CODE GENERATION AND OPTIMIZATION USING A MACHINE DESCRIPTION TABLE AND ATTRIBUTES

Raman Tenneti
M.E., I.I.Sciences, Bangalore, India, 1979.

A thesis submitted to the faculty
of Oregon Graduate Center
in partial fulfillment of the
requirements for the degree of
Master of Science
in
Computer Science & Engineering

October, 1986

The thesis "Code generartion and optimization using a machine description table and attributes" by Raman Tenneti has been examined and approved by the following Examination Committee:

Richard B. Kieburtz, Thesis Research Advisor
Professor and Chairman,
Department of Computer Science and Engineering

Richard Hamlet
Professor,
Department of Computer Science and Engineering

Dan Hammerstrom
Associate Professor,
Department of Computer Science and Engineering

Mayer D. Schwartz
Adjunct Associate Professor,
Tektronix, Inc.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to Prof. Richard B. Kieburtz for giving me an opportunity to work on this thesis. He was the creative force behind this thesis and he has spent more time teaching and guiding me on a one on one basis than anyone in my whole college career. He provided motivation, encouragement and always had a solution to my problems. I would like to thank him once again for his patience, time and creativity.

I would like to thank the members of my thesis committee for their suggestions and time. I would like to thank Mark Foster, Jianhua Zhu, Keith Billings, Boris Agapiev and Uppili Srinivasan for providing me with help during implementation with their ideas. I would like to thank everyone at Intersoft Systems Inc. especially Dan Cotton, Tom Funk, Larry Ward and Mike Cole for making it possible to work and go to school for the last three years.

I thank the LORD for making this all happen and would like to dedicate this to Him.

TABLE OF CONTENTS

LIST OF FIGURES

CODE GENERATION AND OPTIMIZATION USING A MACHINE DESCRIPTION TABLE AND ATTRIBUTES

Raman Tenneti

Under the supervision of

Professor Richard Kieburtz

ABSTRACT

Machine description tables and attributes are used to specify translations from an intermediate representation (G-code) to a target code representation of programs (for a functional programming language like LML). Code generators were obtained for target machines VAX 11/780, INTEL 286/310 and MOTOROLA 68000 using machine description tables and attributes. A compiler built on this model can automatically perform some machine dependent optimizations.

1. INTRODUCTION

1.1 Motivation

Since the early history of compilers, researchers have been trying to systematize and automate the production of compilers. The most successful aspect of this attempt has been syntax analysis. It is now a common place to use a table-driven syntax analyzer which is automatically constructed from a generalized context-free grammar specifying the syntax of the source language [Aho-Ullman 77].

During the past decade a number of attempts have been made at automating the process of building code generators for compilers. Code generation can be defined as the process of mapping some intermediate representation of the source program into assembly or binary machine-code. Interest in this area is motivated by the following factors.

(1) Proliferation of architectures have led to the design and manufacture of a large number of similar computer architectures (Intel-8086, Z-8000, MC-68000, TMS-9900) which differ in details of instruction set, registers and addressing modes.

(2) Several authors during the last decade have discussed the need for portable compilers [Graham 80, Wulf 80]. They are needed to automate and simplify the process of code generation so as to isolate target-machine specific aspects of translation. It is then possible to retarget the compiler by changing those por-

tions of the compiler which concern the architecture of the machine.

(3) Portable compilers must rest on the formalization of machine-dependent aspects of compilation such as,

(a) addressing units for storing source-language values (e.g. memory, registers and hardware stack),

(b) addressing modes available to access and retrieve operands,

(c) hardware abstractions that are essential to code generation such as machine data types (the groups of bits that can participate as operands to instructions e.g., byte, word, long, quad etc),

(d) code-selection for intermediate representation,

(e) machine-dependent optimizations such as auto-increment, auto-decrement, two/three address instruction variants, specialized instructions.

1.2 Goals

Goals for this thesis are

(1) To define target machine architectures by a table for code generation purposes.

(2) To derive an efficient code generator from the above machine description table by incorporating some machine-dependent optimizations. These optimizations include subsuming addition via auto-increment and similarly subsuming subtraction via auto-decrement.

(3) To use partial simulation of target machine state in order to optimize code selection, and choice of addressing modes and to eliminate redundant loads and stores.

(4) To retain compilation speed by using a single pass code generation scheme.

1.3 Background and Code Generation Research

For our purposes, previous research in code generation can be broadly classified into three categories: formal treatments, interpretive approaches and descriptive approaches. An extensive review critique of these approaches has been done by [Ganapathi 80].

Formal treatments are attempts to deal with code generation mathematically, usually in order to produce optimal or near-optimal code [Aho-Johnson 76]. Research has been with idealized models of computers and thus far it has been concentrated mainly on compilation of arithmetic expressions. Efficient algorithms for generating provably optimal code on a broad class of uniform register machines have been developed for expressions with no common subexpressions [Sethi and Ullman, 1970; Aho and Johnson, 1976]. Once common subexpressions are encountered, or optimal code needs to be generated for machines with irregular architectures, then the problem of optimal code generation has been proven to be combinatorially difficult [Bruno and Sethi 1976; Aho, Johnson and Ullman, 1977a], and heuristic techniques for generating good code have been theoretically analyzed [Aho, Johnson and Ullman 1977b]. The other two classes of research have tended to focus on implementation methods for real computers, often with loss in efficiency of the generated code,

relative to idealized models [Glanville 80].

Interpretive approaches are improvements over ad-hoc code generation. In this approach, information about the target computer is provided in procedural form using special purpose code generation languages and interpreters of the intermediate code of a compiled program. Examples of this approach are UNCOL [Strong 58, Steel 61], the PL/I optimizer of Elson and Rake [Elson 70], the method developed for PL/C [Wilcox 71], and the work of [Donegan 73, 79]. These methods require considerable hand-coding of tedious low level details, making correctness difficult to ascertain and retargeting a chore [Glanville 80]. Thus retargeting requires development of a code generator for every new machine.

In the third class of methods (descriptive approach) the target machine architecture is defined in a machine-readable descriptive form and the macro approach to code generation can be considered part of this approach [Glanville 80]. Fraser, Glanville, Ripken, Catell and Ganapathi have tried to generate code generators automatically from a machine description. Our approach to code generation was stimulated by the work of Ganapathi and Fischer.

Ganapathi [1980] evaluates many of the earlier descriptive approaches. Fraser [1977] has developed a code generator using rule based system. He uses machine-specific rules to perform storage allocation. His code generator is slow and often emits redundant loads and stores.

Catell [1978] used axioms and recursive goal-directed heuristic search algorithm to derive code sequences. In his approach, subgoals are created as search continues. Heuristics are used, both to order subgoal selection and also to order

patterns when trying to match. Sometimes it is hard and time consuming to derive certain code sequences [Ganapathi 80]. He designed a complete code generator for an intermediate language called TCOL.

Ripken [1977] used a dynamic programming algorithm (extending Aho and Johnson's algorithm) [Aho 76] to generate locally optimal code. An implementation of his dynamic programming algorithm is expected to be slow.

Glanville [1978, Graham 80] used context-free parsing techniques to define a translation to machine code. The input to the code generator is a linearized (or flattened) tree representation of the source program. Every possible instruction variant is described by a grammar rule. Pattern matching is provided by simple SLR parsing. It is purely a syntactic approach to the instruction selection problem. The *tree-pattern-matching* is provided in a completely left-operand biased fashion. That is, when generating code for an entire sub-tree, the code for the left operand is selected without considering the right operand. For example, consider the string *op A B*. The addressing mode for A is selected without seeing B. Thus A could be a register-indirect addressing mode on an iAPX-286. Next, B happens to be a memory datum that gets one of the memory addressing modes. Now comes the time to select a machine op-code. The code generator realizes that memory-to-memory operations cannot be performed in one instruction. Thus, it is forced to move A to a register [Ganapathi and Aho 1985]. It is efficient because of context-free recognition and a single pass approach. Because of purely context-free matching, in certain cases it fails to generate optimized code.

Ganapathi[1982, 1984, 1985] has specified code translation rules set by attribute grammars instead of context-free grammars. Semantic attributes and predicates provide automated semantic handling for his code generator. Predicates are used to define the architectural restrictions on the programming model. Attributes are used to track multiple instruction results. Addressing modes are described by separate individual productions and so are op-codes. Addressing mode selection is left-biased in the true tree-pattern matching sense, but selection of op-codes is not biased toward any operand. Op-code productions have symmetric operand patterns. This symmetry enables the code generator to delay decisions regarding destination requirements. In effect, this decision is made on seeing the entire sub-tree for the operator. Thus the target machine code is produced to store the result of evaluation [Ganapathi and Aho 1985].

Kessler [1986] has implemented a retargetable LISP compiler. This compiler uses architectural description (AD) of the target machine to increase portability and performs extensive optimizations.

1.4 Our Strategy

We have selected G-code as the intermediate representation of a compiled, functional language. G-code is designed to run on a stack machine (G machine) to realize a graph-reduction model of evaluation. The stack is used to hold pointers in a run-time traversal of a graph that represents an applicative expression. We divide the translation from G-code to the target machine code into two logical phases (as shown in Figure 1). The execution of the two phases is overlapped. During the first phase (simulation) we simulate the G machine on a single-assignment machine which has an infinite supply of registers. We call these pseudo-registers. During the second phase (code emission) we map pseudo-registers to actual processor registers and memory locations. We define control of the two phases of translation by rules expressed in a table form. The rules use attributes (which define the partial states of both the stack machine and the register machine) to obtain machine dependent optimizations and also use commands to direct the simulation of the stack machine by the pseudo-register machine, and the simulation of the pseudo-register machine by the actual target machine.

The code generator generator after parsing this machine description table will produce a header file (machdesc.h) which is included in the machine independent source code of the code generator.

SIMULATION:

simulate stacks in pseudo-registers

CODE EMISSION:

map pseudo-registers to processor registers and memory locations

Figure 1. Translation between G-code and target machine code

1.5 Thesis Organization

The above mentioned phases have been presented in the following manner in this thesis.

(1) Chapter 2 discusses selection of intermediate representation for the functional programming language LML.

(2) Chapter 3 discusses design of the machine description table, attributes, pseudo-registers and action codes.

(3) Chapter 4 discusses code selection process. This chapter also includes the examples that have been implemented on the VAX 11/780.

(4) Chapter 5 discusses machine-dependent optimization.

(5) In Chapter 6 implementation results for the VAX 11/780 and M68000 processors are presented.

(6) Ideas for improvements to our implementation and future research are in Chapter 7.

(7) Details of intermediate representation, the libraries that are compiled and the machine description tables for the VAX 11/780 and M68000 processors are given in appendices.

2. G-CODE AS INTERMEDIATE REPRESENTATION

In a classical single-language, single-machine compiler an intermediate form of program code is traditionally used for optimization [Ganapathi 82]. Examples of intermediate forms are pseudo-code quadruples, triples, flattened tree representation of programs. Flattened trees can be generalized to directed acyclic graphs [Aho 77] in order to manifest shared values and avoid redundant computations. But these intermediate forms are inadequate for compiler portability. The design of an intermediate representation (IR) is critical to compiler portability and code generation [Ganapathi 82]. The *level* of an IR determines the work to be redone in transporting a compiler to a new machine. If the level is too high, language dependencies creep in. Similarly if the level is too low, machine dependencies seem unavoidable.

Intermediate representation should reflect aspects of the *model of computation* (abstract architecture) but not of any target computer (concrete architecture). Flattened trees (or DAG's) are very general. They don't represent any aspect of an abstract architecture. They leave too much of source-language dependency. So it makes code generation harder (i.e., language dependent). Triples or quadruples reflect the architecture of one register (triples) or zero register (quadruples) RAM machines.

G-code is appropriate to a different abstract architecture, that of the G-machine [Johnsson 84] which evaluates applicative-expression graphs (with value-

sharing) by reduction.

All language dependent and machine-independent issues are handled by this front-end compiler. The back-end of the compiler (which is described by this thesis) translates the G-code (IR) to target machine code. All of the machine-dependent issues are handled by this back-end. With this approach a compiler for a new machine can be easily generated just by changing the control for the G-code to target machine code translation phase.

2.1 Machine-Independent Phases Of Compiler

The following are the machine-independent phases of any source language.

(1) Lexical analysis (scanning).

(2) Syntax analysis in which the string representation of programs is converted into an abstract syntax tree representation.

(3) Semantic analysis which will do binding, type checking and source-to-source transformations that either optimize or simplify subsequent translation steps.

(4) Data flow optimizations which can be accomplished at the time of generating the intermediate code (ex. constant folding, removing loop-invariant computations, etc.).

2.2 G-CODE

G-code instructions define an abstract architecture, the *G-machine*. The G-machine architecture was originally defined by Thomas Johnsson [Johnsson 84] as an

evaluation model for an ML compiler. This is a machine model which supports evaluation of functional language programs by graph-reduction. In this abstract model, programs are functions whose definitions have been given an operational interpretation as code sequences for the G-machine.

