
         unused2   bit(9),
         its       bit(6),
         offset    bit(18),
         unused3   bit(3),
         bit       bit(6),
         unused4   bit(9)) unaligned;

dcl  1 packed_ptr_model aligned based,
       2 (bit      bit(6),
         segment   bit(12),
         offset    bit(18)) unaligned;

dcl  1 itp_model   aligned based,
       2 (ptr_reg  bit(3),
         unused1   bit(27),
         itp       bit(6),
         offset    bit(18),
         unused2   bit(3),
```

```
        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(addr(opcodes));

/* In the comment defining each operator: i is an index to another macro or to
   a symbol node, s 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,i                    */
        2 add                  fixed initial(2),   /* add i,i                       */
        2 sub                  fixed initial(3),   /* sub i,i                       */
        2 divide               fixed initial(4),   /* divide i,i                    */
        2 mult                 fixed initial(5),   /* mult i,i                      */
        2 and                  fixed initial(6),   /* and i,i                       */
        2 or                   fixed initial(7),   /* or i,i                        */
        2 catenate             fixed initial(8),   /* catenate i,i                  */
        2 catenate_symbol      fixed initial(9),   /* catenate_symbol i,i           */
        2 less_than            fixed initial(10),  /* less_than i,i                 */
        2 greater_than         fixed initial(11),  /* greater_than i,i              */
        2 less_or_equal        fixed initial(12),  /* less_or_equal i,i             */
        2 greater_or_equal     fixed initial(13),  /* greater_or_equal i,i          */
        2 equal                fixed initial(14),  /* equal i,i                     */
        2 not_equal            fixed initial(15),  /* not_equal i,i                 */
        2 shape                fixed initial(16),  /* shape i,i                     */
        2 exponentiate         fixed initial(17),  /* exponentiate i,i              */
        2 complement           fixed initial(18),  /* complement i                  */
        2 deref                fixed initial(19),  /* deref i                       */
        2 negate               fixed initial(20),  /* negate i                      */
        2 lock                 fixed initial(21),  /* lock i                        */
        2 unlock               fixed initial(22),  /* unlock i                      */
        2 test_lock            fixed initial(23),  /* test_lock i,s                 */
        2 case_of              fixed initial(24),  /* case_of n,i                   */
        2 caselimit            fixed initial(25),  /* caselimit n,i                 */
        2 casejump             fixed initial(26),  /* casejump i,n                  */
        2 branch               fixed initial(27),  /* branch s                      */
        2 branch_true          fixed initial(28),  /* branch_true s,i               */
        2 branch_false         fixed initial(29),  /* branch_false s,i              */
```

```
2 label              fixed initial(30),  /* label s                    */
2 procedure          fixed initial(31),  /* procedure s                */
2 end                fixed initial(32),  /* end                        */
2 link               fixed initial(33),  /* link i                     */
2 unused_34          fixed initial(34),
2 element            fixed initial(35),  /* element i                  */
2 list               fixed initial(36),  /* list n                     */
2 arg                fixed initial(37),  /* arg i                      */
2 call               fixed initial(38),  /* call i,i                   */
2 ret                fixed initial(39),  /* ret i                      */
2 reduce             fixed initial(40),  /* reduce s,i                 */
2 block              fixed initial(41),  /* block b                    */
2 select             fixed initial(42),  /* select s,i                 */
2 nop                fixed initial(43),  /* nop i                      */
2 mode_select        fixed initial(44),  /* mode_select s,i            */
2 line_number        fixed initial(45),  /* line_number n'             */
2 addr               fixed initial(46),  /* addr i                     */
2 encode_dims        fixed initial(47),  /* encode_dims i,n            */
2 encode_value       fixed initial(48),  /* encode_value i,i           */
2 arg_list           fixed initial(49),  /* arg_list n                 */
2 encode_mode        fixed initial(50),  /* encode_mode s,n            */
2 split_prep         fixed initial(51),  /* split_prep i,i             */
2 unused_op          fixed dim(52:59),
                                         /* built-in opcodes follow    */
                                         /* zero or one argument bifs  */
2 current            fixed initial(60),  /* current [i]                */
2 errortrap          fixed initial(61),  /* errortrap [i]              */
2 incolumn           fixed initial(62),  /* incolumn [i]               */
2 infilemark         fixed initial(63),  /* infilemark [i]             */
2 initem             fixed initial(64),  /* initem [i]                 */
2 initemmark         fixed initial(65),  /* initemmark [i]             */
2 inlinemark         fixed initial(66),  /* inlinemark [i]             */
2 inpagemark         fixed initial(67),  /* inpagemark [i]             */
2 instream           fixed initial(68),  /* instream [i]               */
2 linesize           fixed initial(69),  /* linesize [i]               */
2 outcolumn          fixed initial(70),  /* outcolumn [i]              */
2 outfilemark        fixed initial(71),  /* outfilemark [i]            */
2 outitem            fixed initial(72),  /* outitem [i]                */
2 outitemmark        fixed initial(73),  /* outitemmark [i]            */
2 outlinemark        fixed initial(74),  /* outlinemark [i]            */
2 outpagemark        fixed initial(75),  /* outpagemark [i]            */
2 outstream          fixed initial(76),  /* outstream [i]              */
2 pagesize           fixed initial(77),  /* pagesize [i]               */
2 unused_bif0        fixed dim(78:79),
                                         /* single argument bifs       */
2 abs                fixed initial(80),  /* abs i                      */
2 atan               fixed initial(81),  /* atan i                     */
```

```
        2 boolean            fixed initial(82),  /* boolean l                    */
        2 ceil               fixed initial(83),  /* ceil l                       */
        2 cos                fixed initial(84),  /* cos l                        */
        2 delete             fixed initial(85),  /* delete l                     */
        2 deletedir          fixed initial(86),  /* deletedir l                  */
        2 detach             fixed initial(87),  /* detach l                     */
        2 exp                fixed initial(88),  /* exp l                        */
        2 find               fixed initial(89),  /* find l                       */
        2 floor              fixed initial(90),  /* floor l                      */
        2 integer            fixed initial(91),  /* integer l                    */
        2 isvoid             fixed initial(92),  /* isvoid l                     */
        2 length             fixed initial(93),  /* length l                     */
        2 log                fixed initial(94),  /* log l                        */
        2 log10              fixed initial(95),  /* log10 l                      */
        2 rank               fixed initial(96),  /* rank l                       */
        2 real               fixed initial(97),  /* real l                       */
        2 sign               fixed initial(98),  /* sign l                       */
        2 sin                fixed initial(99),  /* sin l                        */
        2 size               fixed initial(100), /* size l                       */
        2 sqrt               fixed initial(101), /* sqrt l                       */
        2 symbol             fixed initial(102), /* symbol l                     */
        2 tan                fixed initial(103), /* tan l                        */
        2 trunc              fixed initial(104), /* trunc l                      */
        2 unused_bif1        fixed dim(105:109),
                                                 /* multiple arg bifs            */
        2 create             fixed initial(110), /* create l,l                   */
        2 is                 fixed initial(111), /* is l,l                       */
        2 get                fixed initial(112), /* get [l]                      */
        2 put                fixed initial(113), /* put [l]                      */
        2 void               fixed initial(114), /* void l [l]                   */
        2 split              fixed initial(115), /* split l,l                    */
        2 unused_bifn        fixed dim(116:119),
                                                 /* two argument bifs            */
        2 attach             fixed initial(120), /* attach l,l                   */
        2 createdir          fixed initial(121), /* createdir l,l                */
        2 edit               fixed initial(122), /* edit l,l                     */
        2 max                fixed initial(123), /* max l,l                      */
        2 min                fixed initial(124), /* min l,l                      */
        2 mod                fixed initial(125), /* mod l,l                      */
        2 rename             fixed initial(126), /* rename l,l                   */
        2 round              fixed initial(127); /* round l,l                    */

/* This variable defines the first optional argument bif.  */

declare   first_bif0 fixed int static initial(60);

/* This variable defines the first single argument bif.  */
```

```
declare   first_bif1 fixed int static initial(80);

/* This variable defines the first two argument bif.  */

declare   first_bif2 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,
            3 brief_option     bit,
            3 debug_option     bit,
            3 list_option      bit,
            3 stop_cg_option   bit,
            3 parse_option     bit,
          2 stop_on_macro      fixed,
          2 stop_on_line       fixed,
          2 invocation         fixed,          /* number of activations of the compiler */
          2 greatest_severity  fixed,          /* greatest severity error encountered */
          2 severity_plateau   fixed,          /* cutoff for error printing */
          2 constant_list      dim(3) fixed,   /* index to the chain of symbol nodes that represent literal constants. */
          2 last_block         fixed,          /* The index of the last block */
          2 last_symbol        fixed,          /* The index of the last symbol */
          2 last_free          fixed,          /* The index of the last free_space */
          2 last_ir            fixed,          /* The index of the last macro */
          2 list_seg_index     fixed,          /* index in listing segment */
          2 source_seg_limit   fixed(24),      /* length of source segment */
          2 output_seg_length  fixed(24),      /* length of object segment */
          2 source_seg         pointer,        /* pointer to source segment */
          2 list_seg           pointer,        /* pointer to listing segment */
          2 output_seg         pointer,        /* pointer to output segment */
          2 options            pointer,        /* options used in compilation */
          2 user_id            pointer,        /* name of user compiling */

          2 object_info        aligned,
            3 pathname         pointer,        /* pathname of source segment */
            3 segname          pointer,        /* name of source segment */
            3 clock_time       fixed bin(71),  /* clock reading for compilation */
            3 dtm              fixed bin(71),  /* dtm of segment being compiled */
            3 uid              bit(36),        /* 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 identifers and keywords. */

          2 block              dim(0:127),
```

```
    3 identifiers     dim(97:122) fixed,  /* index to a chain of symbol nodes denoting identifiers of this block. */
    3 keywords        dim(65:90) fixed,   /* index to a chain of symbol nodes denoting keywords of this block. */
    3 level           fixed,              /* the nesting level of this block.  */
    3 free_temps      dim(3) fixed,       /* list of available temporaries, ordered by number of words. */
    3 stack_base      unal pointer,       /* base of template stack frame for this block */
    3 stack_end       fixed(18),          /* size in words of template stack frame for this block */
    3 stack_size      fixed(18),          /* size in words of stack frame for this block (includes temporaries)
    3 temporary_end   fixed(18),          /* last temporary location in use */
    3 entry_location  fixed(18),          /* offset in text of first executable instruction in this block */

