AD ____________________

TECHNICAL REPORT
GIT–ICS–81/16

A RE–USABLE CODE GENERATOR FOR PRIME 50-SERIES COMPUTERS

By

T. Allen Akin

Prepared for

OFFICE OF NAVAL RESEARCH
800 N. QUINCY STREET
ARLINGTON, VIRGINIA 22217

Under

Contract No. N00014–79–C–0873
GIT Project No. G36–643

August 1981

GEORGIA INSTITUTE OF TECHNOLOGY
A UNIT OF THE UNIVERSITY SYSTEM OF GEORGIA
SCHOOL OF INFORMATION AND COMPUTER SCIENCE
ATLANTA, GEORGIA 30332

1981

THE RESEARCH PROGRAM IN
FULLY DISTRIBUTED PROCESSING SYSTEMS

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER
GIT-ICS-81/16

2. GOVT ACCESSION NO.

3. RECIPIENT'S CATALOG NUMBER

4. TITLE *(and Subtitle)*
A Re-usable Code Generator for
PRIME 50-Series Computers

5. TYPE OF REPORT & PERIOD COVERED
Technical Report
August 1981

6. PERFORMING ORG. REPORT NUMBER
GIT-ICS-81/16

7. AUTHOR*(s)*
T. Allen Akin

8. CONTRACT OR GRANT NUMBER*(s)*
N00014-79-C-0873

9. PERFORMING ORGANIZATION NAME AND ADDRESS
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS

11. CONTROLLING OFFICE NAME AND ADDRESS
Office of Naval Research
800 N. Quincy Street
Arlington, Virginia 22217

12. REPORT DATE
August 1981

13. NUMBER OF PAGES
35 + vi

14. MONITORING AGENCY NAME & ADDRESS*(if different from Controlling Office)*

15. SECURITY CLASS. *(of this report)*
Unclassified

15a. DECLASSIFICATION/DOWNGRADING SCHEDULE
n/a

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES
The view, opinions and/or findings contained in this report are those of the author and should not be construed as official Department of the Navy position, policy, or decision unless so designated by other documentation.

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*
Code generator
Compiler
Semantic analyzer
Intermediate form tree structures

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

A code generator is the portion of a compiler that converts an interval representation of the semantics of a program into equivalent machine code. Construction of a code generator requires a major effort, so it should be done as infrequently as possible. One way to make this possible is to build a code generator that may be re-used from compiler to compiler.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

A RE-USABLE CODE GENERATOR
FOR PRIME 50-SERIES COMPUTERS

TECHNICAL REPORT
GIT-ICS-81/16

T. Allen Akin

August, 1981

Office of Naval Research
800 N. Quincy Street
Arlington, Virginia 22217

Contract No. N00014-79-C-0873
GIT Project No. G36-643

The Georgia Tech Research Program in
Fully Distributed Processing Systems
School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

THE VIEW, OPINIONS, AND/OR FINDINGS CONTAINED IN THIS REPORT ARE THOSE OF THE AUTHORS AND SHOULD NOT BE CONSTRUED AS AN OFFICIAL DEPARTMENT OF THE NAVY POSITION, POLICY, OR DECISION, UNLESS SO DESIGNATED BY OTHER DOCUMENTATION.

Summary

This thesis describes the design and implementation of a language-independent reuseable code generator for Prime 400 and 50-Series computers.

A code generator is the portion of a compiler that converts an internal representation of the semantics of a program into equivalent machine code. Construction of a code generator requires a major effort, so it should be done as infrequently as possible. One way to make this possible is to build a code generator that may be re-used from compiler to compiler.

Several factors influence the design of such a code generator, including the nature of the communications channel between the code generator and the rest of the compiler, the structure of the information passed to the code generator (the "intermediate form"), the form of output code desired, and finally the limitations of the machine architecture and existing systems software.

The code generator implemented processes a high-level, tree-structured intermediate form, performing translation by case analysis and optimization by eliminating redundant load operations. It produces a stream of assembly language source code which may then be assembled, loaded, and executed.

Experience with two compiler implementations has shown that the re-usable code generator approach is feasible. However, several improvements in the present code generator would be desirable.

TABLE OF CONTENTS

CHAPTER 1

Introduction

1.1 Motivation

The School of Information and Computer Science maintains a network of five medium-scale Prime computers for both academic and research activities. Despite the natural growth in "Prime expertise" resulting from five years of use, there has been no successful local compiler implementation. The few attempts that have been made were stymied by various difficulties in dealing with the machine architecture and systems software.

Nevertheless, there are several reasons for undertaking further compiler implementation projects on the School's Prime computers:

- Both ICS and the Prime-using segment of the business community are interested in the C programming language [Kernighan 1978]. A wealth of software exists in the form of C programs; a C compiler on the Primes would thus enhance their usability and effectiveness.

- Existing compilers for Prime computers are large programs that seriously impair system throughput when they are run. For example, the PL/I subset G compiler processes programs at approximately 500 lines per CPU minute, referencing 384K bytes of shared code and 256K bytes of private data space. Three concurrent PL/I compilations cause excessive paging on a Prime 550 with 1.5 megabytes of main memory, pushing system response time to the point of user frustration, even for simple operations like logging in. In an academic environment where compilations are frequent, this is unacceptable. Local replacements for existing compilers could improve system throughput as well as offer useful new features.

- Computer science courses, like those in physics or electrical engineering, use laboratories to provide students with "hands-on" experience. ICS majors need access to large software systems like compilers, but commercial compilers are frequently inaccessable for legal reasons. Locally-developed "training compilers" could meet the need.

- Research projects within the School occasionally require language translators for special applications. For example, researchers in the fields of artificial intelligence and fully distributed data processing have identified needs for programming languages that are not implemented on the Primes.

1.2 Compiler Design

To see how a re-usable code generator can help to meet the School's needs, it is necessary to consider current compiler design practices.