The G-machine evaluates applicative expressions, i.e. applications of functions to argument expressions. Such expressions are represented by a graph in a dynamically-allocated, list-structured memory(G-memory). During the process of evaluation, the graph is mutated by a series of reduction steps until it reaches a normal form. Graph reduction is accomplished through the manipulation of a traversal stack that contains pointers into memory.

The traversal stack contains pointers directly to the argument expressions and to the principal application that is being reduced. To reduce the expression, a program compiled for the function *f* is executed. After reduction, the principal application node is overwritten with the representation of its value.

A sequential evaluator has been developed at the Oregon Graduate Center [Kieburtz 85] based on that abstract model. This evaluator, which will be referred to as the G-machine, performs graph reduction where expressions are represented as graphs rather than strings. The G-machine uses a P (pointer) stack which holds the pointers to graph memory and a V (value) stack which holds the intermediate values of basic types (integer, boolean, character) during expression evaluation, and G-memory which is a dynamically allocated list-structure memory. The set of G-code instructions and their meanings are given in Appendix A.

The following example is a LML program to compute the square of 5. The G-code instructions that are generated by the front-end compiler and the execution of those instructions on the G-machine is given in Figure 2.

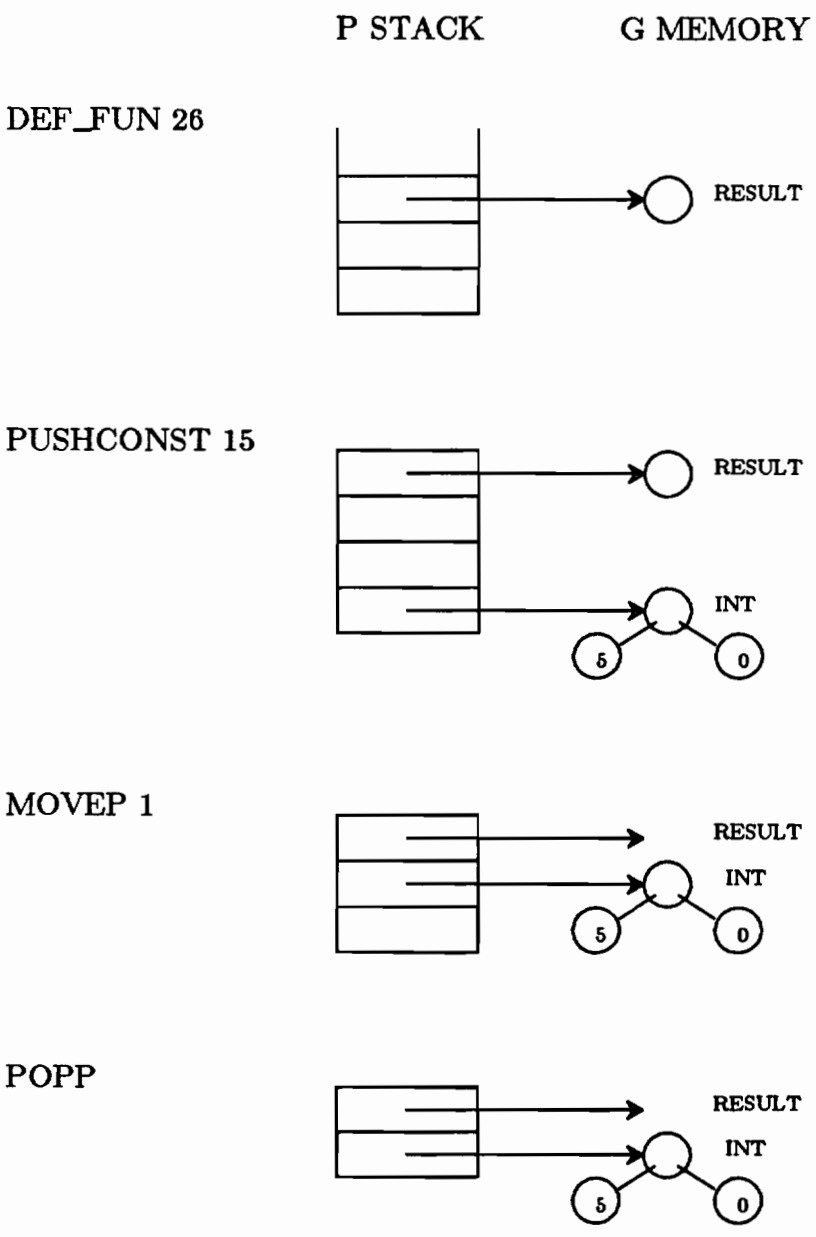
```
let sqr = \x. x * x in sqr 5;
```

```
imports:
exports:
Initial graph image:
  0: INT                 0
  5: FUN     (1)        24
 10: FUN     (2)        26
 15: INT                 5
G-code text:
  0: DEF_FUN            26
  4: PUSHCONST          15
  8: MOVEP               1
 10: POPP                0
 12: JGLOBFUN           24
 16: END_FUN            26

  0: DEF_FUN            24
  4: EVAL                2
  6: GET_FST             0
  8: GET_FST             0
 10: MUL                 1
 12: MOVEV               0
 14: POPP                0
 16: RET_INT             0
 18: END_FUN            24
```

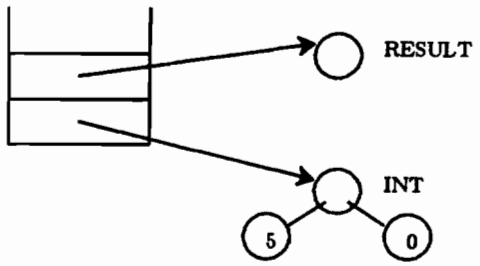
PUSHCONST 15 instruction will push a pointer to INT 5 in the G-memory onto the P-stack. MOVEP 1 instruction will move P[0] to P[2] and pop the P-stack. The POPP instruction will pop the P-stack again. The first GET_FST 0 instruction will push the first of a graph node whose value is basic type into the V-stack. MUL instruction multiplies the top two elements of the V-stack and the result is stored on top of the V-stack. MOVEV 0 will move a value from V[0] to V[1] and pop the V-

stack. RET_INT will update the result node with the basic value from V[0] and returns from the function.

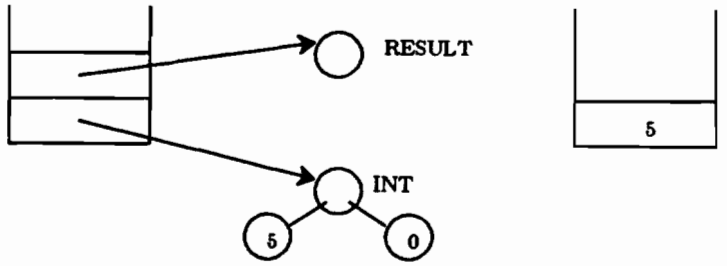


P STACK G MEMORY V STACK

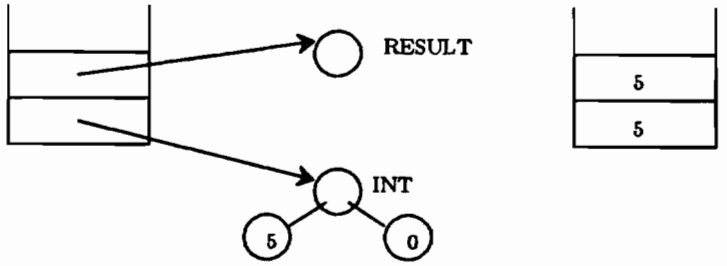
JGLOBFUN 24



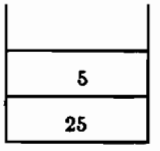
GET_FST 0



GET_FST 0



MUL



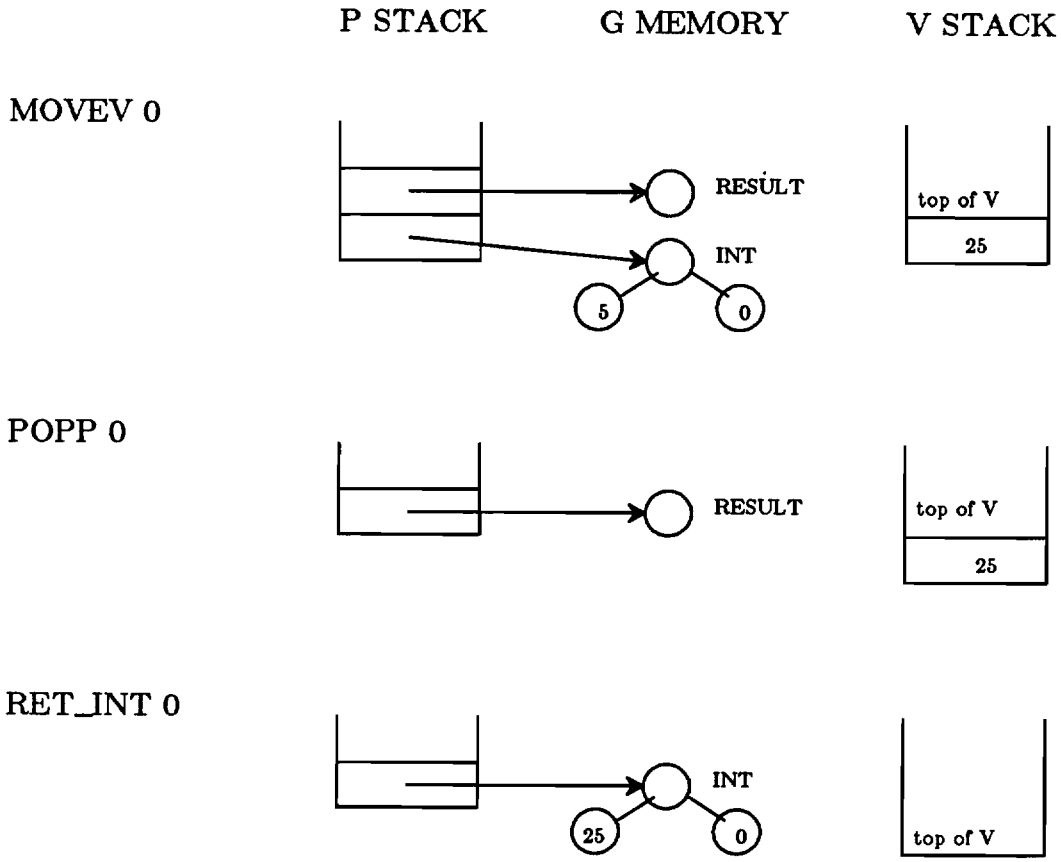


Figure 2. Example of sqr 5

3. MACHINE DESCRIPTION USING ATTRIBUTES AND ACTION CODES

3.1 Machine Description Table

Code generation requires the description of the following aspects of a machine architecture:

(1) A fully automatic code generator might use a formal definition of the target machine codes to infer a code sequence that gives a correct operational interpretation to the intermediate code sequence of a program. Fully automatic code generation is far beyond the present state of the art, however. Instead, code generation is specified by translation sequences that give a correct operational interpretation to individual instructions of the intermediate code.

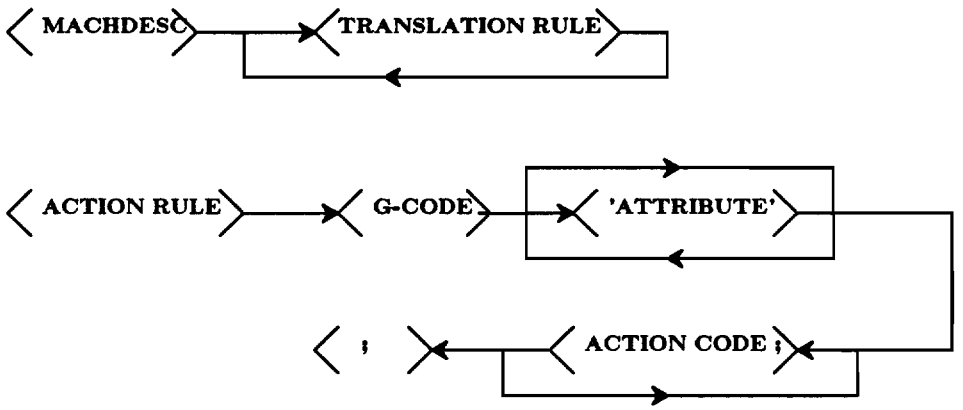
(2) the assembly formats for the target machine instructions,

(3) addressable units for storing the source-language values (e.g memory, registers and hardware stacks). The number of registers available and storage allocation for the P stack and G memory have to be specified in a prefix file.

Recall that we have simulated the stack machine (G machine) through a pseudo-register machine. This machine has infinitely many registers (inexhaustible supply of registers), and is a single variable assignment machine. We have specified the commands to simulate the stack machine on the pseudo-register machine in the machine description table. The machine description table also has entries to map the

pseudo-registers to the target machine registers and memory locations. In this chapter we will discuss the pseudo-registers and the commands to manipulate the stacks on a pseudo-register machine.

In the machine description table translation actions are defined by sequences of translation rules associated with individual G-code instructions. The syntax of a translation description is shown in Figure 3.