  2 symbol            dim(2048),          /* Each element represents an identifier, keyword or literal. */
    3 name            pointer,            /* ptr to the character-string representation of the symbol. */
    3 value           pointer,            /* ptr to of the value of a literal constant. */
    3 cross_refs      fixed,              /* index to the chain of cross-references to this name
    3 cross_end       fixed,              /* index to the end of the cross-reference chain.  */
    3 def_line        fixed,              /* line on which this item was defined.  */
    3 mode            fixed,              /* index to the declaration of this item's mode. */
    3 next            fixed,              /* index to the next item in this chain of symbol nodes. */
    3 location        bit(36),            /* position of a formal parameter or address of  an item. */
    3 count           fixed,              /* For variables and mode components, this is the dimensionality.
                                             For infix operators, this is the precedence. */
    3 general         fixed,              /* For external procedure constants, this is the index to the link macro.
                                             For named constants, this is the index to the macros produced
                                             by value_parser. For operator definitions, this is the index to the
                                             macros produced by procedure_body_parser.  For mode definitions, this
                                             is the index to the chain of symbols representing the components. */
    3 left_relocation bit(5),             /* relocation for "location" field */
    3 attributes,
      4 ref            bit,
      4 list_ref       bit,
      4 label          bit,
      4 variable       bit,
      4 constant       bit,
      4 component      bit,
      4 mode_def       bit,
      4 infix_def      bit,
      4 prefix_def     bit,
      4 external       bit,
      4 input          bit,
      4 output         bit,
      4 defined        bit,
      4 set            bit,
      4 referenced     bit,
      4 runtime_allocate
                       bit,
      4 passed_as_arg  bit,
```

```
        2 register          dim(20),
          3 mode            unal fixed,          /* mode of value */
          3 contents        unal fixed,          /* macro output<0, empty=0, symbol>0 */

        2 temporary         dim(100),
          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             dim(10000),
          3 count           unal fixed,          /* reference count of macro.output */
          3 opcode          unal fixed,          /* operation to perform on operands */
          3 opnd1           unal fixed,          /* macro output<0, empty=0, (symbol | block | constant)
          3 opnd2           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 fs 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(10000) based(os),
          2 name               pointer,
          2 type               fixed,
          2 size               fixed,
          2 keyword            bit,
          2 constant           bit;

declare   tstring              char(token(k).size) unaligned based;

declare   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 bin(14),
      2 char                   unal bit(2),
```

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

dcl 1 value                     aligned based,
        2 mode                  unal bit(18),
        2 reference             unal bit,
        2 list_reference        unal bit,
        2 external              unal bit,            /* value is in external format */
        2 constant              unal bit,
        2 used                  unal bit,            /* for garbage collector */
        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               based pointer aligned,
        integer_model           based fixed binary(35,0),
        real_model              based float binary(63),
        boolean_model           based bit(1) aligned,
        gate_model              based bit(36) aligned,

    1 procedure_model           based structure,
        2 text                  pointer,
        2 stack                 pointer,
        2 creation              fixed bin(71),

        symbol_model            based character(256) varying aligned,

    1 list_model                based structure,    /* internal list */
        2 bound                 unal fixed bin(17), /* current size in words */
        2 size                  unal fixed bin(17), /* maximum size in words */
        2 element               unaligned ptr dim(asize refer(list_model.bound)),

    1 ext_list_model            based structure,
        2 bound                 unal fixed bin(17), /* current size in words */
        2 size                  unal fixed bin(17), /* maximum size in words */
        2 ext_element           dim(asize refer(ext_list_model.bound)),
          3 offset              unal fixed bin(17),
          3 unused              unal bit(18),

        ref_model               based aligned ptr,  /* internal reference */

    1 ext_ref_model             based structure,
```

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

dcl         seal_code_generator_          entry(ptr, entry);
                  /* arg 1  (input)       work seg ptr */
                  /* arg 2  (input)       error printing routine */

dcl         seal_listing_                 entry(ptr, char(*));
                  /* arg 1  (input)       work seg ptr */
                  /* arg 2  (input)       string to be printed */

dcl         seal_listing_$source          entry(ptr, fixed, fixed, fixed);
                  /* 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 */
```

```
/* program */

        ws = work_seg_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;                                    /* zero addresses are illegal */

        /* 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); /* NB: knows that first def is next */
        defs_rel->definition_header.definition_list = rel_defs;

        defs->definition_header.new,
        defs->definition_header.ignore = "1"b;

        /* generate segname definition */

        previous_definition = null;
        call allocate_definition;

        segname_definition = rel(defs);
        name_definition = store_def(storage.segname->vstring);
```

```
defs->definition.value = zero_definition;          /* for class 3 this is the segname thread */
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);       /* NB: knows that next def is next */
defs_rel->definition.segname = rel_defs;           /* for class 3 segname field is offset of
                                                      first non-class-3 definition */

/* Initialize symbol section */

symb_lc = 0;

/* Initialize linkage section */

link_lc = size(link->linkage_header);

/* Initialize variables */

temporary_count = 0;

do i = 1 to hbound(operand(*), 1);
     operand(i) = 0;
end;
```

```
create_scratch_: proc(name) returns(pointer);

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

          call hcs_$make_seg("", "seal."||substr("123456789",storage.invocation,1)||"."||name||"_",
                    "",1011b,p,code);
          if p = null
          then do;
                    call com_err_(code, my_name, "Trying to create scratch segment in process direc[illegible]");
                    call print(112);                    /* fatal error -- abort & unwind */
               end;

          call hcs_$truncate_seg(p,0,code);

          scratch_index = scratch_index + 1;
          scratch(scratch_index) = p;

          return(p);

end create_scratch_;



clean_up: procedure;

dcl       i fixed bin;

          do i = 1 to scratch_index;
               call hcs_$truncate_seg(scratch(i), 0, code);
               call hcs_$terminate_noname(scratch(i), code);
          end;

          return;

end clean_up;
```

```
/* Allocate all constants in the text section */

        do j = 3 to 1 by -1;
             do s = constant_list(j) repeat symbol(s).next while(s ^= 0);
                  vp = symbol(s).value;

                  if symbol(s).passed_as_arg
                  then do;
                                 string(value_temp) = "0"b;
                                 value_temp.mode = allocate_mode_(symbol(s).mode);
                                 value_temp.constant = "1"b;
                                 value_temp.input = "1"b;

                                 if j = 2            /* even value offset required on this list */
                                 then if text_ic = 2 * divide(text_ic, 2, 18, 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 j = 2
                  then text_ic = 2 * divide(text_ic + 1, 2, 18, 0);

                  symbol(s).location = bit(text_ic, 18);
                  symbol(s).left_relocation = rel_text;

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

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

                                      if m ^= 0
                                      then n = m;

                                      text->vstring = text->vstring || substr(vp->vstring, i, n);
                                      i = i + n + 1;
                                      l = l - n - 1;
                                      n = l - n;
```

```
                    end;
                    text_ic = text_ic + divide(length(text->vstring) + 3, 4, 17, 0);
               end;
          else do;
                    n = mode_length(symbol(s).mode);
                    pointer(text, text_ic)->word_copy_image = vp->word_copy_image;
                    text_ic = text_ic + n;
               end;
     end;
end;
```

```
/* The SEAL stack frame is organized into the following units, in order:
        standard Multics stack frame header.
        seal_operators_ work space.
        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 pointers (initialized with fault_tag_3's).
        value headers

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

   The storage allocater 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 = lbound(block(0).identifiers(*), 2) to hbound(block(0).identifiers(*), 2);
                  do s = block(b).identifiers(i) repeat symbol(s).next while(s^=0);
                       string(value_temp) = "0"b;
                       if symbol(s).referenced
                       then if symbol(s).variable
                            then if symbol(s).input | symbol(s).output
                                 then do;
                                           /* Allocate a parameter */

                                           symbol(s).defined = "1"b;          /* suppress sxl1/tsx1 instruction */
                                           position = binary(substr(symbol(s).location, 19, 18), 18) * 2;
                                           symbol(s).location = bit(position, 18) | arg_list_up_zero_indirect;
                                      end;
                                 else do;
                                           /* Allocate a variable */

                                           if ^symbol(s).ref
```

```
               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_dims
                         then do;
                                   call print(100);          /* too many dimensions */
                                   dims = max_no_dims;
                                   symbol(s).count = max_no_dims;
                              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, 18)
                              | activation_record_up_zero_indirect;
               p = addrel(stack_base, stack_offset - stack_frame_first_available_location);
               p->temporary_seal_name.value_header = string(value_temp);
               p->temporary_seal_name.element_size = element_size;
               if symbol(s).mode > symbol_type
               then k = hbound(even_offset_required(*), 1);
               else k = symbol(s).mode;
               p->temporary_seal_name.make_offset_even = even_offset_required(k);
               p->temporary_seal_name.runtime_allocate = symbol(s).runtime_allocate;
               stack_offset = stack_offset + size(p->temporary_seal_name);
          end;
else if symbol(s).constant & symbol(s).mode ^= proc_type
     then do;
               /* Allocate a named constant */
```

```
                                        j = symbol(s).general;
                                        symbol(s).location = symbol(j).location;
                                        symbol(s).left_relocation = symbol(j).left_relocation;
                                   end;
          end;
     end;
     block(b).stack_end = stack_offset - stack_frame_first_available_location;

     /* Begin pass two. */

     value_offset, i, stack_end = stack_offset;
     stack_offset = stack_frame_first_available_location;
     do while(stack_offset < stack_end);
          name = addrel(stack_base, stack_offset - stack_frame_first_available_location);
          p = addrel(stack_base, i - stack_frame_first_available_location);
          string(p->value) = name->temporary_seal_name.value_header;
          element_size = name->temporary_seal_name.element_size;
          if name->temporary_seal_name.runtime_allocate
          then do;
                    string(name->its_model) = its_to_temporary_storage_ft3 || bit(binary(1, 18), 18);
                    name->its_model.segment = bit(binary(rel(p), 15), 15);
               end;
          else do;
                    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 preeceeding word,
                                           which is odd. Since the offset is now even, add 1
                                           to get the desired odd boundary for the value header. */