In what has come to be the "classic" scheme, a compiler is composed of four parts:

- The lexical analyzer converts a stream of characters supplied by the user into higher level "tokens." This process is analogous to the way a person groups written letters to form words.
- The syntax analyzer groups the tokens produced by the lexical analyzer into structures, according to the rules of a grammar. In a similar vein, a person groups words to form phrases and sentences.
- The semantic analyzer extracts the "meaning" of the structures produced by the syntax analyzer. In the case of people, sentences are "interpreted."
- The code generator in a sense inverts the preceding processes: it synthesizes a sequence of instructions that meets the lexical and syntactic requirements of a computer's machine language while insuring that the sequence is "semantically equivalent" to the original program. As with a human translator, it is essential that the code generator have an excellent command of the language into which it is translating. Otherwise, there is much less incentive to use a compiler; it might be more economical to produce machine code by hand.

The machine instructions produced by the code generator are interpreted by a computer to perform the task expressed by the original program.

1.3 Pragmatics

Today, lexical and syntax analysis are well-understood; a large body of theoretical results has made it possible to automate the construction of lexical and syntax analyzers (see for example [Aho 1972]). Semantic analysis is more complex and consequently less well-understood. Code generation is in a similar state; the best automatic code generation algorithms are little better than heuristically controlled searches [Graham, 1980].

For economic reasons, it is desirable to minimize the amount of "custom-crafted" single-use code in a given compiler implementation. Compilers are large, complex pieces of software, and writing one is an expensive task. However, if portions of a compiler may be re-used in subsequent implementations, the total amortized cost of the compiler can be reduced.

In large measure, the lexical, syntax, and semantic analysis portions of a compiler for language X can be made independent of the machine on which X is to run. Similarly, the code generation portion of a compiler can be made largely independent of the language being compiled. In theory, one would have a single "front end" for each language to be compiled, and a single "back end" for each target machine. In this way, a compiler for any language would be available for any machine; it would only be necessary to connect the appropriate front end to the proper back end. This scheme maximizes re-usability of compiler implementation code, thus minimizing cost. Unfortunately, it is not possible in practice; there are significant differences between languages and between target machines which make such an approach infeasible. However, it is possible to make substantial progress toward the goal of re-usability with a practical implementation of a back end for one machine.

A re-usable code generator for Prime hardware is desirable for another reason. The Prime architecture (relevant aspects of which will be discussed in more detail below) suffers from a number of shortcomings that make it inhospitable to high-level language compilers. For example, a number of operations are simply missing from the instruction set; it is possible to do 32 bit wide logical "and" operations, but there is no 32 bit logical "or" instruction. The available addressing formats make it necessary to reference separately-compiled objects with indirect addresses, thus requiring knowledge of external objects at compile time as well as forcing the use of different addressing techniques. The method of memory segmentation is closely connected to the implementation of several instructions, causing array indexing to fail when an array overlaps a boundary between memory segments. For these and other reasons, code generation for the Primes is particularly difficult. It seems best to invest the effort in building a code generator just once, making it re-usable for future compiler implementations.

1.4 Problem

The ICS Prime computer systems support a number of "software tools" designed to simplify the processes of lexical and syntactic analysis. Unfortunately, there is no analogous support for code generation.

The central problem of this thesis may be stated as follows: Design and implement a code generator for Prime computers. The code generator must present an interface that may be used by a variety of front-end language and processors. Furthermore, the interface should depend on features of the underlying machine architecture as little as possible.

The code generator should produce instructions that are common to all the machines in the ICS Computing Laboratory Prime complex. The code generator should be "fast," at least in comparison to the code generation phases of compilers that are already available. It should produce machine code programs of high enough quality that there is little temptation to use an existing compiler or to write programs directly in machine code when code efficiency is the major issue.

1.5 Related Efforts

A number of other efforts have influenced the direction of this thesis:

- The author's senior design project at Georgia Tech [Akin 1979] involved, among other things, the construction of a compiler for a microcomputer systems programming language. The need for identical source programs to run on two different microcomputers led to the factoring of the compiler into a machine independent front end and two machine dependent back ends. Experience with the interface between segments of the compiler strongly affected the design of the interface for the present code generator.

- Stephen Johnson's portable C compiler [Johnson 1979] shows a successful approach to code generation that differs from the one taken in this project: "pcc" uses a machine independent code generator with tables that are tailored by end-users for particular machines. However, Johnson's ideas on code optimization were used in the current effort without much change.

- The Charrette Ada Compiler project [Lamb 1980] provided insight into the problems of developing a truly language-independent intermediate form (interface between front end and code generator). In the Charrette project, the front end produced a tree-structured intermediate form known as TCOL-Ada. The success of TCOL-Ada was an important factor in the decision to use a tree-structured intermediate form in this project. (Ada is a trademark of the U. S. Department of Defense.)

CHAPTER 2

Design Considerations

The design of the code generator was largely driven by environmental factors: existing means of communication between front end and code generator, machine code file formats, system software limitations, and constraints imposed by the machine architecture. Fortunately, there were a few degrees of freedom in the design, particularly in the format of the intermediate code used for communication between front ends and the code generator.

2.1 Communications Channel

Given that compilers using the code generator will be composed of a front end (lexical/syntactic/semantic analysis) and a back end (code generation), how should communication between the two components be arranged? There are several points to consider:

- The amount of information passed from front end to back end varies with the size of the source code program, and (as will be seen below) the entire source program must be processed by the front end before code generation can begin. Therefore, no assumptions can be made about limiting the amount of intermediate code; the communications medium must be capable of queueing a large amount of data.

- Both the medium and the encoding of the intermediate information should be independent of source language and target machine.

- The programming methodology described in Software Tools [Kernighan 1976] has been incorporated in the Software Tools Subsystem [Akin 1980] running on the ICS Prime computers. The Subsystem provides significant advantages to users willing to follow certain conventions for inter-program communication.