G-CODE instructions are listed in APPENDIX A

ATTRIBUTES are listed in SECTION 3.3

ACTION CODES are listed in SECTION 3.2

Figure 3. Syntax of Translation Rules

For each G-code instruction the G-code is first matched in the machine description table and the table can have a set of rules for a G-code instruction. For a G-code it will match the attributes. The attributes will enable that translation

rule and it will execute the action codes for that translation rule.

The generation of target machine code is specified by the machine description table. For each G-code instruction there could be one or more of the following.

1) The attributes that are to be matched for instruction selection.

2) The operations on the P and V stacks (e.g. pushing and popping).

3) The code to be delayed or emitted if there is any.

4) Register allocation if an instruction (ex. GET_FST) requires a target machine register.

3.2 Action Codes

Action codes describe the action taken by the code generator in composing a translation sequence for an individual G-code instruction. These actions include

(1) simulation actions, affecting the state of the P and V stacks of the abstract G-machine,

(2) simulation actions updating the contents and attributes stored in pseudo-registers of the abstract machine,

(3) code emission actions directing the production of code sequences for the target machine,

(4) actions to allocate the target machine registers.

Action codes can be divided into two categories:

1) Commands which update the contents of pseudo registers,

2) Expressions which may refer to the contents of pseudo registers but do not update their contents.

The following symbols are used in the action codes either to denote a stack or a pseudo-register.

Token	Meaning
$N, $R	Pseudo-register variables. These variables are given value by an ALLOC_SUDO command or by assignment. This is used in ADD instruction in the following manner $N = ALLOC_SUDO "add12" V[0] V[1] $N
$N.cnst	This variable stores the literal/constant value in the pseudo-register. This literal value then becomes an attribute of the pseudo-register that holds it, and may be tested by subsequent translation actions.
$N.reg	This represents the target machine's hardware register whose index is stored in the pseudo-register.
d_label	This is the instruction number that precedes each G-code instruction.
op1	Denotes the first operand of a G-code instruction.
op2	Denotes the second operand of a G-code instruction. In the FUN instruction op1 is the number of arguments and op2 is the code address.
POP_V	Pops the simulated V stack one place.
POP_P	Pops the simulated P stack one place.
PUSH_V	Pushes a pseudo-register on top of the V stack. PUSH_V $N;

or
PUSH_V V[1];

PUSH_P — Pushes a pseudo-register on top of the P stack.

GET_IN_REG — Allocates a target machine register.

EMIT — Emits the code that follows it. For example VAX 11/780 code for ADD is:

```
EMIT "addl2 4(%VS), (%VS)";
```

ALLOC_SUDO — Allocates a pseudo-register.

MACH_CODE — Any string of characters that is embedded between quotes (") is considered as target machine code.

The code generator will interpret the following action codes and will execute different functions to simulate the stack machine on a pseudo-register machine.

```
ACTION CODE ----->     COMMANDS  |  EXPRESSIONS

COMMANDS ----->

    POP_V

    POP_P

    PUSH_V             V [ op1 ]

    PUSH_V             $N

    PUSH_V             op1

    PUSH_P             P [ op1 ]

    PUSH_P             $N

    PUSH_P             op1

    PUSH_P             V [ NUMBER ]

    MACH_CODE          V [ NUMBER ]

    MACH_CODE          V [ NUMBER ]      V [ NUMBER ]

    MACH_CODE          V [ NUMBER ]      V [ NUMBER ]  $N

    MACH_CODE          NUMBER $R.reg  $N.reg

    EMIT               MACH_CODE

    DELAY              MACH_CODE

EXPRESSIONS ----->

    V [ NUMBER ]   =  V [ NUMBER ]
```

```
V [ NUMBER ]    =  $N

V [ op1 ]       =  V [ NUMBER ]

P [ NUMBER ]    =  P [ NUMBER ]

P [ NUMBER ]    =  $N

P [ op1 ]       =  P [ NUMBER ]

$N              =  P [ NUMBER ]

$N              =  P [ op1 ]

$N.cnst         =  op1

$N              =  ALLOC_SUDO

$R              =  GET_IN_REG  $R

$R.reg          =  ALLOC_REG
```

3.3 Attributes and Pseudo-Registers

We have used attributes to propagate information about the state of the register machine. Attribute values indicate the partial state of the machine. An abstract pseudo-register holds the value of an element of either the P or V stacks. Any finite operation on either of the P or V stacks can be simulated by loads and stores to the pseudo-registers. The P and V stacks are represented in simulation by linked list structures. The contents of each element of these list structures is a pointer to a pseudo-register.

Each pseudo-register stores the attributes of the value contained in the P or V stack element that it represents. Attributes are used to represent literal values, when they are known, as well as target machine register assignments.

Eight attribute evaluation functions are used in the current implementation. In the following table these functions and their meanings are given. V[n] represents the nth element of the V stack and V[0] represents the top of V stack.

(1) *op1_lit_0* : checks whether top of the V stack V[0] is known to be equal to literal 0.

(2) *op2_lit_0* : checks whether V[1] is known to be equal to literal 0.

(3) *op1_lit_1* : checks whether V[0] is known to be equal to literal 1.

(4) *op2_lit_1* : checks whether V[1] is known to be equal to literal 1.

(5) *arg_0* : checks whether the argument field of a G-code instruction is known to be equal to literal 0.

(6) *arg_1* : checks whether the argument field of a G-code instruction is known to be equal to literal 1.

(7) *IS_EQUALS* : checks whether the top two elements of the V stack are known to be equal.

(8) *is_in_reg* : checks whether a target machine hardware register has been assigned to store the contents of an abstract pseudo-register or not.

Each cell of the P stack logically contains a pointer to the G memory node.

In our implementation a register is used to store the address of the memory location.

We have used the target machine hardware registers for the V stack elements in our implementation for VAX 11/780.

3.4 Delayed Code Emission

The machine description table will specify whether for a particular G-code instruction emission of the target machine code is to be delayed or immediate (this can be specified with the action codes DELAY or EMIT). We decided to delay the code whenever it is not compulsory to emit code. The code that is delayed is stored in the pseudo-register's code buffer. Each pseudo-register has a dynamically allocated code buffer and each entry of the code buffer has a boolean tag to indicate whether code is delayed or emitted. Code is emitted from the delayed code buffer at the end of each basic block (i.e., at the end of conditional expressions and at the end of functions) or whenever there is an access to G-memory.

An example of usage of the attributes and action codes for G-code instructions ADD and SUB is given below.

```
ADD   'op1_lit_1'   :   "incl" V[1];
                        V[0] = V[1];;

ADD                 :
                        "addl2" V[0] V[1] $N;;
```

In the above example, ADD is the G-code for which the translation is being specified in the machine description table. 'op1_lit_1' is the attribute which checks whether V[0] is known to be equal to 1 or not. "incl" V[1] is the action code which specifies delay the code to do auto-increment on V[1] and the action code V[0] = V[1] indicates that V[0] should point to the V[1]'s pseudo-register. If V[0] is not equal to 1 then the default action code "addl2" V[0] V[1] $N will be executed.

```
SUB    IS_EQUALS   :        "clrl" V[0];;
```

In the above example IS_EQUALS is an attribute which indicates that if the two operands of arithmetic operation SUB are known to be equal then perform the action codes specified after colon (:). The above attribute checks the constant values of the top two elements of the V stack. These constant values are stored in the abstract pseudo-registers, and if they are equal "clrl" code will be delayed in the code buffer of V[0].

It is advisable to define a default entry for each G code instruction. When none of the attributes match for an instruction the code generator generator will perform the action codes specified in the default entry. The target machine code is generated from the specifications of the machine description table. The machine description table can be used either for a simple macro expansion process or to generate efficient code based on attributes and simulation of stacks. By using and propagating attributes we could achieve machine-dependent optimization and thus were able to generate better target machine code.

3.5 Instruction Formats

In this implementation the user has to provide the code generator generator with the structure of the target machine instruction set (if he is not doing simple macro expansion). The user is given different formats so that the code generator can frame the target machine instructions. The user will specify what format each target machine instruction belongs to. This information is used while parsing the machine description table. The following is a list of format numbers that are used to define the VAX 11/780 instructions.

Format 1 r0

Format 2 r0, r1

Format 3 r0, r1, r2

r0, r1, r2 represent target machine registers. In the above formats, the results are stored in the last register.

Example to illustrate the format table (Appendix G):

ADD : "addl2" V[0] V[1] $N;;

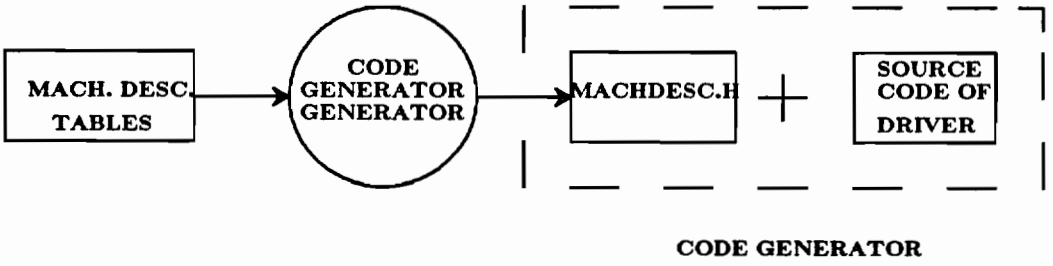
In the above example "addl2" instruction is stored in the code buffer. Its format is #2. If V[0].reg = 'r5' and V[1].reg = 'r7' then the code generator will frame the target machine instruction as *addl2 r5, r7.*

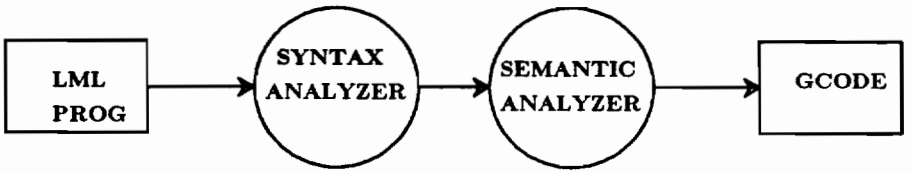
In summary, the components of target architectures needed for instruction selection are described to a code generator generator in the form of tables. The next chapter describes translation of the G-code to target machine code using the translation tables.

4. PROCESS OF CODE SELECTION

4.1 Code-Generator Generator

Code generation is the process of transforming the intermediate representation of the source program (LML) into assembly or binary machine-code [Ganapathi 80]. Generation of a code generator for a target machine is a two step process. The target machine is described using attributes and action codes in the form of a table. In the first step, this machine description table is input to the code-generator generator (CGG). The CGG after parsing this table will create an include file (*machdesc.h*). The second step is to recompile the code generator with the newly generated *machdesc.h* file. The code generator consists of a driver that accepts G-code and a set of functions to perform action codes as specified by the *machdesc.h* file. The block diagram of code generator generator is shown in Figure 4.





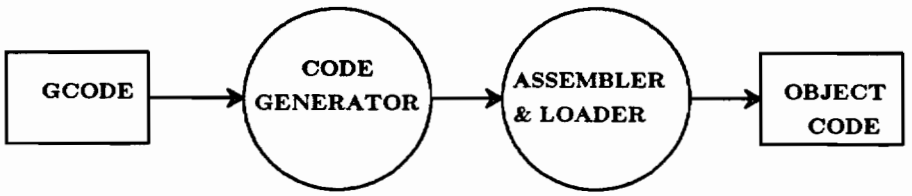


Figure 4. Block diagram of code generator generator

4.2 Instruction Selection Based on Patterns and Attributes

The target machine description table should contain one or more entries for each G-code instruction. For each G-code instruction the RHS of an action rule has action codes and the LHS of the action rule has the attributes that are to be satisfied. The user writing the target machine description table can prescribe multiple action codes for a G-code instruction. The user has the flexibility to generate different target machine instructions for a G-code instruction based on attribute matching in the left hand side of an action rule. The target machine description table dictates a target machine instruction selection when a pattern is matched (so this can be considered as a sequence of pattern-action statements).

Within the *machdesc.h* file, the G-code instructions are stored in the form of op-codes. The CGG also stores the bit vectors representing the attributes and the function indexes of action codes in the *machdesc.h* file. The code generator looks up the target machine description table to match the G-code that is to be compiled and selects the first entry whose attributes are matched.

A non-optimized machine description table can be generated with little effort, if the user who is writing the machine description table has a good understanding of both the G-code and the target machine code. With some additional effort an optimized machine description table can be obtained.