                                        value_offset = value_offset + 1;
                                   end;
                    value_offset = value_offset + 1;                  /* space for value header */
                    string(name->itp_model) = itp_to_activation_record_ft3 || bit(value_offset, 18);
                    value_offset = value_offset + element_size;
               end;
          stack_offset = stack_offset + size(name->itp_model);
          i = i + size(p->value);
     end;
     block(b).temporary_end = value_offset;
     block(b).stack_size = value_offset;
     block(b+1).stack_base = addrel(stack_base, 2 * divide(i-stack_frame_first_available_location+1, 2, 18, 0));
end;

s = last_symbol;
```

```
/* 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,
     (cp, p) ptr,
      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 do;
                   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 Q register code */
                   p->mode.string = bit(binary(m,4),4);
                   text_ic = text_ic + 1;
                   symbol(m).location = rel(p);
```

```
                    symbol(m).left_relocation = rel_text;
                    end;
               else;                                         /* already encoded */
          else do;

/* This mode 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 Q register code */
               p->mode.string = left_parn;
               bitlen = length(symbol(m).name->vstring)*9;
               p->mode.string = p->mode.string || addrel(symbol(m).name,1)->image;

               do c = symbol(m).general repeat symbol(c).general while(c^=0);
                    cm = symbol(c).mode;
                    cp = pointer(text, substr(symbol(cm).location,1,18));
                    p->mode.string = p->mode.string || cp->mode.string;
                    bitlen = length(symbol(c).name->vstring) * 9;
                    p->mode.string = p->mode.string || addrel(symbol(c).name,1)->image;
               end;

               p->mode.string = p->mode.string || right_parn;
               text_ic = text_ic + divide(length(p->mode.string)+35,36,17,0);
               symbol(m).location = rel(p);
               symbol(m).left_relocation = rel_text;
               end;

          return(rel(p));

end allocate_mode_;
```

```
/* compile the program! */

        do ir = 1 to storage.last_ir;
             opcode = macro(ir).opcode;
             operand(1) = macro(ir).opnd1;
             operand(2) = macro(ir).opnd2;
             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 [illegible]ence 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 ioa_("stop on line ^d.^/debug:", stop_on_line);
                                    call debug;
                              end;
                        else;
                  else do;
                              /* Get information about this triple operator */

                              operator_info_ptr = addrel(addr(seal_patterns_$operator_table), opcode);

                              if stop_on_macro = ir
                              then do;
                                          call ioa_("stop on macro ^d.^/debug:", ir);
                                          call debug;
                                    end;

                              /* Get the modes 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;

        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(i

        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;
                                 text = pointer(text, text_ic);
                                 text_rel = pointer(text_rel, text_ic);
                                 text_ic = text_ic + 1;
                                 string(text->instruction) = string(p->instruction);
                                 left_relocation = rel_absolute;
                            end;
                       else do;
                                 /* Until a better way is found, assume duidl is OK. */

                                 call get_address(operand(binary(arg, 2)), "1"b);
                                 text = pointer(text  text_ic);
                                 text_rel = pointer(text_rel, text_ic):
                                 text_ic = text_ic + 1;
                                 string(text->instruction) = address;
                                 text->instruction.opcode = p->instruction.opcode;
                            end;
                       text_rel->relocation.left = left_relocation;
                  end;
        end;
     end;                          /* and do the next triple */
end;
```

```
/* 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(bit(9) 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).mode > symbol_type
                    then return(user_mode_code);
```

```
                    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(103);                   /* 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;
```

```
/* 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.
            (input)    direct_mod_allowed      N,dl or N,du may be used as address.
            (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, temp) 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(105);
                       address = (18)"1"b;                              /* let him try to address that! */
                       left_relocation = "0"b;
                       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, 18);

                                  /* use special relocation code so that undefined labels can be diagnosed later. */

                                  left_relocation = (6)"1"b || 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;
```

```
                              /* try to use a du or dl modifier */

                              p = symbol(opnd).value;
                              if p->word.left_half = "0"b
                              then do;
                                        address = p->word.right_half || (12)"0"b || dl_mod;
                                        left_relocation = rel_absolute;
                                        return;
                                   end;
                              else if p->word.right_half = "0"b
                                   then do;
                                             address = p->word.left_half || (12)"0"b ||
                                             left_relocation = rel_absolute;
                                             return;
                                        end;
                         end;

          /* normal case */

          address = symbol(opnd).location;
          left_relocation = symbol(opnd).left_relocation;

          if address = "0"b
          then call print(110);                              /* undefined address */
          return;
     end;
else do;
          /* operand is the output of another macro */

          i = -opnd;
          if macro(i).count > 0
          then macro(i).count = macro(i).count - 1;
          else do;
                    /* reference count <= 0 */

                    call print(106);
                    address = (18)"1"b;
                    left_relocation = "0"b;
                    return;
               end;

          opnd = macro(i).output;

          if opnd = 0
          then do;
                    /* macro has no output at this point */
```

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

          if opnd > 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 */

                              opnd = macro(-opnd).output;
                              address = temporary(opnd).location;
                              left_relocation = rel_absolute;
                         end;
                    return;
               end;
     end;
end get_address;
```

```
/* 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);

dcl       (register_index, temporary_index, value) fixed;

/* program */

          value = register(register_index).contents;

          if value = empty_register
          then return;
                                                       /* symbol node */
          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_ic);
                    text_ic = text_ic+1;
                    string(text->instruction) = 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;
```

```
interpret_pattern: proc;

/* automatic items */

dcl      j fixed bin,
         q pointer,
         offset fixed bin(18),
         (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(  "0110111010"b,        /* lca  335(0) */
                            "0110111100"b),       /* lcq  336(0) */

         load_op dim(20) static bit(10) aligned
                  initial(  "0100111010"b,        /* lda  235(0) */
                            "0100111100"b,        /* ldq  236(0) */
                            "1000110110"b,        /* dfld 433(0) */
                            (10)"0"b,
                            "1110100000"b,        /* ldx0 720(0) */
                            "1110100010"b,        /* ldx1 721(0) */
                            "1110100100"b,        /* ldx2 722(0) */
                            "1110100110"b,        /* ldx3 723(0) */
                            "1110101000"b,        /* ldx4 724(0) */
                            "1110101010"b,        /* ldx5 725(0) */
                            "1110101100"b,        /* ldx6 726(0) */
                            "1110101110"b,        /* ldx7 727(0) */
                            "0111010000"b,        /* epp0 350(0) */
                            "0111010011"b,        /* epp1 351(1) */
                            "0111010100"b,        /* epp2 352(0) */
                            "0111010111"b,        /* epp3 353(1) */
                            "0111110000"b,        /* epp4 370(0) */
                            "0111110011"b,        /* epp5 371(1) */
                            "0111110100"b,        /* epp6 372(0) */
                            "0111110111"b),       /* epp7 373(1) */

         (tra     initial("1110010000"b),        /* 710(0) */
          ora     initial("0101111010"b),        /* 275(0) */
          fld     initial("1000110010"b),        /* 431(0) */
          staq    initial("1111011110"b),        /* 757(0) */
          sxl1    initial("1001000010"b))        /* 441(0) */
                  bit(10) static aligned,
         nop_instruction bit(36) aligned static initial("000000000000000000000001001000000111"b),  /* 000000 011007 */
         is_zero fixed bin initial(0) static;
```

```
/* 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(5):                                      /* LOADC  - Register, Value */
        if p_op = 1
        then load_complement = "0"b;
        else load_complement = "1"b;

        /* 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 */

                j = -register(arg1).contents;
                if j > 0                                 /* decrement reference count because we have */
                then macro(j).count = macro(j).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);
```

```
        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 lcq N,dl for ldq -N */

                            q = symbol(arg2).value;
                            if q->word.left_half = (18)"1"b
                            then do;
                                       address = bit(binary(- q->integer_model, 18), 18) || (12)"0"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(3):                                    /* NOT_INREG - Register, Value */
        if p_op = 2
        then not_flag = "0"b;
        else not_flag = "1"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                                /* decrement reference count because we have */
                  then macro(j).count = macro(j).count - 1;       /* made arg1 addressable */
             end;
        return;
```

```
operator(3):                                    /* RESULT - Register, Mode     */
        j = register(arg1).contents;                   /* see if this is anyone's output */
        if j < 0
        then do;
                if macro(-j).count > 0
                then call print(109);                  /* macro output destroyed even though ref count ^= 0 */
                macro(-j).output = 0;
             end;
        if macro(ir).count > 0                         /* save register value only if it will be used */
        then do;
                register(arg1).mode = arg2;
                register(arg1).contents = -ir;         /* chain register to macro */
                macro(ir).output = -arg1;              /* chain macro to register */
             end;
        return;

operator(4):                                    /* SET_BLOCK - block index */
        b = arg1;
        return;

operator(5):                                    /* SETUP   - Mode, Value       */
        call get_address(arg2, "0"b);
        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_op(BP_reg);
        text_rel->relocation.left = left_relocation;
        return;

operator(7):                                    /* ERASE   - Register          */
        call save_register(arg1);
        return;

operator(8):                                    /* CASE_USAGE - Case #, Index of caselimit macro */
        j = macro(-arg2).output;
        offset = binary(substr(symbol(j).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(j).left_relocation;
        return;

operator(10):                                   /* ALLOCATE_CASE - Number of Cases      */
        s = s + 1;
        symbol(s).location = bit(text_ic, 18);
```

```
        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 *,ql */
        text_ic = text_ic + 1;
        text->instruction.opcode = tra;
        text->instruction.tag = ql_mod;
        text_rel->relocation.left = rel_text;
        text_ic = text_ic + arg1;
        return;

operator(11):                                       /* FILL_USAGE - index of label          */
        do j = 1 to hbound(register(*), 1);
             call save_register(j);
        end;
        offset = binary(substr(symbol(arg1).location, 1, 18));
        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 = "1"b;
        symbol(arg1).location = bit(text_ic, 18);
        symbol(arg1).left_relocation = rel_text;
        if "0"b 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, 18) | linkage_section_up_zero_indirect;
        symbol(arg1).left_relocation = rel_link15;
        link = pointer(link, link_ic);
```

```
        link_rel = pointer(link_rel, link_ic);

        /* Form the negative offset of this link in the linkage section, */
        /* in two's complement form, naturally. */
        substr(string(link->its_model), 1, 18) = ^bit(binary(link_ic - 1, 18), 18);
        link_rel->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_rel->its_model.offset = rel_defs;

        defs = pointer(defs, defs_ic);
        defs_rel = pointer(defs_rel, defs_ic);
        defs_ic = defs_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, defs_ic);
        defs_rel = pointer(defs_rel, defs_ic);
        defs_ic = defs_ic + size(defs->type_pair);

        defs->type_pair.type = "000000000000000100"b;     /* type 4 is segname$entryname */
        defs->type_pair.segname,
        defs->type_pair.entryname = store_def(symbol(arg1).name->vstring);
        defs_rel->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(b).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_rel = pointer(text_rel, text_ic - 1);
        text->entry_sequence.entry_definition = rel(defs);
        text_rel->entry_sequence.entry_definition = rel_defs;
```

```
        if symbol(arg1).external
        then defs->definition.entry = "1"b;

        defs->definition.retain = "1"b;                /* class 0 is a value relative to the text section */
        defs->definition.class = "000"b;

        defs->definition.symbol = name_definition;
        defs_rel->definition.symbol = rel_defs;

        defs->definition.segname = segname_definition;
        defs_rel->definition.segname = rel_defs;

        return;
                                          /* FILL_LIST - index of expression to put on list */
operator(15):
        return;
                                          /* ALLOCATE_LIST - */
operator(16):
        return;
                                          /* ARG_PTR - Register, Index of arg_list macro */
operator(17):
        text = pointer(text, text_ic);
        text_ic = text_ic + 1;

        call get_address(arg2, "0"b);
        position = binary(substr(address, 4, 15), 15) + 2 * argument_index;
        address = bit(position, 18) | activation_record_up_zero;
        string(text->instruction) = address;
        text->instruction.opcode = store_op(arg1);

        argument_index = argument_index + 1;
        return;
                                          /* ALLOCATE_ARG_LIST - number of arguments*/
operator(18):
        macro(ir).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 8 places into the au, which is where we want it. */
        substr(string(text->instruction), 1, 18) = bit(binary(2 * arg1, 8), 8);
        text->instruction.opcode = fld;
        text->instruction.tag = dl_mod;

        text = pointer(text, text_ic);
        text_ic = text_ic + 1;
        text->instruction.offset = "000000000000100"b;    /* 4 */
```

```
        text->instruction.opcode = ora;
        text->instruction.tag = dl_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 = staq;
        return;

                                              /* CALL */
operator(19):
                                              /* TYPE_CHECK */
operator(20):
        return;
                                              /* RETURN */
operator(21):
        done = "1"b;
        return;

                                              /* MODE_SELECT */
operator(22):
        return;
                                              /* GOTO */
operator(23):                                         /* increment loop index (do loop will add 1 back...) */
        pattern_ic = pattern_ic + arg3 - 1;
        return;

                                              /* SET_STACK_SIZE */
operator(24):
        text = pointer(text, block(b).entry_location);
        offset = block(b).stack_size;
        offset = 16 * divide(offset + 15, 16, 18, 0);     /* make stack size mod 16 */
        text->instruction.offset = bit(binary(offset, 15), 15);
        return;

                                              /* DEFINE - Value */
operator(25):                                         /* 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;
             else return;
        call get_address(arg1, "0"b);                 /* turn off ,* --- we want to store into the name */
        substr(address, 31, 6) = no_mod;
        text = pointer(text, text_ic);
        text_rel = pointer(text_rel, text_ic);
        text_ic = text_ic + 1;
        string(text->instruction) = address;          /* this register always contains an ITP tag */
        text->instruction.opcode = sxl1;
        text_rel->relocation.left = left_relocation;
```

```
        return;

operator(26):                                    /* ADD_REFERENCE - Value */
        if arg1 < 0                                      /* macro output ? */
        then if macro(-arg1).count <= 0
             then call print(111);                       /* attempt to add to reference count which is <= 0 */
             else macro(-arg1).count = macro(-arg1).count + 1;
        return;

operator(27):                                    /* NOTE_FLOWCHANGE */
        before_first_flowchange = "0"b;
        return;

operator(28):                                    /* IF_OPERAND - Value, check code */
        if arg2 = is_zero
        then do;
                if arg1 ^= 0
                then pattern_lc = pattern_lc + arg3 - 1;        /* take failure label */
             end;
        return;

end interpret_pattern;
```