The requirement for a communications medium of unbounded size clearly indicates the need for some sort of file on mass storage. The Software Tools Subsystem particularly encourages files of textual data (ASCII characters). The requirement for language and machine independence favors the use of character representation of integers, which can be produced by virtually all I/O support systems and which port easily from machine to machine. (This has the advantage that, during debugging, the compiler writer can view or edit the output of his front end without having to code special tools for the purpose; unfortunately, the conversion to and from the textual representation slows down code generation.) Therefore the communications channel should be one or more temporary text files, and the encoding technique will be conversion to character representation of integers.

2.2 Intermediate Form

The intermediate form (IMF) is the language used by the front end to communicate the semantics of a compiled program to the code generator. It may be considered the instruction set of a "virtual" computer, in which case the code generator is best viewed as a translator of virtual machine instructions into actual machine instructions.

The design of the IMF breaks down into three parts: the selection of "operators" (virtual machine instructions), the definition of the primitive data types on which the operators are used, and the selection of a structure in which the operators are imbedded.

2.2.1 Operators

The choice of IMF operators is essentially unconstrained, but a number of relevant observations may be drawn from experience:

- Operators may be low-level (close to actual machine instructions) or high-level (more abstract, closer to typical programming language operations).

- Higher-level operators provide more context information, allowing more straightforward translation to efficient machine instructions. For example, a "range check" operator must be implemented with two compare-and-skip tests on the Prime. There is a clever way of interlacing the two tests which is valuable for range testing but practically useless for combining two tests in the general case. The use of a range check operator in the IMF allows the code generator to produce efficient code for range checking without wasting time trying to optimize the more common general case (two tests in a row).

- High-level IMF operators may simplify code generation algorithms. For example, the presence of an "if-statement" operator might guarantee the code generator that control enters statements in the "else-part" of the "if" from only one point. This would allow tracking of register contents across the basic block boundary at the beginning of the "else-part." Without the "if-statement" operator, it might be necessary to construct a complete program flow graph to get the same information.

- If an IMF operator is conceptually similar to a high-level language construct, that construct is easily translated by simply generating the IMF operator.

2.2.2 Data Types

IMF operators express data manipulations and abstractions of control flow. Additional information is required to describe the data that is to be manipulated. The situation is complicated by inherently machine-dependent data definitions that are available in languages like C and Ada.

In the present work, this issue was addressed by parameterizing the types of data handled by the IMF "virtual machine." This allows machine dependencies in data description to be restricted to fairly small parts of the front end.

2.2.3 Structure

[Gries 1971] discusses a variety of structures for intermediate forms: triples, indirect triples, quadruples, Polish notation, etc. The tree structure selected for this project has a number of advantages:

- Trees are easily generated during top-down or bottom-up parses.

- When expressions are represented as trees, there is no need for the front end to handle allocation of temporary variables.

- Trees are easily linearized by converting them to Polish notation. Thus, they meet the requirements of the sequential communications channel between the front end and the code generator.

- Tree formats are flexible; for instance, operators with varying numbers of operands are easily accommodated.
- Many algorithms related to code generation are expressed in terms of operations on graphs; constant folding, operand reordering, common subexpression elimination, and global register tracking are examples. These algorithms may often be applied to the tree-structured IMF directly.

2.2.4 Results

The intermediate form devised for this project is tree-structured, with about 70 operators and seven primitive data types (see Appendix A). It defines an expression-oriented virtual machine language with sufficient power to support programming languages on the level of C or Pascal. For a simple example of the intermediate form, see Appendix B; for a tutorial and a complete set of examples, see [Akin 1981].

2.3 Output

There are two alternative formats of code generator output: object code and assembly language source code.

Object code is compact, and comparatively quick to produce on most machines. On the Primes, unfortunately, object code formats are extremely complex. Furthermore, Prime has scheduled changes to its object code formats in the near future, so use of the current formats would guarantee quick obsolescence of the code generator.

Assembly language source code is bulky and therefore incurs extra overhead in production. In addition, the assembler must be invoked to produce the final object file. Prime's assembler uses a three-pass algorithm; the first pass essentially does nothing but recoup information that was available to the code generator but was, of necessity, lost in the translation to assembly source. Thus, the time required for this first pass is simply wasted. However, use of the assembler insulates the code generator from details of the object code format.

In the final analysis, there was no choice: the planned changes to Prime's object code formats make assembly language output the only viable option. Accordingly, the code generator was designed to produce source code for a final pass by the assembler.

2.4 System Software Limitations

A few code generator features were mandated by the limitations of Prime's system software.

For example, the names of all entry points (typically procedure names) must be listed at the beginning of the object code module in which they are defined. This implies that the names of all procedures must be known before any code can be generated in a given module. One approach, ruled out because of its slowness, would be to scan the entire stream of IMF, picking out procedure names and generating entry point declarations for them. The approach actually taken requires the front end to generate *another* stream of IMF containing procedure names as it makes the first pass over the source program. The code generator reads this stream first, produces the list of entry points, then reads the main stream and generates code.

A major problem with maximum program size follows from the decision to use Prime's assembler. Despite the 256 megawords of address space available to each user, the assembler cannot handle a module any larger than 65,536 words. Although this is a disadvantage from the user's point of view, it simplifies the code generator since 16 bit arithmetic is sufficient to calculate any address within a module.

2.5 Machine Architecture

The Prime P400/P550 architecture ([Prime 1979]) imposed a number of constraints on code generator functionality.

Each Prime computer supports a number of "addressing modes" (actually different instruction sets) in order to maintain compatibility with earlier Prime product lines. The code generator's target addressing mode is "64V mode," the only addressing mode common to all Prime machines in the ICS laboratory that is capable of addressing more than 64K words of data. 64V mode provides an accumulator-based instruction set and segmented virtual memory.

In 64V mode, only the memory segments referenced by three base registers are readily accessable. The "stack frame" referenced by register

SB contains the activation record for the last procedure invoked, the "link frame" referenced by register LB contains static and global variables and non-reentrant code, and the "procedure frame" referenced by register PB contains the reentrant code of the currently-executing procedure. Neither the stack frame nor the link frame can occupy more than one memory segment, so it is safe for the code generator to use hardware indexing, which fails on multi-segment data structures.