4.3 Examples of Translating G-code using Attributes

The following is a function to compute square of the number 5. The syntax for the following source language is LML. This function has been compiled using Prof. Richard Kieburtz's LML compiler which generates G-code (intermediate code). The G-code is also given below and the explanation of each G-code instruction is given in Appendix A. The machine description table that translates G-code to the VAX 11/780 code is given in appendix C.

let sqr = \x. x * x in sqr 5;

```
Inst.          G-CODE                VAX 11/780 CODE
no

          0:   INT     0
1                                     ML0:
2                                     .long 1,0,0
          5:   FUN     24
3                                     ML5:
4                                     .long 3, _ML24,1
          10:  FUN     26
5                                     ML10:
6                                     .long 3, _ML26,2
          15:  INT     5
7                                     ML15:
8                                     .long 1,5,0

          DEF_FUN      26
9                                     .text
10                                    .globl _ML26
11                                    _ML26:
          PUSHCONST    15
12                                    movl $ML15, -(%PS)
          MOVEP        1
13                                    movl (%PS)+, 4(%PS)
          POPP         0
14                                    addl2 $4, %PS
          JGLOBFUN     24
15                                    jsb _ML24
16                                    rsb
```

```
        DEF_FUN     24
17                                  .text
18                                  .globl _ML24
19                                  _ML24:
        EVAL
20                                  jsb _eval
        GET_FST     0
21                                  movl (%PS), r0
22                                  movl 4(r0), r1
        GET_FST     0
23                                  movl 4(r0), r2
        MUL         1
24                                  mull2 r1, r2
        POPP
25                                  addl2 $4, %PS
        RET_INT
26                                  movl (%PS), r6
27                                  movl $VAL, (r6)
28                                  movl r2, 4(r6)
29                                  movl $0, 8(r6)
30                                  rsb
```

In the above example the program should start executing from DEF_FUN 26. The instructions prior to that are used to initialize graph memory.

In our implementation we have used memory locations to implement the P stack and G memory. A cell of G memory node contains 3 words. The 1st word stores the tag of the node and the next two words store the data values (integer, boolean, pointer etc) as shown in Figure 5.

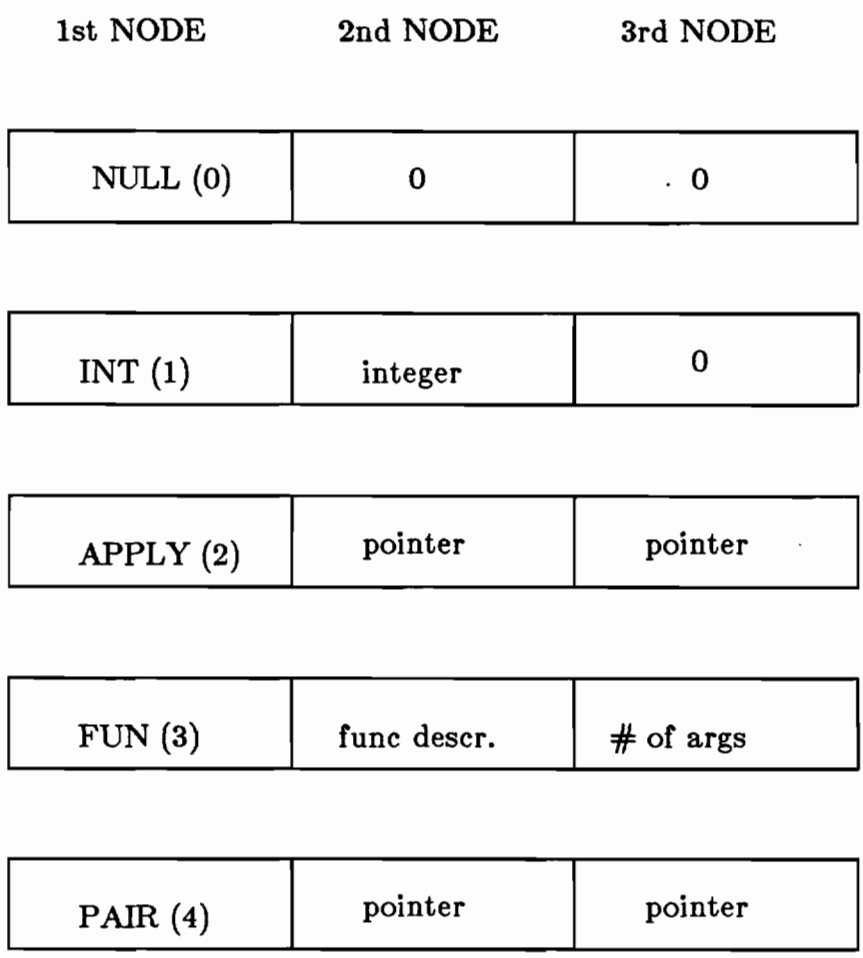


Figure 5. G-memory formats

The P stack stores pointers to the G memory. In the above example %PS indicates the top of P stack. r0, r1, r2 are VAX 11/780's target machine registers.

The action codes that are executed for the GET_FST 0 instruction will be explained below.

The GET_FST instruction (as indicated in Appendix A) fetchs the contents of the first cell of the G node pointed to by the top of P stack onto the V stack. This operation on the VAX 11/780 requires loading of top of the P stack into a register

and indexing from the contents of that register [4(r0)] to get the first element.

The machine description table has the following entries for the GET_FST instructions (appendix C).

```
GCODE     ATTRIBUTES        ACTION CODES

GET_FST   'is_in_reg'   :
                            $R = P[op1];
                            $N = ALLOC_SUDO ;
                            $N.reg = ALLOC_REG ;
                            PUSH_V $N;
                            "movl" 4 $R.reg $N.reg;;
GET_FST                 :
                            $R = P[op1];
                            $R.reg = GET_IN_REG $R;
                            $N = ALLOC_SUDO ;
                            $N.reg = ALLOC_REG ;
                            PUSH_V $N;
                            "movl" 4 $R.reg $N.reg ;;
```

When the compiler encounters the GET_FST 0 instruction it will look in the machine description table for the translation. The first entry in the machine description table for the GET_FST instruction indicates that if the 'is_in_reg' attribute is true then execute the action codes specified after the colon. The 'is_in_reg' attribute executes a function which will check whether the simulated P[0]'s pseudo register has been assigned to a target machine register or not.

The compiler checks the simulated P stack's 0th element's 'is_in_reg' attribute. If that attribute is true, then CGG will execute the action codes specified accordingly. In the above case it is not true, so it will take the default case. We have explained the action codes for the default entry in the following paragraphs.

1. $R = P[op1]$ In this action code $R represents a temporary pseudo-register and this will point to P[0] (because operand 1 is 0). This action code doesn't generate any machine code as shown in Figure 6.

2. $R.reg = GET\_IN\_REG$ R This action code indicates that the value of $R (in this example $R and P[0] point to the same pseudo-register) is to be loaded into the target machine's hardware register. The GET_IN_REG action code will get a free register (if no register is free an algorithm to get a free register is executed which is discussed in chapter 5) and will set the attribute 'is_in_reg' to true for P[0]. The code that is delayed because of this action is *movl (%PS), r0.*

3. $N = ALLOC\_SUDO$ This action code will get a new pseudo-register and $N represents this new pseudo-register. This action doesn't generate any machine code.

4. $N.reg = ALLOC\_REG$ This action code is supposed to allocate a target machine register for the V stack. This action will return 'r1' as the register in which the V stack element is to be stored.

5. $PUSH\_V$ N This action code will push the newly obtained pseudo-register on top of the simulated V stack. This action doesn't generate any machine code.

6. "*movl*" *4* $R.reg $N.reg This action code delays the code to move the P stack's element to the V stack. Delaying the code emission is a default and it is very easy to delay the code rather than deciding when to emit or delay the code. The code that is delayed because of this action is *movl 4(r0), r1.*

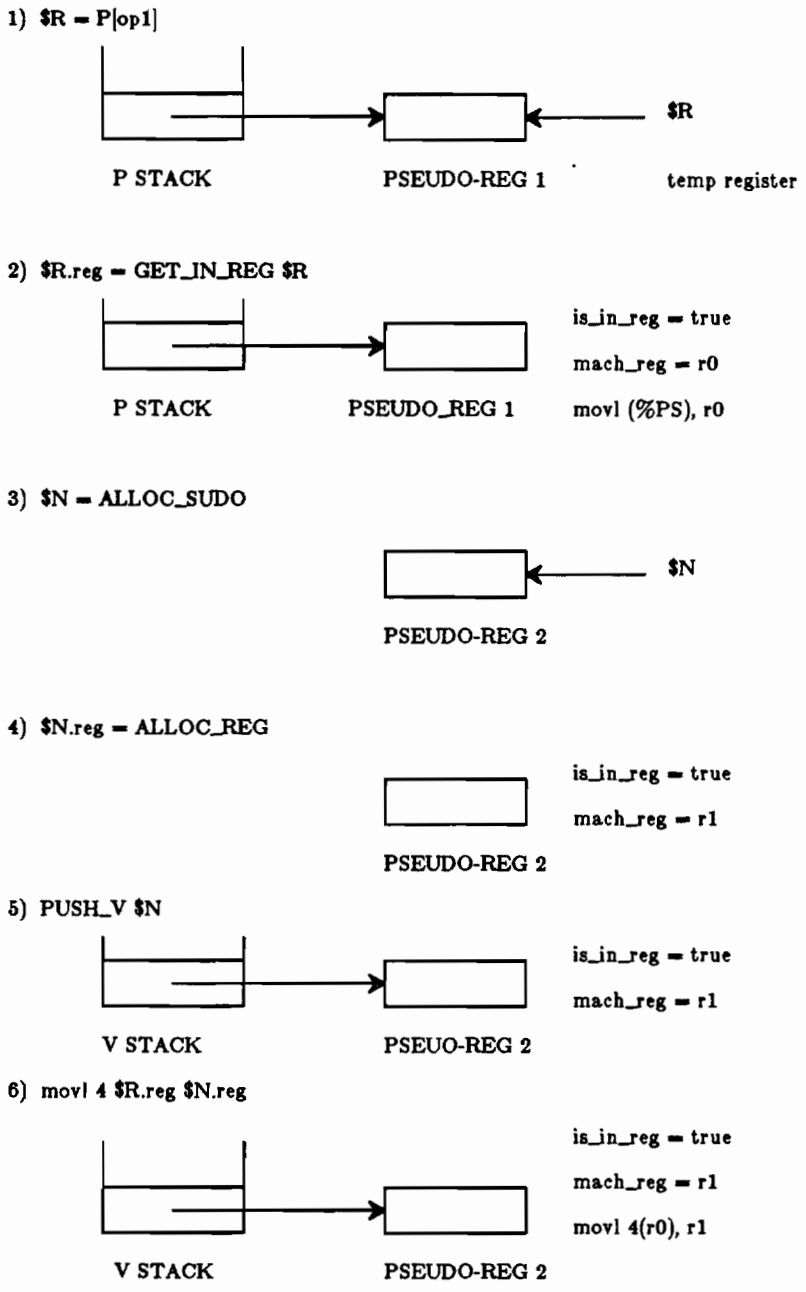


Figure 6. Explanation of ACTION CODES for GET_FST

The consequence of the above action codes is the following target machine code.

```
movl (%PS), r0
movl 4(r0), r1
```

When the compiler comes across the next GET_FST 0 instruction the 'is_in_reg' attribute of P[0] will be true. Because of that, the compiler matches the first entry for GET_FST, and generates only the following code.

```
movl 4(r0), r2
```

The action codes for the first entry of the GET_FST instruction are same as the default entry except for the action code# 2 ($R.reg = GET_IN_REG $R).

We have done similar optimizations for the V stack. A non-optimized target machine table (or code generated by simple macro expansion or pattern matching techniques) wouldn't have been able to generate the above code.

In summary, G-code is translated to the target machine code by CGG using the target machine description table. With the help of attributes and the simulation of the P and V stacks through pseudo-registers, the code generator can produce target machine code. A basic code generator can be implemented by specifying a single action code sequence for each G-code. Machine dependent optimizations can be achieved by adding attribute-guarded action code sequences to the machine description table.

5. MACHINE-DEPENDENT OPTIMIZATION AND REGISTER ALLOCATION

Compilers that do optimization produce a more efficient representation of user programs. The optimization phase normally aims both for compact object code size and execution speed [Ganapathi 80]. A large number of these optimizations are machine-dependent. The optimization strategies include:

(1) Using special instructions to subsume additions and subtractions of a constant value (e.g., using auto increment and auto decrement) [Ganapathi 80].

(2) Peephole optimizations (for instance, the UNIX C compiler makes a separate pass over assembly code to improve short code sequences [Ritchie 78]). Fraser recently has implemented a machine-independent peephole optimizer that tries to optimize adjacent pairs of assembler instructions [Fraser 80]. For a window of more than two instructions, peephole optimization is very slow and requires more 'context' information [Ganapathi 80]. Attributes are a good means of maintaining the contextual information.

(3) Avoiding redundant loads and stores into or from target machine registers and using target machine registers in preference to memory locations.

The problem with older compiler design is that there were lot of hand-coded optimizations in code generators. It is difficult to follow and debug the code of the compiler when it is written in this manner. Expressing machine-dependent optimizations using attributes can make it easier to write and debug a compiler.

In our code generator we implemented the following optimizations of the target machine code.

(1) Identifying opportunities for special machine-dependent instructions (auto-increment and auto-decrement for VAX 11/780) through attributes.

(2) Avoiding redundant loads and stores into registers.

(3) representing V-stack cells with machine registers avoids code for a memory-mapped stack.