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

get_temporary: proc(n) returns(fixed);

dcl       n fixed parameter,
          (even_offset, i, l) 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(n + 1, 2, 17, 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 =
                                                  bit(binary(i, 18), 18) || activation_record_up_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(block(b).temporary_end,18) || activation_record_up_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);
```

```
                return(temporary_count);
            end;
end get_temporary;
```

```
/* This subroutine is called to place a no-longer-needed stack temporary on the free list. */

free_temporary: proc(ft);

dcl       ft fixed parameter,
          (i,n) fixed;

          n = temporary(ft).size;
          if n > 2
          then i = 3;
          else i = n;

          temporary(ft).mode = block(b).free_temps(i);          /* push onto free list */
          block(b).free_temps(i) = 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;
```

```
/* 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);
                i = block(b).stack_end;
                n = i * 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 = i 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 + seal_patterns_$entry_control_word1_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(i, 2, 18, 0), 18);
                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 = "0"b;                    /* symbol block starts at symbol|0 */
        defs_rel->definition.value = rel_symbol;

        defs->definition.class = "010"b;                  /* class 2 is a value relative to symbol section */

        defs->definition.symbol = name_definition;
        defs_rel->definition.symbol = rel_defs;

        defs->definition.segname = segname_definition;
        defs_rel->definition.segname = rel_defs;
```

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

store_def: proc(def_string) returns(bit(18) unaligned);

dcl       def_string char(*) varying aligned,
          d pointer,
          1 name    aligned based,
          2 size    unal 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, 0);     /* length + roundup + s[illegible] */
          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, defs_ic);
          defs_ic = defs_ic + size(defs->definition);

          defs->definition.forward = zero_definition;
          defs_rel->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 = rel_defs;

          defs->definition.new = "1"b;
          previous_definition = defs;
          return;

end allocate_definition;
```

```
/* 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.definitions_offset = bit(text_ic, 18);   /* since defs follow text section in object */
        link_rel->linkage_header.definitions_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 + size(symb->seal_symbol_block);

        symb->seal_symbol_block.dcl_version = 1;
        symb->seal_symbol_block.identifier = "symbtree";
        symb->seal_symbol_block.gen_number = 1;            /* temp, for now */
        symb->seal_symbol_block.gen_created = addr(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_$);
        string(symb->seal_symbol_block.userid) = 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 = "000000000000000010"b;
        symb->seal_symbol_block.stat_boundary = "000000000000000010"b;

        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->seal_symbol_block.backpointer = "0"b;
        symb_rel->seal_symbol_block.backpointer = rel_negative_symbol;
        symb->seal_symbol_block.block_size = bit(binary(size(symb->seal_symbol_block), 18), 18);
```

```
/* Procedure to store a character string in the symbol section and return
   a "stringpointer" to it. */

store_symbol: proc(symbol_string) returns(bit(36) unaligned);

dcl       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;
```

```
/* Create the source map. */

          symb = pointer(symb, symb_ic);
          symb->source_map.version = 1;              /* the order of the next 3 statements is critical! */
          n, symb->source_map.number = 1;            /* this one sets n to be the number of source files */
          symb_ic = symb_ic + size(symb->source_map); /* this one uses n to allocate the source_map */
                                                     /* this one uses symb_ic to allocate the pathname */
          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 MPM Part III, Sections 11.3 and 11.4 */

          defs_ic = 2 * divide(defs_ic + 1, 2, 18, 0);
          link_ic = 2 * divide(link_ic + 1, 2, 18, 0);
          symb_ic = 2 * divide(symb_ic + 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_ic, 18);   /* save block & source_map, but not relocation bits */

          saved_symb_ic = symb_ic;                   /* don't generate relocation bits for the relocation bits! */

          i = 1;
          do p = text_rel, defs_rel, link_rel, symb_rel;
               p = pointer(p, 0);                    /* point to base of section */
               go to section(i);

section(1):
               m = text_ic;
               sp->seal_symbol_block.rel_text = bit(symb_ic, 18);
               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;
```

```
          sp->seal_symbol_block.rel_symbol = bit(symb_ic, 18);

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(j);
               if rel_code ^= ""b
               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)"1"b
                         then do;
                                   /* Address is on a usage chain which was never filled...undefined label. */
                                   /* Temporarily call ioa_ until seal error routine can take symbolic args */

                                   n = binary(substr(rel_code, 7, 12));
                                   call ioa_("seal_code_generator_: The label ""^a"" is undefined!",
                                        symbol(n).name->vstring);
                                   call print(113);
                                   text = pointer(text, divide(j, 2, 18, 0));        /* j 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;

          i = i + 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, 18);
```

```
/* 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("000000000000000"b, 1, k);
        else do;
                do while(k > 1023);
                     symb->packed_relocation.string = symb->packed_relocation.string
                                  || rel_exp_absolute || "1111111111"b;
                     k = k - 1023;
                end;

                symb->packed_relocation.string = symb->packed_relocation.string
                                  || rel_exp_absolute || bit(binary(k, 10), 10);

             end;

        return;

end expanded_absolute;
```

```
/* 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 */

          mapp = pointer(object_seg, object_ic);
          object_ic = object_ic + size(mapp->map);

          map.decl_vers = 1;
          map.identifier = "obj_map";

          map.text_offset = "0"b;
```

```
        map.text_length = bit(text_ic, 18);

        map.def_offset = bit(text_ic, 18);
        map.def_length = bit(defs_ic, 18);

        map.link_offset = bit(binary(text_ic + defs_ic, 18), 18);
        map.link_length = bit(link_ic, 18);

        map.symb_offset = bit(binary(text_ic + defs_ic + link_ic, 18), 18);
        map.symb_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(mapp);
        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 ioa_("lr = ^d^/debug:", lr);
        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 PG on 5/9/73
"
          followon
          name      seal_patterns_
          segdef    operator_table

" This is a list of the possible operand types.

          equ       N,0                      No check
          equ       A,1                      Any
          equ       B,2                      Boolean
          equ       G,3                      Gate
          equ       I,4                      Integer
          equ       P,5                      Proc
          equ       R,6                      Real
          equ       S,7                      Symbol
          equ       Re,8                     Ref
          equ       Li,9                     List
          equ       C,10                     Constant
          equ       U,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,    7
          equ       entry_operator,vec+8
          equ       complement_any_operator,vec+9
```

```
        equ     negate_any_operator,vec+10
        equ     return_operator,vec+11
        equ     check_mode_operator,vec+12
        equ     less_than_any_operator,vec+13
        equ     Integer_to_Real,vec+14
        equ     Real_to_Integer,vec+15
        equ     call_operator,vec+16
        equ     get_operator,vec+17
        equ     put_operator,vec+18
        equ     fixedoverflow_operator,vec+19
        equ     reload_registers_operator,vec+20
        equ     undefined_label_operator,vec+21

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

        equ     LOAD_OP,1
        equ     INREG_OP,2
        equ     RESULT_OP,3
        equ     SET_BLOCK_OP,4
        equ     SETUP_OP,5
        equ     LOADC_OP,6
        equ     ERASE_OP,7
        equ     NOT_INREG_OP,8
        equ     CASE_USAGE_OP,9
        equ     ALLOCATE_CASE_OP,10
        equ     FILL_USAGE_OP,11
        equ     NOP_OP,12
        equ     GEN_LINK_OP,13
        equ     GEN_DEF_OP,14
        equ     FILL_LIST_OP,15
        equ     ALLOCATE_LIST_OP,16
        equ     ARG_PTR_OP,17
        equ     ALLOCATE_ARG_LIST_OP,18
        equ     CALL_OP,19
        equ     TYPE_CHECK_OP,20
        equ     RETURN_OP,21
        equ     MODE_SELECT_OP,22
        equ     GOTO_OP,23
        equ     SET_STACK_SIZE_OP,24
        equ     DEFINE_OP,25
        equ     ADD_REFERENCE_OP,26
        equ     NOTE_FLOWCHANGE_OP,27
        equ     OPERAND_OP,28

"" These symbols are used by the OPERAND patt[illegible]n-op.
```

```
          equ       is_zero,0

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

          equ       A_reg,1
          equ       Q,2
          equ       EAQ,3
          equ       Any,4
          equ       K0,5
          equ       K1,6
          equ       K2,7
          equ       K3,8
          equ       K4,9
          equ       K5,10
          equ       K6,11
          equ       K7,12
          equ       AP,13
          equ       AB,14
          equ       BP,15
          equ       BB,16
          equ       LP,17
          equ       LB,18
          equ       SP,19
          equ       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       oi,1

" index, without mode check.
          equ       ni,3

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

" symbol: no mode check, operand is symbol inde
          equ       s,3

" block: no mode check, operand is block inde
          equ       b,3
```