The accumulator-based architecture of the 64V mode instruction set has an advantage: there is only one register for each of the primitive data types (integer, floating point, etc.), so a very simple register management algorithm suffices. Unfortunately, some of the registers physically overlap, so problems do arise occasionally.

Most 64V mode instructions cannot directly address all locations in memory. Typically, instructions are restricted to a small local address range and must use indirect addressing to reference memory outside that range. Since the addresses of external objects cannot be known at compile time, it must be assumed that they will lie out of the local address range and thus must be addressed indirectly. It is frequently the case (e.g., with procedure calls) that an object must be referenced before it is known to be internal or external, leaving the code generator with a difficult decision: should direct addressing be used? There are three possible approaches: (1) always use indirect addressing, thereby adding considerable unnecessary overhead to internal object references; (2) scan the entire IMF stream and determine which objects are external, then generate indirect addresses for those objects only; (3) require the front end to supply another stream of IMF listing externally defined objects. As in the case of the entry points described above, the most viable solution is to require the additional stream of IMF.

CHAPTER 3

Implementation Overview

The implementation of the code generator proceeded in four steps: identification and selection of IMF operators, hand generation of machine code sequences for those operators, development of case-analysis algorithms to select proper code sequences for generation, and development of simple optimization algorithms.

3.1 IMF Operator Selection

The final set of operators is semantically very close to the operations of the language C [Kernighan 1978], since one of the initial purposes of the code generator was to support a C compiler. Several operators were added to support Pascal-like operations, particularly range checking, and to provide "escape hatches" for calls to run-time support routines. All operators were subjected to examination based on the criteria discussed in Chapter II before selection. The complete set of operators is listed in Appendix A.

3.2 Code Sequence Selection

According to the hypothesis advanced in Chapter II, the use of high-level IMF operators should contribute to the quality of the output code, since the additional information supplied by the operators contributes to selection of special cases. The next implementation step was thus to develop, by hand, the sequences of machine code that should be generated for each operator on each type of operand in each context in which the operator could legally appear.

The selection of code sequences began with hand-coding by the author and continued with several iterations of examination and improvement by the author and two expert assembly language programmers. Several criteria guided the code sequence selection process:

- Execution time should be minimized. This usually involved careiul study of the 64V mode instruction timings, searching for faster alternative code sequences.
- Code size should be minimized. Where two alternatives were equally

fast or not readily compared in speed, the smaller sequence was preferred.

- Memory references should be minimized. In practice it has often been found that a code sequence that is theoretically faster turns out to be inferior because it involves two single-word fetches, rather than an interleaved double-word fetch.

- Since the Primes buffer access to main memory with a high-speed cache, the number of memory references that can be satisfied from cache storage should be maximized. As in the case above, unexpected irregularities in execution times arise because of the pattern of accesses to the cache memory. If at all possible, references to a single memory location should be placed temporally close together, to maximize the likelihood of finding the contents of that location in the cache.

- Generation of "overhead code" (like loading the auxiliary base register to access some location in memory) should be avoided as long as possible. In many cases the extra instructions are subsumed by the addressing modes used in subsequent instructions. Unfortunately, this guideline fails in certain cases, particularly when code motion optimizations might remove the loading of the auxiliary base register from a loop.

- Code sequences should be matched to their most common usages. For example, most alternatives in multiway branches are selected by case label values that form a small, dense set of integers. With some effort, these branches can usually be implemented with a few "computed go-to" instructions, which are considerably faster than a sequence of tests and branches.

IMF operators may appear in a number of different contexts, which strongly affect the code sequence that must be generated for them. The code generator recognizes five such contexts internally:

- Reach. In this case, the operator is being used to return the address of an object in memory, if possible, and a value in a register otherwise. This context is of particular use in evaluating the left-hand-sides of assignments and the operands of arithmetic operators. Generally, use of the "reach" context implies the generation of a "memory reference" instruction in the final code sequence.

- Load. In this context, the operator is being used to return a value in a register. This is the usual context for obtaining the result of an arithmetic operator.
- Flow. An operator in "flow" context yields a change in the flow-of-control, rather than a value in a register. This is the context in which loop termination expressions are evaluated, for example.
- Void. Voided operators yield side effects only. This is the context in which most programming language statements are evaluated; for instance, an assignment statement has only the side effects of evaluating both the left and right hand sides, then copying the value of the right into the object on the left. In "load" context, the same assignment would also yield the value of the right hand side.
- Argument Pointer (often abbreviated "AP"). AP is the context in which actual parameters of procedures are evaluated. In such a context, all operators must yield the address of an object in memory, even if it is necessary to allocate a section of memory and copy the result of the operator into it.

Within a given context, an operator will be translated according to one or more "cases", usually depending on the accessability of its operands. For example, the "subtract" operator has three cases, depending on whether its right operand may be addressed directly, its left operand may be addressed directly, or neither operand may be addressed directly (e.g., both are expressions yielding values in registers).

Finally, within a given case in a given context, final instruction sequences may be devised for each type of data that a giveh operator may use. Separate registers and different instructions are required for the manipulation of, say, integer and floating point data.

As an example, the final list of code sequences for "load" context amounted to 6000 lines of text. On the average, there were three cases for each operator, and five subcases for each applicable data type. The majority of subcases for the arithmetic operators were quite similar to one another in form.

Code sequences for the other contexts were derived directly from the "load" sequences. In virtually all cases, it was necessary only to delete some code from the "load" sequence or append an instruction or two to satisfy the requirements for other contexts.

3.3 Algorithms for Code Generation

The overall control routine for the code generator is simple:

```
for each input module
   for each entry point
      output an entry point declaration
   for each static data declaration
      reserve link-frame space for a pointer
      output an "indirect pointer" to the external object
   for each static data definition
      reserve link-frame space for the object
      initialize the object's value
   for each procedure
      reconstruct the IMF tree
      walk the tree, transforming IMF to machine code
      optimize the machine code
      convert machine code to assembly language source
```

The heart of the code generator is the procedure handling algorithm. In the next few sections, it will be examined in detail.