5.1 Special instructions

In this implementation the code generator uses attributes in order to identify opportunities to generate special instructions. Simulation of the P and V stacks helps to schedule instructions. Because of attributes like 'op1_lit_1' 'op2_lit_1' the code generator can subsume addition and subtraction via auto-increment and auto-decrement. The following entries of machine description table indicate how auto-increment and auto-decrement can be used for ADD and SUB instructions.

```
ADD     'op1_lit_1' :
                              "incl" V[1];
                              V[0]=V[1]; ;
ADD     'op2_lit_1' :
                              "incl" V[0]; ;

SUB     'op1_lit_1' :
                              "decl" V[1];
                              V[0] = V[1];;
```

In the above instructions the attribute 'op1_lit_1' means that the content of V[0] is constant 1. For example in the case of the SUB instruction, if V[0] is equal to 1, the code generator can emit a special instruction to decrement (decl) V[1] by 1.

5.2 Deleting Redundant Code

The code generator that is generated from the machine description table successfully avoids many redundant loads and stores. Sometimes the code generator doesn't even generate any code (e.g. the MOVEV instruction doesn't generate any code. It acts to pop the simulated V stack. When the code generator comes across a MOVEV instruction, it will assign V[0] to V[1] and it will pop the V stack and releases the hardware register that is assigned to V[1]).

By delaying code the code generator can determine where is the last use of a P stack element and the register allocation algorithm will reuse a register based on its last use. In the example that is discussed in section 4.3 the second GET_FST 0 instruction doesn't load the P stack element into a register again because the 'is_in_reg' attribute of P[0] is true (a preceding GET_FST 0 instruction would have loaded the P stack element from memory into a register 'r0').

5.3 Register Allocation

The machine description table has an action code GET_IN_REG, which will move the contents of the P stack from memory into a register. This is to take advantage of the cheaper address path. The status of register usage is maintained in the form of a bit vector (e.g., if registers 1 and 3 are being used out of the 8 available registers, then bits 1 and 3 will be set to 1 and the rest of the bits will be zero). The function that implements GET_IN_REG will check the bit vector to determine whether there are any free registers. If there is no free register available then the register allocation algorithm is invoked. Otherwise, it will allocate the first available free register, set the corresponding bit to indicate that the particular register is occupied, and set the attribute 'is_in_reg' to true for the pseudo-register that represents the P stack's contents. If there is no free register available then the code generator has to dump the contents of a register that is being used (if the contents of the register are not in the memory), so that a free register can be obtained. In this implementation a free register is obtained when the following conditions are met.

(1) Free a register at its last use.

(2) To free all registers at the end of a conditional branch.

(3) Free the register that will not be used for the longest time (this is preemptive).

Some of the special instructions that are handled by the CGG are conditional expressions. The code generator saves the simulated P and V stacks and the register usage bit vector, before traversing the true branch of the conditional expression. It

will simulate the P and V stacks through pseudo-registers during this branch. But before traversing the false branch of the conditional expression, the P and V stacks and register usage bit vector are restored and register allocation will continue. At the end of the conditional expression, the code is emitted and all the registers are freed.

The above register allocation algorithm has been implemented in the following manner. One of the critical factors is to find the *last use* of a pseudo-register (the P and V stack elements point to pseudo-registers). To determine the last use of an element in a block, the code generator will store the G-code instructions in a code buffer and will simulate the P and V stacks (shadow stacks) through pseudo-registers. During this phase target machine code is delayed, not emitted, but the instruction number at which a pseudo-register is used is updated (i.e., each pseudo-register has the last instruction number at which it is used). When the code generator comes across END_FUN instruction it will go through the G-code buffer to generate target machine code and will simulate the P and V stacks with pseudo-registers. When the code generator has to get a free register it will check the shadow P stack's pseudo-registers to determine which register is not used for the longest time and will dump its contents into memory and will set it's 'is_in_reg' attribute to false and will free the register for re-use.

In summary because of attributes and because of simulating the P and V stacks through pseudo-registers we were able to do machine-dependent optimizations and we have avoided redundant code. We have implemented a simple register allocation algorithm. The code generator is able to avoid redundant loads and stores and

is also able to take advantage of cheaper addressing modes. The compiler is also able to generate special machine-dependent instructions (e.g., subsuming of addition via auto increment and auto decrement).

6. IMPLEMENTATION AND RESULTS

We have implemented the code generator for the VAX 11/780 and the SUN Workstation. The code generator has produced efficient code and is not slow in generating code. An un-optimized code generator (macro expansion) for G-code has produced 10-30% more code than the optimized code generator. The amount of optimization that is achieved is entirely program dependent, but the code generator is able to produce on the average 10% less code. The goals of the implementation are

(1) use one-pass parsing to generate code so that efficiency of code generation is not lost.

(2) flexibility to add optimizations incrementally; all optimizations are optional.

(3) to take advantage of attributes to generate target machine code.

The following pages contain a listing of the code generated by the optimized code generator and unoptimized code generator on VAX 11/780. The machine description table for the optimized code generator is given in appendix C and the table for the unoptimized code generator is given in appendix D.

In the following examples the number before the target machine instruction indicates the type of optimization that was obtained.

1 --- Constant folding.

2 --- Registers are used instead of memory locations.

3 --- Deletion of code during optimization.

4 --- Specialized instructions were used (auto-increment, auto-decrement).

```
=======================================================

letrec linfib = \x.\y.\n.

    if n = 0 then x
    else
    if n = 1 then y
    else
    linfib y (x+y) (n-1)

in linfib 0 1 10

=======================================================
```

VAX 11/780 CODE WITHOUT OPTIMIZATION

```
    .globl _Fmain
    _Fmain:
    jsb _ML34
    rsb

    ML0:
    .long 1,0,0

    ML5:
    .long 1,0,0

    ML10:
    .long 3,_ML24,3

    ML15:
    .long 3,_ML32,1

    ML20:
    .long 3,_ML33,2

    ML25:
    .long 3,_ML34,2

    ML30:
    .long 1,10,0

    ML35:
    .long 1,1,0

    ML40:
    .long 1,0,0

    .text
    _ML24:

    movl 4*2(%PS), -(%PS)

    jsb _eval

    addl2 $4, %PS

1   movl 4*2(%PS), r0
2   movl 4(r0), -(%VS)
```

```
 0: INT    0

 5: INT    0

10: FUN    (3) 24

15: FUN    (1) 32

20: FUN    (2) 33

25: FUN    (2) 34

30: INT    10

35: INT    1

40: INT    0

 0: DEF_FUN          24

 4: COPYP             2

 6: EVAL              4

 8: POPP              0

10: GET_FST           2
```

```
2   movl $0, -(%VS)

2   subl3 (%VS), 4(%VS), (%VS)

3   movl (%VS)+, 4*0(%VS)

3   movl (%VS)+, 4*0(%VS)

    jneq _MLL28

1   movl 4*0(%PS), -(%PS)

    jsb _eval

    movl 4*4(%PS), r1
    movl (%PS)+, r0
    movl (r0), (r1)
    movl 4(r0), 4(r1)
    movl 8(r0), 8(r1)

    addl2 $8, %PS

    addl2 $4, %PS

    rsb

    _MLL28:

1   movl 4*2(%PS), r0
2   movl 4(r0), -(%VS)

3   movl $1, -(%VS)

4   subl3 (%VS), 4(%VS), (%VS)

3   movl (%VS)+, 4*0(%VS)

3   movl (%VS)+, 4*0(%VS)

    jneq _MLL30

1   movl 4*1(%PS), -(%PS)

    jsb _eval
```

```
12: GET_BYTE          0

14: SUB               1

16: MOVEV             0

22: MOVEV             0

18: JNOT_ZERO        28

24: COPYP             0

26: EVAL              4

28: UPDATE_P          4



30: POP2              0

32: POPP              0

34: RET               0

36: LABEL            28

40: GET_FST           2


42: GET_BYTE          1

44: SUB               1

46: MOVEV             0

52: MOVEV             0

48: JNOT_ZERO        30

54: COPYP             1

56: EVAL              4

58: UPDATE_P          4
```

```
    movl 4*4(%PS), r1
    movl (%PS)+, r0
    movl (r0), (r1)
    movl 4(r0), 4(r1)
    movl 8(r0), 8(r1)

    addl2 $8, %PS

    addl2 $4, %PS

    rsb

    _MLL30:

    movl 4*2(%PS), -(%PS)

    movl $ML15, -(%PS)

    movl $APPLY, (%GM)+
    movq (%PS)+, (%GM)+
    moval -12(%GM), -(%PS)

    movl (%PS)+, 4*2(%PS)

1   movl 4*1(%PS), -(%PS)

    movl 4*2(%PS), -(%PS)

    movl 4*2(%PS), -(%PS)

    movl $ML20, -(%PS)

    movl $APPLY, (%GM)+
    movq (%PS)+, (%GM)+
    moval -12(%GM), -(%PS)

    movl $APPLY, (%GM)+
    movq (%PS)+, (%GM)+
    moval -12(%GM), -(%PS)

    movl (%PS)+, 4*2(%PS)

    movl (%PS)+, 4*0(%PS)

    jsb _ML24
    rsb
```

```
60: POP2            0

62: POPP            0

64: RET             0

66: LABEL           30

70: COPYP           2

72: PUSHCONST       15

76: MK_APP          0


78: MOVEP           2

80: COPYP           1

82: COPYP           2

84: COPYP           2

86: PUSHCONST       20

90: MK_APP          0


92: MK_APP          0


94: MOVEP           2

96: MOVEP           0

98: JGLOBFUN        24
```

```
                                                    0: DEF_FUN          33
    .text
    _ML33:

                                                    4: COPYP            1
1   movl 4*1(%PS), -(%PS)

                                                    6: EVAL             3
    jsb _eval

                                                    8: POPP             0
    addl2 $4, %PS

                                                   10: EVAL             3
    jsb _eval

                                                   12: GET_FST          0
1   movl 4*0(%PS), r0
2   movl 4(r0), -(%VS)

                                                   14: GET_FST          1
1   movl 4*1(%PS), r0
2   movl 4(r0), -(%VS)

                                                   16: ADD              1
2   addl2 4(%VS), (%VS)

                                                   18: MOVEV            0
3   movl (%VS)+, 4*0(%VS)

                                                   20: POP2             0
    addl2 $8, %PS

                                                   22: RET_INT          0
    movl (%PS), r1
    movl $VAL, (r1)
2   movl (%VS)+, 4(r1)
    movl $0, 8(r1)
    rsb

                                                    0: DEF_FUN          32
    .text
    _ML32:

                                                    4: GET_FST          0
1   movl 4*0(%PS), r0
2   movl 4(r0), -(%VS)

                                                    6: GET_BYTE         1
3   movl $1, -(%VS)

                                                    8: SUB              1
4   subl3 (%VS), 4(%VS), (%VS)

                                                   10: MOVEV            0
3   movl (%VS)+, 4*0(%VS)

                                                   12: POPP             0
    addl2 $4, %PS

                                                   14: RET_INT          0
    movl (%PS), r1
    movl $VAL, (r1)
```

```
2  movl (%VS)+, 4(r1)
   movl $0, 8(r1)
   rsb

   .text
   _ML34:

   movl $ML30, -(%PS)

   movl (%PS)+, 4*1(%PS)

   movl $ML35, -(%PS)

   movl (%PS)+, 4*0(%PS)

   movl $ML40, -(%PS)

   jsb _ML24
   rsb
```

```
 0: DEF_FUN          34

 4: PUSHCONST        30

 8: MOVEP             1

10: PUSHCONST        35

14: MOVEP             0

16: PUSHCONST        40

20: JGLOBFUN         24
```