Appendix C. Multics Seal Code Generator Table.

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

          bool      parg1,401
          bool      parg2,402

" 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      ap|undefined_label_operator

"
"         BEGIN INCLUDE FILE ... stack_frame.incl.alm
"
          equu      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_rel_ptrs  0
          equ       stack_frame.operator_ret_pt   1
          equ       stack_frame.translator_id
          equ       stack_frame.regs,32
          equ       stack_frame.min_length,48

"
```

```
"          END INCLUDE FILE  ... stack_frame.incl.alm
"
" 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

"           OPCODE,TYPE1,TYPE2,LENGTH

operator_table:
            define    unused_0,0,0,0
            define    assign,i,i,10
            define    add,i,i,5
            define    sub,i,i,5
            define    divide,i,i,5
            define    mult,i,i,5
            define    and,i,i,2
            define    or,i,i,2
            define    catenate,i,i,1
            define    catenate_symbol,i,i,2
            define    less_than,i,i,5
            define    greater_than,i,i,5
            define    less_or_equal,i,i,5
            define    greater_or_equal,i,i,5
            define    equal,i,i,8
            define    not_equal,i,i,8
            define    shape,i,i,8
            define    exponentiate,i,i,5
            define    complement,i,0,2
            define    deref,i,0,1
            define    negate,i,0,3
            define    lock,i,0,2
            define    unlock,i,0,2
            define    test_lock,i,s,2
            define    case_of,n,i,1
            define    caselimit,n,i,1
            define    casejump,i,n,1
            define    branch,s,0,1
            define    branch_true,s,i,2
            define    branch_false,s,i,2
            define    label,s,0,1
            define    procedure,s,b,1
            define    end,b,0,1
            define    link,i,0,1
            define    unused_34,0,0,0
            define    element,i,0,1
            define    list,n,0,1
            define    arg,ni,0,1
            define    call,i,i,1
            define    ret,i,0,9
            define    reduce,s,i,1
```

Appendix C. Multics Seal Code Generator Table.

```
define     block,b,0,1
define     select,s,i,2
define     nop,i,0,1
define     mode_select,s,i,1
define     line_number,n,0,1
define     addr,i,0,1
define     encode_dims,i,n,1
define     encode_value,i,i,1
define     arg_list,n,0,1
define     encode_mode,s,n,1
define     split_prep,i,i,0
define     unused_52,0,0,0
define     unused_53,0,0,0
define     unused_54,0,0,0
define     unused_55,0,0,0
define     unused_56,0,0,0
define     unused_57,0,0,0
define     unused_58,0,0,0
define     unused_59,0,0,0
define     current,oi,0,0
define     errortrap,oi,0,0
define     incolumn,oi,0,0
define     infilemark,oi,0,0
define     initem,oi,0,0
define     initemmark,oi,0,0
define     inlinemark,oi,0,0
define     inpagemark,oi,0,0
define     instream,oi,0,0
define     linesize,oi,0,0
define     outcolumn,oi,0,0
define     outfilemark,oi,0,0
define     outitem,oi,0,0
define     outitemmark,oi,0,0
define     outlinemark,oi,0,0
define     outpagemark,oi,0,0
define     outstream,oi,0,0
define     pagesize,oi,0,0
define     unused_78,0,0,0
define     unused_79,0,0,0
define     abs,i,0,3
define     atan,i,0,0
define     boolean,i,0,3
define     ceil,i,0,3
define     cos,i,0,0
define     delete,i,0,0
define     deletedir,i,0,0
define     detach,i,0,0
```

Appendix C. Multics Seal Code Generator Table.

```
define     exp,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_105,0,0,0
define     unused_106,0,0,0
define     unused_107,0,0,0
define     unused_108,0,0,0
define     unused_109,0,0,0
define     create,i,i,2
define     is,i,i,1
define     get,ni,0,1
define     put,ni,0,1
define     void,i,0,1
define     split,i,i,0
define     unused_116,0,0,0
define     unused_117,0,0,0
define     unused_118,0,0,0
define     unused_119,0,0,0
define     attach,i,i,0
define     createdir,i,i,0
define     edit,i,i,3
define     max,i,i,5
define     min,i,i,5
define     mod,i,i,5
define     rename,i,i,0
define     round,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.

"           OPERAND_1,OPERAND_2,PATTERN_OFFSET

unused_0:  def0        0

assign:    def         I,I,assign_II
           def         I,R,assign_IR
           def         R,I,assign_RI
           def         R,R,assign_RR
           def         B,B,assign_BB
           def         P,P,assign_PP
           def         S,S,assign_SS
           def         Re,Re,assign_ReRe
           def         Li,Li,assign_LiLi
           def         A,A,assign_AA

add:       def         I,I,add_II
           def         I,R,add_IR
           def         R,I,add_RI
           def         R,R,add_RR
           def         A,A,add_AA

sub:       def         I,I,sub_II
           def         I,R,sub_IR
           def         R,I,sub_RI
           def         R,R,sub_RR
           def         A,A,sub_AA

divide:    def         I,I,divide_II
           def         I,R,divide_IR
           def         R,I,divide_RI
           def         R,R,divide_RR
           def         A,A,divide_AA

mult:      def         I,I,multiply_II
           def         I,R,multiply_IR
           def         R,I,multiply_RI
           def         R,R,multiply_RR
           def         A,A,multiply_AA

and:       def         B,B,and_BB
           def         A,A,and_AA
```

```
or:         def        B,B,or_BB
            def        A,A,or_AA

catenate:   def        Li,Li,catenate_LiLi

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

" These triples perform the indicated relational test,and give a
" boolean output = "1"b if the test was true.
" In all cases,the comparison is operand1 :: operand2.

less_than:
            def        I,I,less_than_II
            def        I,R,less_than_IR
            def        R,I,less_than_RI
            def        R,R,less_than_RR
            def        A,A,less_than_AA

greater_than:
            def        I,I,greater_than_II
            def        I,R,greater_than_IR
            def        R,I,greater_than_RI
            def        R,R,greater_than_RR
            def        A,A,greater_than_AA

less_or_equal:
            def        I,I,less_or_equal_II
            def        I,R,less_or_equal_IR
            def        R,I,less_or_equal_RI
            def        R,R,less_or_equal_RR
            def        A,A,less_or_equal_AA

greater_or_equal:
            def        I,I,greater_or_equal_II
            def        I,R,greater_or_equal_IR
            def        R,I,greater_or_equal_RI
            def        R,R,greater_or_equal_RR
            def        A,A,greater_or_equal_AA

equal:      def        I,I,equal_II
            def        I,R,equal_IR
            def        R,I,equal_RI
            def        R,R,equal_RR
            def        B,B,equal_BB
            def        P,P,equal_PP
```