3.3.1 Tree Reconstruction

The result of syntactic and semantic analysis by the front end is an intermediate form tree for each procedure in the source program. In order to transmit a tree to the code generator, the front end traverses the tree, directly writing the values of IMF operator parameters and recursively writing the contents of subtrees. The result is a copy of the IMF tree expressed in prefix Polish notation, which is passed through the communications channel to the code generator.

Within the code generator, the tree-building routines have access to a table containing descriptions of each IMF operator: its size, the number and types of its operands, etc. The first step is to read an integer from the input stream; this gives the IMF operator that appears next in the input. The descriptor table is then accessed using the operator number as a key. The table entry gives the size of the tree node, which is then allocated in tree memory. The remainder of the table entry describes the operator's parameters, usually strings, integers, or subtrees. Successive portions of the table entry are interpreted, causing one read from the input stream for each integer parameter, several reads (one for each character) for each string parameter, and a recursive call on the tree builder for each subtree parameter. Values returned for each parameter are placed in the previously-allocated node, and then the node is returned. The final result is a duplicate of the procedure tree built by the front end.

3.3.2 IMF Transformation

The code sequence to be emitted for an operator depends on the context in which it appears, the accessability of its operands, and the type of data being manipulated. Context information is available at the root of the tree and spreads down to the leaves. Operand accessability is known at the leaves and induced from the leaves to the root. Data type information is supplied by the front end for every operator, so it is immediately available.

Postorder tree traversal is an efficient algorithm for propagating accessability information from the leaves up, and preorder tree traversal is efficient for propagating context information down from the root. A simple combination of the two forms the framework for procedure code generation.

Internally, generation of code for an IMF subtree is accomplished by calling one of the routines "reach," "load," "flow," or "void" (ap context is handled as a special case within "load"). This causes the root of the subtree to be visited. Depending on the inherited context and the operator at the root of the tree, one of the generation routines is recursively called for each operand of the root.

Each of the code generation routines returns information about the subtree it just transformed: the location of the result (in memory or in a register), what registers were used to obtain the result, and a linked list of the machine instructions generated to calculate the result at execution time. All of this information comprises the operand accessability data needed for code sequence selection.

Once the operands have been evaluated, code for the root is generated. The code sequences for the operands are then linked with the code sequence for the root. Finally, a tally is made of the registers used and the entire collection of information is returned.

Appendix B illustrates the code generation process for a small subtree.

3.3.3 Optimization

Register tracking forms the basis of the currently implemented optimizer. Within the code generator, each register is associated with a state variable that indicates whether the register's contents are "known" or "unknown." If the contents are known, the register is also associated with an "address descriptor" that pinpoints the memory locations that supplied the register's contents.

The optimization process is a pass over the linked list of procedure code. Whenever possible, general purpose instructions are replaced with faster special-purpose instructions; for example, "LDA =0" (load register A with the value zero) is replaced with "CRA" (clear register A). After replacement, the effects of the instructions on register contents are simulated. Whenever a "load register" instruction is encountered, and the contents of the register to be loaded will not be changed by the instruction, the instruction is eliminated. If a register must be loaded with a new value, and that value is known to reside in another register, the load instruction is replaced with a register-to-register transfer.

The most difficult part of the optimization process is the instruction simulation. In general, a "load" instruction causes a register's state to become known and its contents equal to those of a particular memory location. Most instructions (e.g. arithmetic operations) cause one or more registers' contents to become unknown. "Store" instructions may alter arbitrary locations in memory, thus invalidating a register/memory equivalence; the exact effects are dependent on the particular "store" instruction used. Memory "aliasing" is a particularly nasty problem; the optimizer takes a highly conservative approach and after any store destroys equivalences based on indirect or indexed addresses.

3.4 Implementation

The code generator is written in Ratfor, a Fortran preprocessor language described in [Kernighan 1976]. It is approximately 12,000 lines in length, of which nearly 7,700 (64%) are devoted to the selection of code sequences and 700 (6%) to optimization. The remainder is devoted primarily to input/output, storage management, and simulation of heterogeneous data structures with Fortran arrays.

Although large, the code generator is relatively easy to manage, since most of the Ratfor code deals with independent case analyses. This is reflected in the subprogram call tree, reproduced in part below:

```
module
   initialize
   generate_entries
   generate_static_stuff
   generate_procedures
      reach
         reach_assign
         ...   (approximately 6 other routines)
         reach_seq
         load
      load
         load_addaa
            load, reach
         ...
         load_if
            flow, load, reach
         ...   (approximately 60 other routines)
         load_xor
            load, reach
      flow
         flow_eq
            load, reach
         ...   (approximately 12 other routines)
         flow_switch
            load, reach, void
         load
      void
         void_addaa
```

```
            load, reach
         ...   (approximately 4 other routines)
         void_seq
            load, reach
         load
      optimize
      put_instr
```

The call tree is broad, but not particularly deep. The bulk of the code is in the descendants of "load," and these routines rarely interact with one another.

On the average, a "load" routine is about 77 lines of code. Although this is larger than optimal, most "load" routines are easily comprehended, since they are straightforward case analyses. No attempt was made to eliminate duplicated code. A sample code generation routine is presented in Appendix D.

CHAPTER 4

Experience

The code generator has been used to implement two compilers: a full-scale compiler for the language C and a demonstration compiler for a small teaching language. The C compiler runs almost twice as fast as Prime's Fortran 77 and Pascal compilers (700 lines per minute vs. 400 lines/minute, on a Prime 550). It is also somewhat smaller (in terms of code size) than Fortran 77 or Pascal (2 segments vs. 4 and 3 segments, respectively). Hand inspections and informal benchmarks indicate that the code produced is generally superior to that produced by Pascal, PL/I, and Fortran 77; in particular, fewer base register loads are generated, and operations on packed data structures are performed without resorting to the field manipulation instructions.