VAX 11/780 CODE WITH OPTIMIZATION

```
    .globl _Fmain
    _Fmain:
    jsb _ML34
    rsb
    ML0:
    .long 1,0,0
    ML5:
    .long 1,0,0
    ML10:
    .long 3,_ML24,3
    ML15:
    .long 3,_ML32,1
    ML20:
    .long 3,_ML33,2
    ML25:
    .long 3,_ML34,2
    ML30:
    .long 1,10,0
    ML35:
    .long 1,1,0
    ML40:
    .long 1,0,0
    .text
    _ML24:
    movl 4*2(%PS), -(%PS)
    jsb _eval
    addl2 $4, %PS
1   movl 8(%PS), r0
2   movl 4(r0), r1
2   subl3 $0,r1,r1
    jneq _MLL28
1   movl (%PS), -(%PS)
    jsb _eval
    movl 4*4(%PS), r7
    movl (%PS)+, r6
    movl (r6), (r7)
    movl 4(r6), 4(r7)
    movl 8(r6), 8(r7)
    addl2 $8, %PS
    addl2 $4, %PS
    rsb
    _MLL28:
1   movl 8(%PS), r0
2   movl 4(r0), r1
```

```
4   decl r1
    jneq _MLL30
1   movl 4(%PS), -(%PS)
    jsb _eval
    movl 4*4(%PS), r7
    movl (%PS)+, r6
    movl (r6), (r7)
    movl 4(r6), 4(r7)
    movl 8(r6), 8(r7)
    addl2 $8, %PS
    addl2 $4, %PS
    rsb
    _MLL30:
    movl 4*2(%PS), -(%PS)
    movl $ML15, -(%PS)
    movl $APPLY, (%GM)+
    movq (%PS)+, (%GM)+
    moval -12(%GM), -(%PS)
    movl (%PS)+, 4*2(%PS)
1   movl 4(%PS), -(%PS)
    movl 4*2(%PS), -(%PS)
    movl 4*2(%PS), -(%PS)
    movl $ML20, -(%PS)
    movl $APPLY, (%GM)+
    movq (%PS)+, (%GM)+
    moval -12(%GM), -(%PS)
    movl $APPLY, (%GM)+
    movq (%PS)+, (%GM)+
    moval -12(%GM), -(%PS)
    movl (%PS)+, 4*2(%PS)
    movl (%PS)+, (%PS)
    jsb _ML24
    rsb
    .text
    _ML33:
1   movl 4(%PS), -(%PS)
    jsb _eval
    addl2 $4, %PS
    jsb _eval
1   movl (%PS), r0
1   movl 4(%PS), r2
    addl2 $8, %PS
2   movl 4(r0), r1
2   movl 4(r2), r3
2   addl2 r1,r3
    movl (%PS), r6
```

```
    movl $VAL, (r6)
2   movl r3, 4(r6)
    movl $0, 8(r6)
    rsb
    .text
    _ML32:
1   movl (%PS), r0
    addl2 $4, %PS
2   movl 4(r0), r1
4   decl r1
    movl (%PS), r6
    movl $VAL, (r6)
2   movl r1, 4(r6)
    movl $0, 8(r6)
    rsb
    .text
    _ML34:
    movl $ML30, -(%PS)
    movl (%PS)+, 4(%PS)
    movl $ML35, -(%PS)
    movl (%PS)+, (%PS)
    movl $ML40, -(%PS)
    jsb _ML24
    rsb
```

The unoptimized version of the above example for the VAX 11/780, which does simple macro expansion, occupies 17% more space than the optimized version. One of the main differences between these two code generators is usage of registers for the V stack elements. Another optimization that is obtained in this implementation is: whenever the V stack is popped, free the target machine register it occupies. Here the optimized code generator need not emit code, whereas the unoptimized code generator must pop the stack. In the unoptimized version, the V stack was implemented using memory locations, whereas in the optimized version, registers are used. The optimized code generator has used an auto-decrement instruction instead of the two address add instruction.

The time taken by the above programs is not large enough to be significantly compared. Ideally, a comparison could be made using instruction execution times published by the manufacturer, but this is beyond the scope of this thesis. Furthermore, issues such as cache usage may obscure such a comparison.

The following table contains the function name and the number of move instructions that were saved by the optimized code generator. The following table doesn't include the savings obtained because of auto-increment, auto-decrement and usage of registers instead of memory locations.

COL A : Space occupied by the code generated from the optimized code generator

COL B : Space occupied by the code generated from the unoptimized code generator

COL C : Percentage savings in the space occupied

COL D : Number of move instructions executed by the optimized code generator

COL E : Number of move instructions executed by the unoptimized code generator

COL F : Number of move instructions execution saved by the optimized code generator

COL G : Percentage savings in the number of instructions executed

function (args)	space occupied A	B	C	execution statistics D	E	F	G
factorial (20)	867	1095	22%	651	794	143	18%
Linear Fibonacci (100)	1554	1877	17%	5661	6558	897	14%
Strict Fibonacci (100)	1060	1379	23%	2193	3090	897	29%
ackermann (2 6)	1683	2116	20%	4254	5148	894	16%
tak (10 9 8)	2214	2491	11%	195	211	16	8%
towers of hanoi (1 2 3 5)	2363	2682	12%	2380	2672	292	11%
random numbers (50)	2332	2594	10%	6994	7546	552	7%
sort (10 numbers)	3089	3277	6%	2244	2320	76	3%
primes (100)	3204	3512	9%	23598	26512	2914	11%

7. CONCLUSIONS

By using the methods developed in this thesis it is easy to develop and maintain an unoptimized target machine description table for a new machine.

Machine-dependent optimizations have been incorporated by the use of attributes. Using machine-dependent optimizations the code generator produces better code than a compiler that uses simple macro expansion. The results obtained during this implementation indicate that the code that was produced occupies 10-30% less space than a macro expansion version. This optimization has been achieved primarily by avoiding redundant loads and stores into target machine registers, using specialized instructions (e.g. auto-increment/decrement) and not generating code for some instructions altogether. Usage of registers for temporary storage and keeping track of the life of a variable in a block allowed us to maximize the usage of registers. By incrementally adding new attributes the machine description table can be improved to produce even better code.

All the above optimizations and retargetability are obtained in a single-pass code generation scheme.

7.1 Further Research

Instead of freeing all registers at the end of a basic block, the code generator could allocate and save registers based on global flow analysis of variables that are live or dead across blocks (i.e. across modules). The garbage collector library (written in G-code) has to be ported. Attributes to do run-time optimizations could be added.

References

[Aho and Johnson 76]

A.V. Aho and S.C. Johnson, "Optimal code generation for expression tress", J.ACM, 23, 3, 1976, 488-501.

[Aho, Johnson and Ullman 77a]

A.V. Aho, S.C. Johnson and J.D. Ullman, "Code generation for expression with common subexpressions", J.ACM, 24, 1, 1977, 146-160.

[Aho, Johnson and Ullman 77b]

A.V. Aho, S.C. Johnson and J.D. Ullman, "Code generation for machines with multiregister operations", Fourth ACM Symposium on Principles of Programming languages, 1977, 21-28.

[Aho 77]

A.V. Aho and J.D. Ullman, "Principles of Compiler Design", Addison-Wesley publishing Co., 1977.

[Aho and Ganapathi 85]

A.V. Aho and Mahadevan Ganapathi, "Efficient Tree Pattern Matching: an Aid to Code Generation", Communications of the ACM, Jan., 1985, 334-339.

[Bruno and Sethi 76]

J.Bruno and R.Sethi, "Code generation for a one-register machine", J.ACM, 23, 3, 1976, 502-510.

[Cattell 78]

R.G.G. Cattell, "Formalization and Automatic Derivation of Code Generators", Phd thesis, Carnegie Mellon University 1978.

[Cattell 79]

R.G.G. Cattell, J.M. Newcomer and B.W. Leverett, "Code Generation in a Machine-Independent Compiler", ACM Sigplan Symp. on Compiler Construction, Boulder, Colo., Aug. 1979.

[Cattell 80]

R.G.G. Cattell, "Automatic Derivation of Code Generators from Machine Descriptions", ACM Trans. on Programming Languages and Systems, Vol. 2 No. 2 pp. 173-190, April 1980.

[Donegan 73]

M.K. Donegan, "An Approach to the Automatic Generation of Code Generators", Phd thesis, Rice University, Houston, Texas, 1973.

[Donegan 79]

M.K. Donegan et al., "A Code Generator Language", ACM Sigplan Symp. on Compiler Construction, Boulder, Colo., Aug. 1979.

[Elson 70]

M. Elson and S.T. Rake, "Code Generation Technique for Large Language Compilers", I.B.M Systems Journal Vol. 9 No. 3 pp. 166-188, 1970.

[Fraser 77]

C.W. Fraser, "Automatic Generation of Code Generators", Phd thesis, Yale University, New Haven, Conn., 1977.

[Fraser 79]

C.W. Fraser, "A Compact Machine Independent Peephole Optimizer", Principles of Programming Languages, 1979.

[Fraser 80]

C.W. Fraser and J.W. Davidson, "The Design and Application of a retargetable Peephole Optimizer", ACM Trans. on Programming Languages and Systems, Vol. 2 No. 2 pp. 173-190, April 1980.

[GAH 86]

G-Machine Architecture/Programmers Handbook, Oregon Graduate Center, February 1986. Unpublished document.

[Ganapathi 80]

M. Ganapathi, "Retargetable Code Generation and Optimization using Attribute Grammars", Phd thesis, University of Wisconsin, Madison, 1980.

[Ganapathi 82]

M. Ganapathi and C.N. Fischer, "Description-Driven Code Generation using Attribute Grammars", Communications of the ACM January, 1982, pp. 108-117.

[Ganapathi 84]

M. Ganapathi and C.N. Fischer, "Attributed linear intermediate representation for retargetable code generators" Software Practice and experience, Vol. 14, No. 4, April 1984, pp. 347-364.

[Glanville 77]

R.S. Glanville , "A Machine Independent Algorithm for Code Generation and its Use in Retargetable Compilers", Phd thesis, University of California, Berkeley, Dec. 1977.

[Glanville 78]

R.S. Glanville and S.L. Graham, "A New Method for Compiler Code Generation", Conf. Record Fifth ACM Symp. Principles of Programming Languages, Jan. 1978.

[Graham 80]

S.L. Graham, "Table-Driven Code Generation", IEEE Computer, Vol. 13 No. 8 pp. 25-34, Aug. 1980.

[Johnson 75]

S.C. Johnson, "YACC - Yet Another Compiler Compiler", C.S. Tech Report# 32, Bell Telephone Laboratories, Murray Hill, New Jersey, 1975.

[Johnson 78]

S.C. Johnson, "A Portable Compiler: Theory and Practice", Proc. 5th ACM Symp. Principles of Programming Languages, pp. 97-104, Jan. 1978.

[Johnson, T. 84]

T. Johnson, "Efficient compilation of lazy evaluation", Proc. 1984 ACM SIGPLAN conf. on compiler construction., June, 1984.

[Kessler 86]

R. R. Kessler etc., "EPIC - A retargetable, highly optimizing LISP compiler, Proc. of SIGPLAN 86 Symposium of compiler construction Vol 21, No.7, July 1986 pp. 118-130.

[Kieburtz 85]

R. B. Kieburtz, "The G-machine: A fast-graph-reduction evaluator", Tech. Rep. CS/E-85-002, Oregon Graduate Center, Beaverton, OR, January, 1985.

[Kieburtz 86]

R. B. Kieburtz, "Incremental collection of dynamic, list-structure memories", Tech. Rep. CS/E-85-008, Oregon Graduate Center, Beaverton, OR, January, 1986.

[Knuth 68]

D. E. Knuth, "Semantics of Context-free Languages", Math. Systems Theory, Vol. 2 No. 2 pp. 127-145, June 1968.

[Lesk 79]

M.E. Lesk, "Lex - A Lexical Analyzer Generator", UNIX Programmer's Manual 2, Section 20, 1979.

[Ripken 77]

K. ripken, "Formale Bescheribun von Maschinen, Implementierungen und Optimierender Maschinen-codeerzeugung aus Attributierten Programmgraphe", Technische Univer, Munchen, Munich, Germany, July 1977.

[Ritchie 78]

D.M. Ritche and B.W. Kernighan, "The C Programming Language", Prentice-Hall, Englewood Cliffs, New Jersey, 1978.

[Sethi and Ullman 70]

R.Sethi and J.D. Ullman, "The generation of optimal code for arithmetic expressions", J.ACM, 17, 4, 1970, 715-728.

[Steel 61]

T.B. Steel, Jr., "A First Version of UNCOL", Proceedings WJCC, 19, pp. 371-378, 1961.

[Strong 58]

J. Strong et al., "The Problem of Programming Communication With Changing Machines: A Proposed Solution", CACM Vol.1 No. 8, pp. 12-18, 1958.

[Wilcox 71]

T.R. Wilcox, "Generating Machine Code for High Level Programming Languages", Tech. Report 71-103, Phd thesis, Dept. of Computer Sciences, Cornell University, 1971.

[Wulf 80a]

W. Wulf et al., "TCOLAda: Revised Report on An Intermediate Representation for the Preliminary Ada Language", Tech. Report CMU-CS-80-105, Dept. of Computer Sciences, Carnegie-Mellon University, Feb. 1979.

[Wulf 80b]

W. Wulf et al., "An Overview of the Production-Quality Compiler-Compiler Project", IEEE Computer Vol. 13 No. 8 pp. 38-49, Aug. 1980.

APPENDIX A: G-CODE

G-machine instructions are defined in terms of transformations on an abstract register model. Components of this model are:

<C,P,V,G>

C -- Control sequence
P -- Pointer stack
V -- Value stack
G -- expression Graph

The control, C, represents the dynamic instruction sequence. In a hardware implementation, C might be realized by the program counter of a von Neumann machine.

The pointer stack, P, holds pointers to available components (subexpressions) throughout traversal and reduction of expression graph.

The value stack, V, holds intermediate values of a basic type (integer, boolean, character) during expression evaluation.

The expression graph, G, is the image of in G-memory of an expression under evaluation. For a detailed explanation please refer to [Kieburtz 85].