```
          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       Li,Li,shape_LiLi
          def       Li,I,shape_LiI
          def       Li,R,shape_LiR
          def       Li,B,shape_LiB
          def       Li,P,shape_LiP
          def       Li,S,shape_LiS
          def       Li,G,shape_LiG
          def       Li,A,shape_LiA

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:
          def1      B,complement_B
          def1      A,complement_A

deref:    def1      Re,deref_Re

negate:   def1      I,negate_I
          def1      R,negate_R
          def1      A,negate_A

lock:     def1      G,lock_G
          def1      A,lock_A

unlock:   def1      G,unlock_G
          def1      A,unlock_A

test_lock:
          def       G,N,test_lock_GN
```

```
            def         A,N,test_lock_AN

" fill in Nth slot in transfer vector I.
" operand 1 is case number.
" operand 2 is caselimit triple, transfer vector.

case_of:    def         C,N,case_of_CN

" This triple generates code to test expression.
" operand 1 is number of cases.
" operand 2 is case expression.
" output is True if the expression lies within
" the case range.
" False if it does not.

caselimit:
            def         C,I,caselimit_CI

" This triple actually performs the case transfer.
" operand 1 is case expression.
" operand 2 is number of cases.
" output is the transfer vector and case transfer instructions.

casejump:   def         I,C,casejump_IC

" This triple generates a transfer.
" operand 1 is the label.

branch:     def1        N,branch_N

" This triple generates a conditional transfer.
" operand 1 is the label.
" operand 2 is the boolean expression.

branch_true:
            def         N,B,branch_true_NB
            def         N,A,branch_true_NA

branch_false:
            def         N,B,branch_false_NB
            def         N,A,branch_false_NA

" This triple defines a label's address.
" operand1 is the symbol node of the label.

label:      def1        N,label_N
```

Appendix C. 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       N,N,procedure_NN

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

end:        def1      N,end_N

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

link:       def1      P,link_P

unused_34:
            def0      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.

element:    def1      N,element_N

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

list:       def1      C,list_C

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

arg:        def1      N,arg_N

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

call:       def       P,N,call_PN
            def       A,N,call_AN

ret:        def1      I,ret_I
            def1      R,ret_R
```

```
          def1      B,ret_B
          def1      P,ret_P
          def1      S,ret_S
          def1      Re,ret_Re
          def1      Li,ret_Li
          def1      A,ret_A
          def1      N,ret_N

" if operand 1 < 0 it is the index of a user-supplied operator,
" otherwise, operand 1 is the opcode of the language
" builtin infix operator.

reduce:   def       P,Li,reduce_PLi
          def       C,Li,reduce_CLi

" operand 1 is the index of the current block.

block:    def1      N,block_N

" This triple performs subscript checking
" and then subscript evaluation.
" operand 1 is the symbol node of the list.
" operand 2 is the subscript expression.
" output is the 'select'ed element.

select:   def       N,I,select_NI
          def       A,I,select_AI

" The output of this triple is just its input.

nop:      def1      N,nop_N

" This triple computes the index of a selector on a mode name.
" operand 1 is the symbol node of the Mode name.
" operand 2 is the selector.
" output is not yet determined.

mode_select:
          def       N,N,mode_select_NN

" operand 1 is line number of source program.
" no output.

line_number:
          def1      C,line_number_C

" This triple is currently unused.
```

Appendix C. Multics Seal Code Generator Table.

```
addr:      def1       N,addr_N

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

encode_dims:
           def        N,C,encode_dims_NC

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

encode_value:
           def        N,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:
           def1       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        N,C,encode_mode_NC

split_prep:
           def0       0

unused_52:
unused_53:
unused_54:
unused_55:
unused_56:
unused_57:
unused_58:
unused_59:
           def0       0
```

Appendix C. Multics Seal Code Generator Table.

```
" Zero or one-argument builtin functions.

current:
errortrap:
incolumn:
infilemark:
initem:
initemmark:
inlinemark:
inpagemark:
instream:
linesize:
outcolumn:
outfilemark:
outitem:
outitemmark:
outlinemark:
outpagemark:
outstream:
pagesize:
unused_78:
unused_79:
                def0      ]
```

Appendix C. Multics Seal Code Generator Table.

```
" Single argument builtin functions.

abs:        def1        I,abs_I
            def1        R,abs_R
            def1        A,abs_A

atan:       def1        I,atan_I
            def1        R,atan_R
            def1        A,atan_A

boolean:    def1        I,boolean_I
            def1        R,boolean_R
            def1        A,boolean_A

ceil:       def1        R,ceil_R
            def1        I,ceil_I
            def1        A,ceil_A

cos:        def1        I,cos_I
            def1        R,cos_R
            def1        A,cos_A

delete:     def1        S,delete_S

deletedir:
            def1        S,deletedir_S

detach:     def1        S,detach_S

exp:        def1        I,exp_I
            def1        R,exp_R
            def1        A,exp_A

find:       def1        S,find_S
            def1        A,find_A

floor:      def1        R,floor_R
            def1        I,floor_I
            def1        A,floor_A

integer:    def1        R,integer_R
            def1        I,integer_I
            def1        B,integer_B
            def1        S,integer_S
            def1        A,integer_A

isvoid:     def0        J
```

Appendix C. Multics Seal Code Generator Table.

```
length:   def1      Li,length_Li

log:      def1      I,log_I
          def1      R,log_R
          def1      A,log_A

log10:    def1      I,log10_I
          def1      R,log10_R
          def1      A,log10_A

rank:     def0      0

real:     def1      I,real_I
          def1      R,real_R
          def1      B,real_B
          def1      S,real_S
          def1      A,real_A

sign:     def1      I,sign_I
          def1      R,sign_R
          def1      A,sign_A

sin:      def1      I,sin_I
          def1      R,sin_R
          def1      A,sin_A

size:     def1      S,size_S
          def1      A,size_A

sqrt:     def1      I,sqrt_I
          def1      R,sqrt_R
          def1      A,sqrt_A

symbol:   def1      I,symbol_I
          def1      R,symbol_R
          def1      B,symbol_B
          def1      S,symbol_S
          def1      Re,symbol_Re
          def1      Li,symbol_Li
          def1      A,symbol_A

tan:      def1      I,tan_I
          def1      R,tan_R
          def1      A,tan_A

trunc:    def1      R,trunc_R
          def1      A,trunc_A
```

```
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_N0
          def      N,S,create_NS

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

is:       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:     def      N,N,void_NN

split:    def0     0

unused_116:
unused_117:
unused_118:
unused_119:
          def0     0
```