Examination of the code generator's output indicates a few areas that need improvement, though. The most obvious is register tracking across basic block boundaries, particularly in loops. Truly excellent code can be produced whenever an arithmetic loop control variable can be pushed into an index register, but present optimization forces stores and loads at the boundaries of the basic block containing the loop body.

The intermediate form could stand a few modifications. For example, there is no way to specify that an array is 65,536 words long. This is no great problem at the moment, but should be fixed in the future. As another example, comparison of structure or array operands requires information on the length of the operands. There is presently no space reserved in the IMF comparison operators for this information.

Several informal measurements of code generator performance have been made. Initially, almost 50% of code generation execution time was devoted to reading the ASCII textual input. Special-casing the character-to-binary conversion and eliminating some logical redundancy within the input routine reduced execution time by 30%. Presumably, elimination of the character-to-binary conversion would speed up execution even more.

CHAPTER 5

Conclusions

The separation of lexical/syntactic/semantic analysis from code generation and the development of a standard "intermediate form" allows many compilers to use the same code generator.

Use of the code generator significantly reduces the amount of effort required to implement compilers on Prime computers.

The case-analysis approach to code generation is effective, at least when compared to the algorithms used in existing compilers for Prime computers. As a consequence, however, the code generator is a very large piece of software, with a great number of almost-identical runs of code.

The code generator's effectiveness has been demonstrated for a C front end, but it seems likely some additions must be made to make it equally effective for Pascal and other languages.

CHAPTER 6

Recommendations

There are several areas in which the code generator might be improved.

Many of the special cases that are currently handled by open code essentially involve emitting special instructions when an operand has a particular value. Clearly these could be encoded in a table, with consequent reduction in code generator size and complexity (although possibly increasing run time, as well).

Data packing is not treated properly in the current implementation. The only operator that explicitly deals with packed data is the FIELD operator, and it must be inserted in the proper places by the front end. The definition of FIELD is machine-dependent in the extreme. A better approach would be to generalize the concept of "address descriptor" used throughout the code generator, allowing any operator to take packed operands directly. A few IMF operators (INDEX and SELECT, especially) would need to be extended to take full advantage of the added generality. There are several special cases (for instance comparison of fields to constants) which should be exploited.

In the intermediate form, data types are restricted to integer, unsigned, long integer, long unsigned, single precision floating, double precision floating, and stowed (structures and arrays). Machine independence could be improved by using precision and range specifications like those available in Ada.

The code generator is written in Ratfor, a FORTRAN preprocessor language. This has the advantage of considerable support from the Software Tools Subsystem running on the ICS Prime computers, and it makes use of the fast FORTRAN 66 compiler. However, the lack of pointers and heterogeneous data structures in FORTRAN makes the code slower and more obtuse than it needs to be. If possible, the code generator should be re-written in a more reasonable language. C would be a good candidate; Pascal would also be a good choice if a better compiler implementation becomes available.

One of the important features of block structured languages that the code generator does not directly support is that of nested scopes. This feature requires a "display" of pointers to currently-active stack frames. Although the display can be conveniently fabricated with existing IMF operators, the present storage allocation algorithm and forward reference resolution techniques are inadequate. For example, to process a procedure B nested in a procedure A, the offsets of A's variables in its stack frame must be known. This will not be the case unless code for A has been generated. Unfortunately, this cannot be done unless the code for B has been queued somewhere, since the code for B precedes the code for A. Thus there is a circular chain of dependencies. One possible solution would be for the front end to allocate space for A's variables, then provide the offsets for use by B and let the code generator handle relocation when it actually generates code for A. This can be done at present by treating all local variables as members of structures, but this imposes an unacceptable amount of machine-dependence on the front end.

The present optimization algorithm is not adequate. Global propagation of register state information would be very valuable, particularly in arithmetic loops. The register tracking scheme now equivalences a register and one location in memory; a better approach would be to build "equivalence classes" containing all registers and memory locations known to have the same value, providing more opportunities to eliminate load instructions and perhaps providing enough information to hoist code from loops.

The 32I mode architecture available on the Prime 550 and higher-numbered models is a multi-register architecture differing somewhat from 64V mode. It would be interesting to see if the ideas used in this code generator could be applied to a 32I mode code generator, or if the same intermediate form would be useful.

The arrival of a VAX 11/780 at ICS within a year poses a similar question, since the VAX is a general register machine. Could a code generator be devised for the VAX, allowing cross-compilation from VAX to Prime and vice versa? Would the intermediate form prove portable enough to permit retargeting and transport of compilers? Since the VAX and the Prime 550 are both virtual memory machines with a natural word width of 16 bits, and there are very few other explicit machine dependencies in the intermediate form, it seems likely that an attempt to implement a retargetable compiler would be successful.

APPENDIX A

Intermediate Form

The following list enumerates the primitive data types supported by the intermediate form. For a more complete description, see [Akin 1981].

INT_MODE	16-bit signed integer
LONG_INT_MODE	32-bit signed integer
UNS_MODE	16-bit unsigned integer
LONG_UNS_MODE	32-bit unsigned integer
FLOAT_MODE	32-bit floating point
LONG_FLOAT_MODE	64-bit floating point
STOWED_MODE	structure or array data

The following list completely enumerates the intermediate form operators. For complete descriptions, see [Akin 1981].