G-CODE	OP	DESC
ALLOC	40	allocate a node without a value <ALLOC -.C,P,V,G> ==> <C,n.P,V,[n:(-,-)]+G>
binops		
ADD	08	add integers V[0] = V[0] + V[1]
SUB	10	subtract integers
MUL	12	product of integers
DIV	13	quotient of integers
AND	18	logical and
OR	19	logical or <opc j.C,P,i_0...i_j.V,G> ==> <C,P,opc(i_j,i_0)..i_j.V,G> where opc is one of ADD,SUB,MUL,DIV,AND,OR
CALLGLOBFUN	228	call the function whose address is given as an argument
COPYP	112	copy indexed cell of P-stack to top of stack <COPYP m.C,n_0...n_m.P,V,G> ==> <C,n_m.n_0...n_m.P,V,G>
COPYV	120	copy indexed cell of V-stack to top of stack <COPYV m.C,P,v_0...v_m.V,G> ==> <C,P,v_m.v_0...v_m.V,G>
DECR	25	decrement value at top of V-stack <DECR -.C,P,i.V,G> ==> <C,P,(i-1).V,G>
DEF_FUN	254	start of a function
END_FUN	255	end of a function
EVAL	94	evaluate a function to normal form <EVAL j.C,n_0...n_j.P,V,[n_j:@(p_1,p_2)]+G> ==> <C,n.P,V,[n_j:v]+G>
FUN	402	initialize the heap with a function descriptor
GET_BYTE	132	get literal value of argument into V-stack <GET_BYTE b.C,P,V,G> ==> <C,P,b.V,G>
GET_FST	58	first of a pair(value is basic type) into V-stack <GET_FST j.C,n_0...n_j.P,V,[n_j:@(v_1,n_2)]+G> ==> <C,n_0...n_j.P,v_1.V,[n:@(v_1,n_2)]+G>
GET_SND	58	second of a pair(value is basic type) into V-stack <GET_SND j.C,n_0...n_j.P,V,[n_j:@(n_1,v_2)]+G> ==> <C,n_0...n_j.P,v_2.V,[n:@(n_1,v_2)]+G>
INCR	24	increment value at top of V-stack <INCR -.C,P,i.V,G> ==> <C,P,(i+1).V,G>
INT	401	initialize the heap with an integer value
JMP	208	an unconditional jump to argument label
JFUN	208	jump to the argument label
JNOT_NEG	209	jump to label on non-negative value
JNEG	210	jump to label on negative value
JNOT_ZERO	211	jump to label on non-zero value
JZERO	212	jump to label on zero value

JGLOBFUN	215	jump to global function
LABEL		a symbolic label
MK_APP	41	make an application node <MK_APP -.C,n_0...n_1.P,V,G> ==> <C,n.P,V,[n:@(n_0,n_1)]+G>
MK_PR	42	make a constructed pair <MK_PR j.C,n_0.n_1.P,V,G> ==> <C,n.P,V,[n:@(n_0,n_1)]+G>
MK_VAL	43	make a basic node form the value at V[0] <MK_VAL -.C,P,v.V,G> ==> <C,n.P,V,[n:(v,0)]+G>
MOVEP	113	move a pointer from P[0] to cell indexed argument + 1 <MOVEP m.C,n_0...n_m.n_{m+1}.P,V,G> ==> <C,n_1...n_m.n_0.P,V,G>
MOVEV	121	move a value from V[0] to cell indexed argument + 1 <MOVEV m.C,P,v_0...v_m.v_{m+1}.V,G> ==> <C,P,v_1...v_m.v_0.V,G>
NEG	27	negate integer value at top of V-stack <NOT -.C,P,i.V,G> ==> <C,P,(-i).V,G>
NOT	28	bitwise(one's) complement of top of V-stack <NOT -.C,P,b.V,G> ==> <C,P,(1's complement b).V,G>
POPP	48	pop P-stack <POPP m.C,n_0.P,V,G> ==> <C,P,V,G>
POPV	56	pop V-stack <POPV m.C,P,v_0.V,G> ==> <C,P,V,G>
POP2	45	pop P-stack twice
POP4	46	pop P-stack four times
POP8	47	pop P-stack eight times
PUSH_LIT	190	push literal value of argument into V-stack <PUSH_LIT i.C,P,V,G> ==> <C,P,i.V,G>
PUSHCONST	182	push pointer to global constant onto P-stack <PUSHCONST addr.C,P,V,G> ==> <C,addr.P,V,G>
FST	51	first element of a pair (non-basic type) <FST -.C,n.P,V,[n:(v_1,n_2)]+G> ==> <C,n_1.P,V,[n:@(v_1,n_2)]+G>
RET	104	return from function call
RET_INT	105	update(arg) with basic value from V[0];RET
SND	52	second element of a pair (non-basic type) <FST -.C,n.P,V,[n:(v_1,n_2)]+G> ==> <C,n_2.P,V,[n:@(v_1,n_2)]+G>
UPDATE	96	update cell pointed to by indexed element of P-stack

APPENDIX B: PREFIX AND LIBRARIES

The user defines the prefix to the code generator which contains certain standard register names (e.g the P stack is denoted as PS). An example of prefix for VAX 11/780 is attached below.

PREFIX FILE

```
        .data
        .text

#                               Executable
        .org 65535

#                               Register names
        .set PS, 11
        .set GM, 10
        .set hp, 09
        .set VS, 08

#                               Allocate memory size
        .set HPMEMORYSIZE, 10000
        .set APMEMORYSIZE, 10000
        .set SPMEMORYSIZE, 10000

        .set GMMEMORYSIZE, 10000
        .set PSMEMORYSIZE, 10000
        .set VSMEMORYSIZE, 10000

#                               G memory node types
        .set NIL,    0
        .set VAL,    1
        .set APPLY,  2
        .set FUN,    3
        .set PAIR,   4
        .set INJECT,  5
        .set printcnt, 0
```

```
#                              Machine required information
        .data
        .text
        .align 1

        GL6:
        .long 3, _hd,1
        GL7:
        .long 3, _tl,1
        GL8:
        .long 3, _null,1
        GL9:
        .long 3, _fst,1
        GL10:
        .long 3, _snd,1
        GL21:
        .long 3, _fail,1
        GINP:
        .long 3, _Finput,1

#                              MAIN
        .globl _main
        .globl collect_gar
        .globl err1
        .globl end1

        _main:
        .word 0x0

                pushl       $HPMEMORYSIZE
                calls $1,_malloc
                movl  r0, %hp
                addl2    $HPMEMORYSIZE, %hp

                pushl       $GMMEMORYSIZE
                calls $1,_malloc
                movl  r0, %GM

                pushl       $PSMEMORYSIZE
                calls $1,_malloc
                movl  r0, %PS
                addl2    $PSMEMORYSIZE, %PS

                pushl       $VSMEMORYSIZE
                calls $1,_malloc
                movl  r0, %VS
```

```
        addl2   $VSMEMORYSIZE, %VS

        movl $APPLY, (%GM)+
        addl2 $8, %GM
        moval -12(%GM), -(%PS)
        movl $APPLY, (%GM)+
        addl2 $8, %GM
        moval -12(%GM), -(%PS)
        movl $APPLY, (%GM)+
        movl $GINP, (%GM)+
        movl $NIL, (%GM)+
        moval -12(%GM), -(%PS)

        moval PR , -(sp)

PR:
.data 1
PR1:
.ascii    "12Result is : %d12 "
INP:
.ascii      "%d "
.text

        jsb _Fmain
        jmp _lprint
```

SUFFIX FILE

```
.globl _lprint
_lprint:
cmpl $VAL, (r0)
jneq _skip2
movl (%PS)+, r1
movl 4(r1), r0
pushl r0
pushl $PR1
calls $2,_printf
jmp end1
_skip0:
movl (%PS), r0
cmpl $NIL, (r0)
jeql end1
cmpl $APPLY, (r0)
jneq _skip1
jsb _eval
_skip1:
cmpl $VAL, (r0)
jneq _skip2
movl (%PS)+, r1
movl 4(r1), r0
pushl r0
pushl $PR1
calls $2,_printf
decl %printcnt
cmpl $0, %printcnt
bleq end1
jmp _skip3
_skip2:
movl (%PS)+, r0
movl 8(r0), -(%PS)
movl 4(r0), -(%PS)
addl2 $2, %printcnt
_skip3:
cmpl $0, %printcnt
jeql end1
jmp _skip0

.text
.globl _Finput
_Finput:
```

```
movl $0, (%GM)+
addl2  $8, %GM
moval -12(%GM), -(%PS)
movl 8(%PS), -(%PS)
movl (%PS)+, r6
movl (%PS), r7
movl (r6), (r7)
movl 4(r6), 4(r7)
movl 8(r6), 8(r7)
movl $VAL, (%GM)+
movl %GM, r0
pushl r0
pushl $INP
calls $2,_scanf
addl2 $4, %GM
movl $0, (%GM)+
moval -12(%GM), -(%PS)
movl 12(%PS), r7
movl $PAIR, (r7)
movl (%PS)+, r6
movl r6, 4(r7)
movl (%PS)+, r6
movl r6, 8(r7)
addl2 $4, %PS
rsb


#                              Global collect garbage
collect_gar:

#                              Error function
err1:

#                              End of program
_MLL253:

end1:
calls $0, _exit
```

EVAL

```
#                                   Misc variables
.set argcount, 2

.data
.text
.globl _eval
_eval:
_localeval:
movl (%PS), r0
cmpl $APPLY, (r0)
jneq _evalexit

movl (%PS), r0
movl %PS,  -(%hp)                   # save P stack

movl $0, %argcount

_unwind:
addl2 $1, %argcount
movl 8(r0), -(%PS)
movl 4(r0), r0
cmpl $APPLY, (r0)
jeql _unwind

movl 8(r0), r1
cmpl %argcount, r1
jlss _notenough
jsb *4(r0)
movl (%PS), r0
cmpl $APPLY, (r0)
jneq _evalexit

jmp _localeval

_notenough:
movl (%hp)+, %PS

_evalexit:
rsb

err1:

#                                   End of program
```

```
endl:
calls $0, _exit
```

The shell script to create the archive is given below.

```
#! /bin/csh -f
# assembles and appends file to archive file in a
# verbose manner
# and insures random access with ranlib
as -J $argv[1] -o $argv[1].o
ar qv archive $argv[1].o
ranlib archive
```

The shell script to assemble is given below.

```
#! /bin/csh -f
# assembles and loads file
as -J $argv[1] -o $argv[1].o

#  only load the archive file if it exits
if (-e archive) then
ld -X /lib/crt0.o $argv[1].o -o $argv[1].out -lc archive
else
ld -X /lib/crt0.o $argv[1].o -o $argv[1].out -lc
endif

#       exec and clean up
$argv[1].out | more
rm $argv[1].out
rm $argv[1].o
```