Appendix C. Multics Seal Code Generator Table.

```
" Two argument builtin functions.

attach:    def        S,S,attach_SS

createdir:
           def        S,S,createdir_SS

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

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

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

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

rename:    def        S,S,rename_SS

round:     def        R,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
"" by one 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       Q,arg2
           DEFINE     arg1
           stq        arg1
           RESULT     Q,I
           RETURN

assign_IR:
           ADD_REFERENCE        arg1
           LOAD       EAQ,arg2
           tsx0       ap|Real_to_Integer
           DEFINE     arg1
           stq        arg1
           RESULT     Q,I
           RETURN

assign_RI:
           ERASE      A_reg
           ADD_REFERENCE        arg1
           LOAD       Q,arg2
           tsx0       ap|Integer_to_Real
           DEFINE     arg1
           dfst       arg1
```

```
          RESULT     EAQ,R
          RETURN

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

assign_BB:
          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_II:
          IF_INREG   Q,arg1
          then {     adq        arg2       }
          else {     IF_INREG   Q,arg2
                     then {     adq        arg1       }
                     else {     LOAD       Q,arg1
                                adq        arg2       }}
          RESULT     Q,I
          RETURN

add_IR:
          IF_INREG   Q,arg1
          then {     ERASE      A_reg
                     tsx0       ap|Integer_to_Real
                     dfad       arg2       }
          else {     LOAD       Q,arg1
                     ERASE      A_reg
                     tsx0       ap|Integer_to_R
                     dfad       arg2       }
          RESULT     EAQ,R
          RETURN
```

```
add_RI:
          IF_INREG  Q,arg2
          then {    ERASE     A_reg
                    tsx0      ap|Integer_to_Real
                    dfad      arg1       }
          else {    LOAD      Q,arg2
                    ERASE     A_reg
                    tsx0      ap|Integer_to_Real
                    dfad      arg1       }
          RESULT    EAQ,R
          RETURN

add_RR:
          IF_INREG  EAQ,arg1
          then {    dfad      arg2       }
          else {    IF_INREG  EAQ,arg2
                    then {    dfad       arg1      }
                    else {    LOAD       EAQ,arg1
                              dfad       arg2      }}
          RESULT    EAQ,R
          RETURN

add_AA:
          IF_INREG  Any,arg1
          then {    SETUP     Any,arg2
                    tsx0      ap|add_any_operator }
          else {    IF_INREG  Any,arg2
                    then {    SETUP      Any,arg1
                              tsx0       ap|add_any_operator }
                    else {    LOAD       Any,arg1
                              SETUP      Any,arg2
                              tsx0       ap|add_any_operator }}
          RESULT    Any,Any
          RETURN

sub_II:
          LOAD      Q,arg1
          sbq       arg2
          RESULT    Q,I
          RETURN

sub_IR:
          ERASE     A_reg
          LOAD      Q,arg1
          tsx0      ap|Integer_to_Real
          dfsb      arg2
          RESULT    EAQ,R
```

```
          RETURN

sub_RI:
          IF_INREG   Q,arg2
          then {     ERASE      A_reg
                     tsx0       ap|Integer_to_Real
                     dfsb       arg1
                     fneg       0          }
          else {     LOADC      Q,arg2
                     ERASE      A_reg
                     tsx0       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
          tsx0       ap|subtract_any_operator
          RESULT     Any,Any
          RETURN


divide_II:
          ERASE      A_reg
          LOAD       Q,arg1
          div        arg2
          RESULT     Q,I
          RETURN

divide_IR:
          ERASE      A_reg
          LOAD       Q,arg1
          tsx0       ap|Integer_to_Real
          dfdv       arg2
          RESULT     EAQ,R
```

```
          RETURN

divide_RI:
          ERASE      A_reg
          LOAD       Q,arg2
          tsx0       ap|Integer_to_Real
          dfdi       arg1
          RESULT     EAQ,R
          RETURN

divide_RR:
          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       ap|divide_any_operator
          RESULT     Any,Any
          RETURN

multiply_II:
          ERASE      A_reg
          IF_INREG   Q,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.
          lls        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 that
" the number in the AQ was single precision.
          tnc        2,ic
          tsx0       ap|fixedoverflow_operator
          lrs        36
          RESULT     Q,I
          RETURN
```

```
multiply_IR:
        ERASE       A_reg
        LOAD        Q,arg1
        tsx0        ap|Integer_to_Real
        dfmp        arg2
        RESULT      EAQ,R
        RETURN

multiply_RI:
        ERASE       A_reg
        LOAD        Q,arg2
        tsx0        ap|Integer_to_Real
        dfmp        arg1
        RESULT      EAQ,R
        RETURN

multiply_RR:
        IF_INREG    EAQ,arg1
        then {      dfmp        arg2        }
        else {      IF_INREG    EAQ,arg2
                    then {      dfmp        arg1        }
                    else {      LOAD        EAQ,arg1
                                dfmp        arg2        }}
        RESULT      EAQ,R
        RETURN

multiply_AA:
        IF_INREG    Any,arg1
        then {      SETUP       Any,arg2
                    tsx0        ap|multiply_any_operator        }
        else {      IF_INREG    Any,arg2
                    then {      SETUP       Any,arg1
                                tsx0        ap|multiply_any_operator}
                    else {      LOAD        Any,arg1
                                SETUP       Any,arg2
                                tsx0        ap|multiply_any_operator}}
        RESULT      Any,Any
        RETURN

and_BB:
        IF_INREG    A_reg,arg1
        then {      ana         arg2        }
        else {      IF_INREG    A_reg,arg2
                    then {      ana         arg1        }
                    else {      LOAD        A,ar
                                ana         arg         }}
        RESULT      A_reg,B
```

```
          RETURN

and_AA:
          IF_INREG  Any,arg1
          then {    SETUP     Any,arg2
                    tsx0      ap|and_any_operator }
          else {    IF_INREG  Any,arg2
                    then {    SETUP     Any,arg1
                              tsx0      ap|and_any_operator }
                    else {    LOAD      Any,arg1
                              SETUP     Any,arg2
                              tsx0      ap|and_any_operator }}
          RESULT    Any,B
          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      ap|or_any_operator  }
          else {    IF_INREG  Any,arg2
                    then {    SETUP     Any,arg1
                              tsx0      ap|or_any_operator  }
                    else {    LOAD      Any,arg1
                              SETUP     Any,arg2
                              tsx0      ap|or_any_operator  }}
          RESULT    Any,B
          RETURN

catenate_LiLi:

catenate_symbol_SS:

catenate_symbol_AA:
          NOP
          RETURN

less_than_II:
```

```
            ERASE       A_reg
            LOAD        Q,arg1
            cmpq        arg2
            tsx0        ap|lt_to_bool
            RESULT      A_reg,B
            RETURN

less_than_IR:
            LOAD        Q,arg1
            ERASE       A_reg
            tsx0        ap|Integer_to_Real
            dfcmp       arg2
            tsx0        ap|lt_to_bool
            RESULT      A_reg,B
            RETURN

less_than_RI:
            LOAD        Q,arg2
            ERASE       A_reg
            tsx0        ap|Integer_to_Real
            dfcmp       arg1
            tsx0        ap|gt_to_bool
            RESULT      A_reg,B
            RETURN

less_than_RR:
            IF_INREG    EAQ,arg1
            then {      dfcmp       arg2
                        tsx0        ap|lt_to_bool          }
            else {      IF_INREG    EAQ,arg2
                        then {      dfcmp       arg1
                                    tsx0        ap|gt_to_bool        }
                        else {      LOAD        EAQ,arg1
                                    dfcmp       arg2
                                    tsx0        ap|lt_to_bool        }}
            RESULT      A_reg,B
            RETURN

less_than_AA:
            ERASE       A_reg
            LOAD        Any,arg1
            SETUP       Any,arg2
            tsx0        ap|less_than_any_operator
            RESULT      A_reg,B
            RETURN

greater_than_II:
```

```
          ERASE      A_reg
          LOAD       Q,arg1
          cmpq       arg2
          tsx0       ap|gt_to_bool
          RESULT     A_reg,B
          RETURN

greater_than_IR:
          LOAD       Q,arg1
          ERASE      A_reg
          tsx0       ap|Integer_to_Real
          dfcmp      arg2
          tsx0       ap|gt_to_bool
          RESULT     A_reg,B
          RETURN

greater_than_RI:
          LOAD       Q,arg2
          ERASE      A_reg
          tsx0       ap|Integer_to_Real
          dfcmp      arg1
          tsx0       ap|lt_to_bool
          RESULT     A_reg,B
          RETURN

greater_than_RR:
          IF_INREG   EAQ,arg1
          then {     dfcmp      arg2
                     tsx0       ap|gt_to_bool       }
          else {     IF_INREG   EAQ,arg2
                     then {     dfcmp      arg1
                                tsx0       ap|lt_to_bool       }
                     else {     LOAD       EAQ,arg1
                                dfcmp      arg2
                                tsx0       ap|gt_to_bool       }}
          RESULT     A_reg,B
          RETURN


greater_than_AA:
          ERASE      A_reg
          NOP
          RETURN


less_or_equal_II:
          ERASE      A_reg
```

```
        IF_INREG  Q,arg1
        then {    cmpq      arg2
                  tsx0      ap|lt_or_eq_to_bool }
        else {    IF_INREG  Q,arg2
                  then {    cmpq      arg1
                            tsx0      ap|gt_or_eq_to_bool }
                  else {    LOAD      Q,arg1
                            cmpq      arg2
                            tsx0      ap|lt_or_eq_to_bool }}
        RESULT    A_reg,B
        RETURN

less_or_equal_IR:
        LOAD      Q,arg1
        ERASE     A_reg
        tsx0      ap|Integer_to_Real
        dfcmp     arg2
        tsx0      ap|lt_or_eq_to_bool
        RESULT    A_reg,B
        RETURN

less_or_equal_RI:
        LOAD      Q,arg2
        ERASE     A_reg
        tsx0      ap|Integer_to_Real
        dfcmp     arg1
        tsx0      ap|gt_or_eq_to_bool
        RESULT    A_reg,B
        RETURN

less_or_equal_RR:
        IF_INREG  EAQ,arg1
        then {    dfcmp     arg2
                  tsx0      ap|lt_to_bool         }
        else {    IF_INREG  EAQ,arg2
                  then {    dfcmp     arg1
                            tsx0      ap|gt_to_bool         }
                  else {    LOAD      EAQ,arg1
                            dfcmp     arg2
                            tsx0      ap|lt_to_bool         }}
        RESULT    A_reg,B
        RETURN

less_or_equal_AA:
        ERASE     A_reg
        NOP
        RETURN
```

```
greater_or_equal_II:
        ERASE       A_reg
        IF_INREG    Q,arg1
        then {      cmpq        arg2
                    tsx0        apl$gt_or_eq_to_bool }
        else {      IF_INREG    Q,arg2
                    then {      cmpq        arg1
                                tsx0        apl$lt_or_eq_to_bool }
                    else {      LOAD        Q,arg1
                                cmpq        arg2
                                tsx0        apl$gt_or_eq_to_bool }}
        RESULT      A_reg,B
        RETURN

greater_or_equal_IR:
        LOAD        Q,arg1
        ERASE       A_reg
        tsx0        apl$Integer_to_Real
        dfcmp       arg2
        tsx0        apl$gt_or_eq_to_bool
        RESULT      A_reg,B
        RETURN

greater_or_equal_RI:
        LOAD        Q,arg2
        ERASE       A_reg
        tsx0        apl$Integer_to_Real
        dfcmp       arg1
        tsx0        apl$lt_or_eq_to_bool
        RESULT      A_reg,B
        RETURN

greater_or_equal_RR:
        IF_INREG    EAQ,arg1
        then {      dfcmp       arg2
                    tsx0        apl$gt_or_eq_to_bool }
        else {      IF_INREG    EAQ,arg2
                    then {      dfcmp       arg1
                                tsx0        apl$lt_or_eq_to_bool }
                    else {      LOAD        EAQ,arg1
                                dfcmp       arg2
                                tsx0        apl$gt_or_eq_to_bool }}
        RESULT      A_reg,B
        RETURN

greater_or_equal_AA:
        ERASE       A_reg
```

```
        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        ap|eq_to_bool
        RESULT      A_reg,B
        RETURN

equal_IR:
        ERASE       A_reg
        LOAD        Q,arg1
        tsx0        ap|Integer_to_Real
        dfcmp       arg2
        tsx0        ap|eq_to_bool
        RESULT      A_reg,B
        RETURN

equal_RI:
        ERASE       A_reg
        LOAD        Q,arg2
        tsx0        ap|Integer_to_Real
        dfcmp       arg1
        tsx0        ap|eq_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        ap|eq_to_bool
        RESULT      A_reg,B
        RETURN

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      }}
" 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       ap|eq_to_bool
          RESULT     A_reg,B
          RETURN

equal_PP:
equal_SS:
equal_AA:
          NOP
          RETURN

not_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       ap|ne_to_bool
          RESULT     A_reg,B
          RETURN

not_equal_IR:
          LOAD       Q,arg1
          ERASE      A_reg
          tsx0       ap|Integer_to_Real
          dfcmp      arg2
          tsx0       ap|ne_to_bool
          RESULT     A_reg,B
          RETURN

not_equal_RI:
          LOAD       Q,arg2
          ERASE      A_reg
          tsx0       ap|Integer_to_Real
          dfcmp      arg1
          tsx0       ap|ne_to_bool
          RESULT     A_reg,B
          RETURN
```

```
not_equal_RR:
          IF_INREG   EAQ,arg1
          then {     dfcmp       arg2        }
          else {     IF_INREG    EAQ,arg2
                     then {      dfcmp       arg1        }
                     else {      LOAD        EAQ,arg1
                                 dfcmp       arg2        }}
          tsx0       apine_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       apine_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_II:
exponentiate_IR:
exponentiate_RI:
exponentiate_RR:
exponentiate_AA:
          NOP
          RETURN
```

```
complement_B:
            LOAD        A_reg,arg1
            era         =o400000,du
            RESULT      A_reg,B
            RETURN

complement_A:
            LOAD        A_reg,arg1
            tsx0        ap|complement_any_operator
            RESULT      Any,B
            RETURN

deref_Ra:
            NOP
            RETURN

negate_I:
            LOADC       Q,arg1
            RESULT      Q,I
            RETURN

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

negate_A:
            LOAD        Any,arg1
            tsx0        ap|negate_any_operator
            RESULT      Any,Any
            RETURN

lock_G:
lock_A:
unlock_G:
unlock_A:
test_lock_GN:
test_lock_AN:
            NOP
            RETURN

case_of_CN:
            CASE_USAGE              arg1,arg2
            RETURN
```

```
caselimit_CI:
          ERASE       A
          lda         1,dl
          LOAD        Q,arg1
          cwl         arg2
          tsx0        apleq_to_bool
          RESULT      A_reg,B
          RETURN

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

branch_N:
          NOTE_FLOWCHANGE
          tra         arg1
          RETURN


branch_true_NB:
          NOTE_FLOWCHANGE
          LOAD        A_reg,arg2
          tmi         arg1
          RETURN

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

branch_false_NB:
          NOTE_FLOWCHANGE
          LOAD        A_reg,arg2
          tpl         arg1
          RETURN

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