ADDAA_OP	add, assign result to left operand
ADD_OP	add
ANDAA_OP	logical and, assign result to left
AND_OP	logical and
ASSIGN_OP	copy value
BREAK_OP	break out of a loop or case
CASE_OP	case alternative in a switch
COMPL_OP	one's-complement
CONST_OP	define constant
CONVERT_OP	convert data modes
DECLARE_STAT_OP	declare an external static object
DEFAULT_OP	default alternative in a switch
DEFINE_DYNM_OP	define a dynamic local object
DEFINE_STAT_OP	define a static local or global object
DEREF_OP	dereference a pointer
DIVAA_OP	divide, assign result to left operand
DIV_OP	divide
DO_LOOP_OP	test-at-the-bottom loop
EQ_OP	test for equality
FOR_LOOP_OP	generalized loop
GE_OP	test for greater-or-equal

GOTO_OP	jump to label
GT_OP	test for greater-than
IF_OP	conditional statement/expression
INDEX_OP	select an array element
INITIALIZER_OP	initial value of an object
LABEL_OP	target of a jump
LE_OP	test for less-than-or-equal-to
LSHIFTAA_OP	shift left, assign result to left
LSHIFT_OP	shift left
LT_OP	test for less-than
MODULE_OP	beginning of input module
MULAA_OP	multiply, assign result to left
MUL_OP	multiply
NEG_OP	two's-complement
NEXT_OP	force next loop iteration
NE_OP	test for inequality
NOT_OP	logical negation
NULL_OP	null
OBJECT_OP	reference a variable
ORAA_OP	logical or, assign result to left
OR_OP	logical or
POSTDEC_OP	C postdecrement
POSTINC_OP	C postincrement
PREDEC_OP	C predecrement
PREINC_OP	C preincrement
PROC_CALL_ARG_OP	procedure call argument
PROC_CALL_OP	procedure call
PROC_DEFN_ARG_OP	procedure formal parameter
PROC_DEFN_OP	procedure definition
REFTO_OP	generate reference to object
REMAA_OP	remainder, assign result to left
REM_OP	remainder
RETURN_OP	return from procedure
RSHIFTAA_OP	right shift, assign to left operand
RSHIFT_OP	right shift

SAND_OP	sequential (short-circuit) and
SELECT_OP	select field of a structure
SEQ_OP	left-to-right sequence
SOR_OP	sequential (short-circuit) or
SUBAA_OP	subtract, assign result to left
SUB_OP	subtract
SWITCH_OP	multiway branch
UNDEFINE_DYNM_OP	undefine local dynamic object
WHILE_LOOP_OP	test-at-the-top loop
XORAA_OP	exclusive-or, assign to left operand
XOR_OP	exclusive-or
ZERO_INITIALIZER_OP	initialize object to zero
FIELD_OP	extract bit field from a word
CHECK_RANGE_OP	check within range
CHECK_UPPER_OP	check less than upper bound
CHECK_LOWER_OP	check greater than lower bound

APPENDIX B

IMF Transformation

The following tables are excerpted from the case analyses of the IMF operators ADD_OP and SUB_OP. To generate code, the cases are examined left-to-right. The phrase "A not in right regs" may be translated into English as "Register A is not used during the evaluation of the right operand." Note that these operators are members of the same operator class (reversible dyadic one-register) and have very similar code sequences.

REACH CONTEXT/ADD_OP/INTEGER

A not in right regs	A not in left regs	A in both regs
load left	load right	load right
reach right	reach left	allocate temp
ADD right	ADD left	STA temp
		load left
		ADD temp
		deallocate temp

REACH CONTEXT/SUB_OP/INTEGER

A not in right regs	A not in left regs	A in both regs
load left	load right	load right
reach right	reach left	allocate temp
SUB right	SUB left	STA temp
	TCA	load left
		SUB temp
		deallocate temp

Consider the generation of code for the following program fragment:

```
integer a, b, c;
begin
   ...a - (b + c)...
end
```

The following intermediate form code would be generated by the front end:

```
62      SUB_OP
1          INT_MODE
40         OBJECT_OP
1             INT_MODE
...           object id for 'a'
2          ADD_OP
1             INT_MODE
```

```
40          OBJECT_OP
1              INT_MODE
...            object id for 'b'
40          OBJECT_OP
1              INT_MODE
...            object id for 'c'
```

The code generation process for this subtree might be traced as follows:

Control enters through "reach" at the subtree rooted with SUB_OP. Following the definition of SUB_OP that is available internally, "reach" invokes itself recursively to evaluate the left operand.

The left operand is a simple object, which is reached without difficulty. "Reach" returns an address descriptor for the object (say, "SB%+20"), a null set of registers (none were used), and a null list of code (none was generated).

"Reach" invokes itself recursively to evaluate the right operand of the SUB_OP.

The right operand is an ADD_OP. "Reach" invokes itself recursively to evaluate the left operand of the ADD.

The left operand is a simple object. "Reach" returns an address descriptor (say, "SB%+30"), a null set of registers, and a null code list.

"Reach" invokes itself recursively to evaluate the right operand of the ADD.

The operand is a simple object. "Reach" returns an address descriptor (say "SB%+40"), a null set of registers, and a null code list.

Control returns to the instantiation of "reach" at the ADD_OP. The case analysis for ADD is consulted; the first case applies. "Reach" returns the result "in register," a set of registers containing register A, and the code list "LDA SB%+30; ADD SB%+40."

Control returns to the instantiation of "reach" at the SUB_OP. The case analysis for SUB is consulted; since the right operand used the register A, the second case applies. "Reach" returns the result "in register," a set of registers containing only register A, and the code list "LDA SB%+30; ADD SB%+40; SUB SB%+20; TCA."

At this point, good code for the entire subtree has been generated. It may stand alone or be used by some other code tree of which this subtree was a part.

APPENDIX C

User-Oriented Documentation

[Akin 1981] is a compendium of information pertaining to the use of the code generator, rather than its internal structure. Its size (121 pp. single-spaced) precludes its inclusion here. The following paragraphs describing the contents of the User's Guide are excerpted from it.