```
          tpl       arg1
          RETURN

label_N:
          NOTE_FLOWCHANGE
          FILL_USAGE            arg1
          RETURN

procedure_NN:
          GEN_DEF   arg1
entry_location:
          eax7      0            " will be stack size.
          eppbp     sb|40,*      " seal_operators_$operator_table
          tspab     bp|entry_operator
entry_control_word1:
          oct       0            " filled in later. will be
                                 " 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:
          GEN_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      BP,arg1
          ARG_PTR   BP,arg2
          RETURN
```

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

call_PN:
          ERASE      BB
          ERASE      SB
          ERASE      LP
          IF_OPERAND           arg2,is_zero
          then {     eppab     ap|null_arg_list     }
          else {     LOAD      AB,arg2              }
          epplp      sp|seal_frame.linkage_ptr,*
          LOAD       BP,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_AI:
```

```
          TYPE_CHECK           Li,arg1
" fall into select_NI code.

select_NI:
          ERASE     A_reg
          lda       1,dl
          LOAD      Q,arg1
          qrl       18
          cwl       arg2
          tze       2,ic
          tsx0      ap|subscript_error_operator
          LOAD      Q,arg2
          eppbp     arg1,*ql
          RESULT    BP,Re
          RETURN



nop_N:
          nop       0,dl
          RETURN

mode_select_NN:
          MODE_SELECT          arg1,arg2
          RETURN

line_number_C:
          RETURN

addr_N:
          RETURN

encode_dims_NC:

encode_value_NC:

encode_mode_NC:
          NOP
          RETURN

abs_I:
          ERASE     Q
          LOAD      A_reg,arg1
          cmpa      0,dl
          tpl       2,ic
          neg       0
          lrs       36
```

```
          RESULT     Q,I
          RETURN

abs_R:
          LOAD       EAQ,arg1
          cmpa       0,dl
          tpl        2,ic
          fneg       0
          RESULT     EAQ,R
          RETURN

abs_A:
          NOP
          RETURN

atan_I:
atan_R:
atan_A:
          NOP
          RETURN

boolean_I:
          ERASE      A_reg
          LOAD       Q,arg1
          cmpq       0,dl
          tze        2,ic
          ldq        =o400000,du
          llr        36
          RESULT     A_reg,B
          RETURN

boolean_R:
          LOAD       EAQ,arg1
          dfcmp      =0.0e0,du
" Next instruction assumes that a floating point zero
" leaves the A_reg = 0.
          tze        2,ic
          lda        =o400000,du
          RESULT     A_reg,B
          RETURN

boolean_A:

ceil_R:
ceil_I:
ceil_A:
```

```
cos_I:
cos_R:
cos_A:

delete_S:
deletedir_S:
detach_S:

exp_I:
exp_R:
exp_A:

find_S:
find_A:

floor_R:
floor_I:
floor_A:

integer_I:
            LOAD        Q,arg1
            RESULT      Q,I
            RETURN

integer_R:
            LOAD        EAQ,arg1
            tsx0        ap|Real_to_Integer
            RESULT      Q,I
            RETURN

integer_B:
            LOAD        A_reg,arg1
            lrl         71
            RESULT      Q,I
            RETURN

integer_S:
integer_A:

length_Li:
            NOP
            RETURN

log_I:
log_R:
log_A:
```

```
log10_I:
log10_R:
log10_A:
        NOP
        RETURN

real_I:
        ERASE       A_reg
        LOAD        Q,arg1
        tsx0        ap|Integer_to_Real
        RESULT      EAQ,R
        RETURN

real_R:
        LOAD        EAQ,arg1
        RESULT      EAQ,R
        RETURN

real_B:
        ERASE       Q
        LOAD        A_reg,arg1
        lrl         71
        tsx0        ap|Integer_to_Real
        RESULT      EAQ,R
        RETURN

real_S:
real_A:
        NOP
        RETURN

sign_I:
        ERASE       Q
        ldq         0,dl
        szn         arg1
        tze         5,ic                  " =0     0
        tmi         3,ic                  " <0    -1
        ldq         1,dl                  " >0     1
        tra         2,ic
        lcq         1,dl
        RESULT      Q,I
        RETURN

sign_R:
        ERASE       EAQ
        ldq         0,dl
" The next instruction will correctly test a double precision
```

```
" number for being <0, =0, or >0, which is all we need.
" (there is no dfszn instruction).
            fszn      arg1
            tze       5,ic
            tmi       3,ic
            ldq       1,dl
            tra       2,ic
            lcq       1,dl
            RESULT    Q,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_N0:
create_NS:

is_NN:
            NOP
            RETURN
```

```
get_N:
        ERASE       LP
        ERASE       SB
        IF_OPERAND              arg1,is_zero
        then {      eppbp       ap|null_pointer,*    }
        else {      LOAD        BP,arg1              }
        tsx0        ap|get_operator
        RETURN

put_N:
        ERASE       LP
        ERASE       SB
        IF_OPERAND              arg1,is_zero
        then {      eppbp       ap|null_pointer,*    }
        else {      LOAD        BP,arg1              }
        tsx0        ap|put_operator
        RETURN

void_NN:

attach_SS:

createdir_SS:

edit_IS:
edit_RS:
edit_AA:
        NOP
        RETURN

max_II:
        ADD_REFERENCE           arg2
        LOAD        Q,arg1
        cmpq        arg2
        tpl         2,ic
        LOAD        Q,arg2
        RESULT      Q,I
        RETURN

max_IR:
        ADD_REFERENCE           arg2
        LOAD        Q,arg1
        ERASE       A_reg
        tsx0        ap|Integer_to_Real
        dfcmp       arg2
        tpl         2,ic
        LOAD        EAQ,arg2
```

```
        RESULT      EAQ,R
        RETURN

max_RI:
        ADD_REFERENCE           arg1
        ERASE       A_reg
        LOAD        Q,arg2
        tsx0        ap|Integer_to_Real
        afcmp       arg1
        tmi         2,ic
        LOAD        EAQ,arg1
        RESULT      EAQ,R
        RETURN

max_RR:
        ADD_REFERENCE           arg2
        LOAD        EAQ,arg1
        dfcmp       arg2
        tpl         2,ic
        LOAD        EAQ,arg2
        RESULT      EAQ,R
        RETURN

max_AA:
        NOP
        RETURN

min_II:
        ADD_REFERENCE           arg2
        LOAD        Q,arg1
        cmpq        arg2
        tmi         2,ic
        LOAD        Q,arg2
        RESULT      Q,I
        RETURN

min_IR:
        ADD_REFERENCE           arg2
        ERASE       A_reg
        LOAD        Q,arg1
        tsx0        ap|Integer_to_Real
        dfcmp       arg2
        tmi         2,ic
        LOAD        EAQ,arg2
        RESULT      EAQ,R
        RETURN
```

```
min_RI:
          ADD_REFERENCE          arg1
          ERASE       A_reg
          LOAD        Q,arg2
          tsx0        ap|Integer_to_Real
          dfcmp       arg1
          tpl         2,ic
          LOAD        EAQ,arg1
          RESULT      EAQ,R
          RETURN

min_RR:
          ADD_REFERENCE          arg2
          LOAD        EAQ,arg1
          dfcmp       arg2
          tmi         2,ic
          LOAD        EAQ,arg2
          RESULT      EAQ,R
          RETURN

min_AA:
          NOP
          RETURN


mod_II:
          ERASE       A_reg
          LOAD        Q,arg1
          div         arg2
          lrl         36
          RESULT      Q,I
          RETURN

mod_IR:
mod_RI:
mod_RR:
mod_AA:

rename_SS:

round_RC:
round_AC:

          NOP
          RETURN

          end
```

BIBLIOGRAPHY

Boehm, B. W., "Software and Its Impact: A Quantitative Assessment," DATAMATION 19, 5 (May 1973), 49.

Cuff, R. N., "A Conversational Compiler for Full PL/I," British Computer Society Journal 15, 2 (May 1972), 99-104.

Fenichel, R. R., "On Implementation of Label Variables," Communications of the ACM 14, 5 (May 1971), 349-350.

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

Gries, D., Compiler Construction for Digital Computers, John Wiley & Sons, Inc., 1971.

Honeywell Information Systems, Inc., "The Multics PL/I Language," Document AG94, Waltham, Mass., 1972.

Honeywell Information Systems, Inc., "Honeywell Model 6180 Processor Manual," Waltham, Mass., to be published.

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

M.I.T. Project MAC and Honeywell Information Systems, Inc., "Multics Programmers' Manual," 1973.

Schroeder, M. D., and Saltzer, J. H., "A Hardware Architecture for Implementing Protection Rings," Communications of the ACM 15, 3 (March 1972), 157-170.