> The first chapter of this Guide is the Overview. The Overview is a brief summary of the design and construction of the code generator. This chapter may be of general interest, but it is not necessary to read it in order to learn to use the code generator.
>
> The Code Generator Usage chapter describes the location of the code generator and its associated run-time support libraries, as well as the Software Tools Subsystem commands necessary to access them. Recommended procedure is to study this section, then generate command language programs to do the low-level file access operations.
>
> Input Data Stream Formats gives a bird's-eye view of the formats of the three code generator input streams. This chapter merits some study, although it is supplemented by the Extended Examples.
>
> The three operator definitions chapters (Operators Useful in the Static Data Stream, Operators Useful in the Procedure Definition Stream, Operators Useful in Procedure Definitions) provide a detailed reference for the intermediate form operators interpreted by the code generator. One or two readings through this chapter are desirable; thereafter, it can be used as a reference with the Operator/Function Index and the Table of Contents used as entry points.
>
> The Extended Examples are comprised of several short (but complete) programs written in the language C. These examples include the original C code, annotated versions of the three code generator input streams, and an annotated listing of the code generator's assembly language output. The chapter should be useful in learning how the various intermediate form operators work together, and may be used as a reference when building a new front end.
>
> 'Drift' is a very small expression-based language whose structure closely mimics the code generator's internal world-model. The 'Drift' Compiler is a complete, working compiler using the code generator as a back-end. It serves as an example of one way to construct a front-end for the VCG.
>
> For ease of reference, all the intermediate form operators have been organized by subject in the Intermediate Form Operator/Function Index. Typically, one would look up

some function (e.g., "subscripting") in the Index, find the name of the appropriate intermediate form operator (e.g., INDEX_OP), then look up that operator in the table of contents to find its complete description.

APPENDIX D

Code Generation Routine

The following subprogram generates code for addition of two values in a "load" context. It is typical of the descendants of the code generator routine "load."

```
# load_add --- load value of sum of two subexpressions

   ipointer function load_add (expr, regs)
   tpointer expr
   regset regs

   include VCG_COMMON       # global variables

   logical safe

   regset lregs, rregs, opreg

   ipointer l, r
   ipointer seq, ld, st, gen_mr, reach

   integer lres, rres, lad (ADDR_DESC_SIZE),
      rad (ADDR_DESC_SIZE), opsize, opins,
      tad (ADDR_DESC_SIZE)

   select (Tmem (expr + 1))   # data type
      when (INT_MODE, UNS_MODE) {
         opreg = A_REG
         opsize = 1
         opins = ADD_INS
         }
      when (LONG_INT_MODE, LONG_UNS_MODE) {
         opreg = L_REG
         opsize = 2
         opins = ADL_INS
         }
      when (FLOAT_MODE) {
         opreg = F_REG
         opsize = 2
         opins = FAD_INS
         }
      when (LONG_FLOAT_MODE) {
         opreg = LF_REG
         opsize = 4
         opins = DFAD_INS
         }
   else
      call panic ("ADD_OP has bad data mode (*i)*n"p,
         Tmem (expr + 1))
```

```
l = reach (Tmem (expr + 2), lregs, lres, lad)
r = reach (Tmem (expr + 3), rregs, rres, rad)

select
   when (safe (opreg, rregs)) # right doesn't use opreg
      load_add = seq (l,
         ld (opreg, lres, lad),
         r,
         gen_mr (opins, rad))
   when (safe (opreg, lregs)) # left doesn't use opreg
      load_add = seq (r,
         ld (opreg, rres, rad),
         l,
         gen_mr (opins, lad))
else {                        # both sides use opreg
   load_add = seq (r, ld (opreg, rres, rad))
   call alloc_temp (opsize, tad)
   load_add = seq (load_add,
      st (opreg, tad),
      l,
      ld (opreg, lres, lad),
      gen_mr (opins, tad))
   call free_temp (tad)
   }

regs = or (opreg, or (lregs, rregs))
return
end
```

Acknowledgements

I would like to thank Rich LeBlanc for numerous contributions to the design of the re-usable code generator, as well as specific improvements in the current presentation. His patience and advice during the entire period of my graduate study is much appreciated.

Jon Livesey and Philip Enslow, Jr. made several observations that improved the integrity of the thesis significantly.

I am grateful to Perry Flinn, Gene Spafford, and Dan Forsyth for sharing their knowledge of the inner workings of the Prime architecture and systems software.

Finally, I would like to acknowledge the influence of Brian Kernighan and Bill Plauger, whose philosophies of programming first led me to the construction of software tools.

Bibliography

Aho 1972

Aho, Alfred V., and Jeffrey D. Ullman, *The Theory of Parsing, Translation, and Compiling* Vols. I and II, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1972

Akin 1979

Akin, T. Allen, *MUX: A Small Computer Operating System*, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1979

Akin 1980

Akin, T. Allen, Perry B. Flinn, and Daniel H. Forsyth, Jr., *Software Tools Subsystem User's Guide*, 2nd ed., GIT-ICS-80/03, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1980

Akin 1981

Akin, T. Allen, *A Re-Usable Code Generator for Prime 50-Series Computers: User's Guide*, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1981

Forsyth 1977

Forsyth, Daniel H., Jr., *Senior Design Project Final Report*, School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA, 1977

Graham 1980

Graham, Susan L., "Table-Driven Code Generation", IEEE *Computer*, V 13 No 8, p 25, August, 1980

Gries 1971

Gries, David, *Compiler Construction for Digital Computers*, Wiley, New York, NY, 1971

Johnson 1979

Johnson, S. C., "A Tour Through the Portable C Compiler", *Unix Programmer's Manual*, 7th ed., Bell Laboratories, Murray Hill, NJ 1979

Kernighan 1976

Kernighan, Brian W. and P. J. Plauger, *Software Tools*, Addison-Wesley, Reading, MA 1976

Kernighan 1978

Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1978

Lamb 1980

Lamb, David Alex, Andy Hisgen, Jonathan Rosenberg, and Mark Sherman, *The Charrette Ada Compiler*, Computer Science Department, Carnegie-Mellon University, 1980

Prime 1979

Prime Computer, Inc., *PMA Programmer's Guide*, FDR 3059, 1979

Wulf 1981

Wulf, William A., "Compilers and Computer Architecture", IEEE *Computer*, V 14 No 7, p 41, July, 1981