APPENDIX C: MACHINE DESCRIPTION TABLE FOR VAX 11/780

```
GCODE      ATTRIBUTES          ACTION CODES

ADD        'op1_lit_0'
                           :   V[0]=V[1]; ;

ADD        'op2_lit_0'     :   ;;

ADD        'op1_lit_1'     :
                               "incl" V[1];
                               V[0]=V[1]; ;

ADD        'op2_lit_1'     :
                               "incl" V[0]; ;

ADD                        :
                               $N = ALLOC_SUDO;
                               "add12" V[0] V[1] $N;
                               V[0] = $N; ;

MOD
                           :   $N = ALLOC_SUDO;
                               "mod13" V[0] V[1] $N;
                               V[0] = $N;;

NEG                        :
                               $N = ALLOC_SUDO;
                               "mneg1" V[0] $N;
                               V[0] = $N;;
```

```
GCODE     ATTRIBUTES          ACTION CODES

NOT       'op1_lit_0'     :
                              "incl" V[0];;

NOT       'op1_lit_1'     :
                              "clrl" V[0];;

NOT                       :
                              $N = ALLOC_SUDO;
                              "subl2" V[0] V[1] $N;
                              V[0] = $N;;

AND                       :
                              $N = ALLOC_SUDO;
                              "bitl3" V[0] V[1] $N;
                              V[0] = $N;;

OR                        :
                              $N = ALLOC_SUDO;
                              "bisl3" V[0] V[1] $N;
                              V[0] = $N;;

INCR                      :
                              "incl" V[0];;

DECR                      :
                              "decl" V[0];;

GET_FST   'is_in_reg'     :
                              $R = P[op1];
                              $N = ALLOC_SUDO ;
                              $N.reg = ALLOC_REG ;
                              PUSH_V $N;
                              "movl" 4 $R.reg $N.reg;;

GET_FST                   :
                              $R = P[op1];
                              $R.reg = GET_IN_REG $R;
                              $N = ALLOC_SUDO ;
                              $N.reg = ALLOC_REG ;
                              PUSH_V $N;
                              "movl" 4 $R.reg $N.reg ;;
```

```
GCODE   ATTRIBUTES          ACTION CODES

GET_SND  'is_in_reg'   :
                            $R = P[op1];
                            $N = ALLOC_SUDO ;
                            $N.reg = ALLOC_REG ;
                            PUSH_V $N;
                            "movl" 8 $R.reg $N.reg;;

GET_SND                :
                            $R = P[op1];
                            $R.reg = GET_IN_REG $R;
                            $N = ALLOC_SUDO ;
                            $N.reg = ALLOC_REG ;
                            PUSH_V $N;
                            "movl" 8 $R.reg $N.reg ;;

GET_BYTE               :
                            $N = ALLOC_SUDO;
                            $N.cnst = op1;
                            PUSH_V $N;;

FST                    :
                            $N = ALLOC_SUDO ;
                            PUSH_P $N;
                            P[1] = P[0];
                            POP_P;
                            EMIT "movl (%PS)+, r5";
                            EMIT "movl 4(r5), -(%PS)";;

SND                    :
                            $N = ALLOC_SUDO ;
                            PUSH_P $N;
                            P[1] = P[0];
                            POP_P;
                            EMIT "movl (%PS)+, r5";
                            EMIT "movl 8(r5), -(%PS)";;
```

```
GCODE     ATTRIBUTES          ACTION CODES

POPP                      :
                            EMIT "addl2 $4, %PS";
                            POP_P;;

POPV                      :
                            FREE_REG V[0];
                            POP_V;;

COPYP     'arg_0'         :
                            PUSH_P P[op1];
                            EMIT
                            "movl (%PS), -(%PS)";;

COPYP     'arg_1'         :
                            PUSH_P P[op1];
                            EMIT
                            "movl 4(%PS), -(%PS)";;

COPYP                     :
                            PUSH_P P[op1];
                            EMIT "movl 4*" op1
                                 "(%PS), -(%PS)";;

COPYV                     :
                            PUSH_V V[op1];;
```

```
GCODE     ATTRIBUTES          ACTION CODES

MOVEP     'arg_0'        :
                              EMIT
                              "movl (%PS)+, (%PS)";
                              P[op1 + 1] = P[0];
                              POP_P; ;

MOVEP     'arg_1'        :
                              EMIT
                              "movl (%PS)+, 4(%PS)";
                              P[op1 + 1] = P[0];
                              POP_P; ;

MOVEP                    :
                              EMIT "movl (%PS)+, 4*"
                                    op1 "(%PS)";
                              P[op1 + 1] = P[0];
                              POP_P; ;

MOVEV                    :
                              FREE_REG V[0];
                              V[1] = V[0];
                              POP_V;;

ALLOC                    :
                              $N = ALLOC_SUDO;
                              PUSH_P $N;
                              EMIT
                             "movl $" op1 ", (%GM)+";
                              EMIT "addl2  $8, %GM";
                              EMIT
                             "moval -12(%GM), -(%PS)";;
```

```
GCODE       ATTRIBUTES          ACTION CODES

MK_APP                      :
                                POP_P 2;
                                $N = ALLOC_SUDO;
                                PUSH_P $N;
                                EMIT
                                "movl $APPLY, (%GM)+";
                                EMIT
                                "movq (%PS)+, (%GM)+";
                                EMIT
                                "moval -12(%GM), -(%PS)";;
MK_VAL                      :
                                EMIT V;
                                EMIT
                                "movl $VAL, (%GM)+";
                                EMIT
                                "movl (%VS)+, (%GM)+";
                                EMIT "movl $0, (%GM)+";
                                EMIT
                                "moval -12(%GM), -(%PS)";
                                FREE_REG V[0];
                                POP_V;
                                PUSH_P V[0];;
MK_PR                       :
                                POP_P 2;
                                $N = ALLOC_SUDO;
                                PUSH_P $N;
                                EMIT
                                "movl $PAIR, (%GM)+";
                                EMIT
                                "movq (%PS)+, (%GM)+";
                                EMIT
                                "moval -12(%GM), -(%PS)";;
```

```
GCODE      ATTRIBUTES          ACTION CODES

PUSH_LIT                  :
                              $N = ALLOC_SUDO;
                              $N.cnst = op1;
                              PUSH_V $N;;

PUSHCONST                 :
                              $N = ALLOC_SUDO ;
                              PUSH_P $N;
                              EMIT
                             "movl $ML" op1 ", -(%PS)";;

PUSHGLOBAL                :
                              $N = ALLOC_SUDO ;
                              PUSH_P $N;
                              EMIT "movl $3, (%GM)+";
                              EMIT
                              "movl $_ML" op1 ", (%GM)+";
                              EMIT "movl $1, (%GM)+";
                              EMIT
                              "moval -12(%GM), -(%PS)";;
CALLGLOBFUN
            'if_import'   :   EMIT
                              "movl $GL" op1 ", r6";
                              EMIT "jsb *4(r6)";;
CALLGLOBFUN               :
                              EMIT "jsb _ML" op1 ;;

DEF_FUN                   :
                              EMIT ".text" ;
                              EMIT ".globl _ML" op1;
                              EMIT "_ML" op1 ":" ; ;
```

```
GCODE      ATTRIBUTES          ACTION CODES

JFUN                      :
                             EMIT "jmp _MLL" op1 ;;

JNOT_NEG                  :
                             EMIT V;
                             EMIT "jleq _MLL" op1 ;;

JNEG                      :
                             EMIT V;
                             EMIT "jleq _MLL" op1 ;;

JNOT_ZERO                 :
                             EMIT V;
                             EMIT "jneq _MLL" op1 ;;

JZERO                     :
                             EMIT V;
                             EMIT "jeql _MLL" op1 ;;

JGLOBFUN
        'if_import'       :  EMIT
                             "movl $GL" op1 ", r6";
                             EMIT "jsb *4(r6)";;
JGLOBFUN                  :
                             EMIT "jmp _ML" op1 ;;

J_NOT_PTR
                          :  EMIT
                            "subl3 $APPLY, (r5), r6";
                             EMIT "jlss _MLL" op1;;
J_IF_PTR
                          :  EMIT
                            "subl3 $APPLY, (r5), r6";
                             EMIT "jgeq _MLL" op1;;
```

```
GCODE     ATTRIBUTES          ACTION CODES

UPDATE    'arg_1'        :
                              P[op1] = P[0];
                              POP_P;
                              EMIT "movl (%PS)+, r6";
                              EMIT "movl (%PS), r7";
                              EMIT "movl (r6), (r7)";
                              EMIT
                              "movl 4(r6), 4(r7)";;
UPDATE                   :
                              P[op1] = P[0];
                              POP_P;
                              EMIT "movl 4*" op1
                                   "(%PS), r7";
                              EMIT "movl (%PS)+, r6";
                              EMIT "movl (r6), (r7)";
                              EMIT
                              "movl 4(r6), 4(r7)";;

RET                      :
                              EMIT "rsb";;

RET_INT                  :
                              EMIT V;
                              EMIT "movl (%PS), r6";
                              EMIT "movl $VAL, (r6)";
                              EMIT
                              "movl (%VS)+, 4(r6)";
                              EMIT "rsb";
                              FREE_REG V[0];
                              POP_V;;

EVAL                     :
                              EMIT "jsb _eval";;

FUN                      :
                              PUSH_P op1;
                              EMIT "ML" d_label ":" ;
                              EMIT ".long 3, _ML"
                                   op2 "," op1;;

INT                      :
                              EMIT "ML" d_label ":" ;
                              EMIT ".long 1,"
```

```
                                op1 ",0";;

LABEL                   :
                                EMIT "_MLL" op1 ":" ; ;
EXPORT
                        :       EMIT ".globl _" op1 ;
                                EMIT "_" op1 ":" ;
                                EMIT "jsb _ML" op2;
                                EMIT "rsb";;
IMPORT
                        :       EMIT "GL" op2 ":";
                                EMIT ".long 3, _"
                                     op1 ",1" ;;
NOP                     :       ;;
END_FUN                 :       ;;
```

APPENDIX D MACRO EXPANSION TABLE FOR VAX 11/780

```
GCODE                ACTION CODES


ADD
          :     EMIT "addl2 4(%VS), (%VS)";;
SUB
          :     EMIT "subl3 (%VS), 4(%VS), (%VS)";;
MUL
          :     EMIT "mull2 4(%VS), (%VS)";;
DIV
          :     EMIT "divl3 (%VS), 4(%VS), (%VS)";;
MOD
          :     EMIT "divl3 (%VS), 4(%VS), r0";
                EMIT "mull2 (%VS), r0";
                EMIT "subl3 r0, 4(%VS), (%VS)";;
NEG
          :     EMIT "mnegl (%VS)";;
NOT
          :     EMIT "subl3 (%VS), $1, (%VS)";;
AND
          :     EMIT "bitl2 4(%VS), (%VS)";;
OR
          :     EMIT "bisl2 4(%VS), (%VS)";;
INCR
          :     EMIT "incl (%VS)";;
DECR
          :     EMIT "decl (%VS)";;
GET_FST
          :     EMIT "movl 4*" op1 "(%PS), r0";
                EMIT "movl 4(r0), -(%VS)";;
GET_SND
          :     EMIT "movl 4*" op1 "(%PS), r0";
                EMIT "movl 8(r0), -(%VS)";;
```

```
GCODE                ACTION CODES


FST
          :     EMIT "movl (%PS)+, r0";
                EMIT "movl 4(r0), -(%PS)";;
SND
          :     EMIT "movl (%PS)+, r0";
                EMIT "movl 8(r0), -(%PS)";;
GET_BYTE
          :     EMIT "movl $" op1 ", -(%VS)";;
POPP
          :     EMIT "addl2 $4, %PS";;
POP2
          :     EMIT "addl2 $8, %PS";;
POP4
          :     EMIT "addl2 $16, %PS";;
POP8
          :     EMIT "addl2 $32, %PS";;
POPV
          :     EMIT "addl2 $4, %VS";;
COPYP
          :     EMIT "movl 4*" op1 "(%PS), -(%PS)";;
COPYV
          :     EMIT "movl 4*" op1 "(%VS), -(%VS)";;
MOVEP
          :     EMIT "movl (%PS)+, 4*" op1 "(%PS)";;
MOVEV
          :     EMIT "movl (%VS)+, 4*" op1 "(%VS)";;
ALLOC
          :     EMIT "movl $" op1 ", (%GM)+";
                EMIT "addl2  $8, %GM";
                EMIT "moval -12(%GM), -(%PS)";;
MK_APP
          :     EMIT "movl $APPLY, (%GM)+";
                EMIT "movq (%PS)+, (%GM)+";
                EMIT "moval -12(%GM), -(%PS)";;
MK_VAL
          :     EMIT "movl $VAL, (%GM)+";
                EMIT "movl (%VS)+, (%GM)+";
                EMIT "movl $0, (%GM)+";
                EMIT "moval -12(%GM), -(%PS)";;
MK_PR
          :     EMIT "movl $PAIR, (%GM)+";
                EMIT "movq (%PS)+, (%GM)+";
                EMIT "moval -12(%GM), -(%PS)";;
```

```
GCODE                ACTION CODES


PUSH_LIT
          :     EMIT "movl $" op1 ", -(%VS)";;

PUSHCONST
          :     EMIT "movl $ML"  op1 ", -(%PS)" ;;

CALLGLOBFUN
          :     EMIT "jsb _ML" op1 ;;

DEF_FUN
          :     EMIT ".text";
                EMIT "_ML" op1 ":" ; ;
JFUN
          :     EMIT "jmp _MLL" op1 ;;
JMP
          :     EMIT "jmp _MLL" op1 ;;

JNOT_NEG
          :     EMIT "jeq _MLL" op1 ;;

JNEG
          :     EMIT "jleq _MLL" op1 ;;

JNOT_ZERO
          :     EMIT "jneq _MLL" op1 ;;
JZERO
          :     EMIT "jeql _MLL" op1 ;;

JGLOBFUN
          :     EMIT "jsb _ML" op1 ;
                EMIT "rsb";;

J_NOT_PTR
          :     EMIT "subl3 $APPLY, (r0), r6";
                EMIT "jlss _MLL" op1;;

J_IF_PTR
          :     EMIT "subl3 $APPLY, (r0), r6";
                EMIT "jgeq _MLL" op1;;
```

```
GCODE                ACTION CODES


UPDATE
          :     EMIT "movl 4*" op1 "(%PS), r1";
                EMIT "movl (%PS)+, r0";
                EMIT "movl (r0), (r1)";
                EMIT "movl 4(r0), 4(r1)";
                EMIT "movl 8(r0), 8(r1)"; ;

UPDATE_PR
          :     EMIT "movl 4*" op1 "(%PS), r1";
                EMIT "movl $PAIR, (r1)";
                EMIT "movl (%PS)+, r0";
                EMIT "movl r0, 4(r1)";
                EMIT "movl (%PS)+, r0";
                EMIT "movl r0, 8(r1)";;
RET
          :     EMIT "rsb";;
RET_INT
          :     EMIT "movl 4*" op1 "(%PS), r1";
                EMIT "movl $VAL, (r1)";
                EMIT "movl (%VS)+, 4(r1)";
                EMIT "rsb";;
EVAL
          :     EMIT "jsb _eval";;
FUN
          :     EMIT "ML" d_label ":" ;
                EMIT ".long 3,_ML" op2 "," op1 ;;
INT
          :     EMIT "ML" d_label ":" ;
                EMIT ".long 1," op1 ",0";;
LABEL
          :     EMIT "_MLL" op1 ":" ; ;
EXPORT
          :     EMIT ".globl _" op1 ;
                EMIT "_" op1 ":" ;
                EMIT "jsb _ML" op2;
                EMIT "rsb";;
IMPORT
          :     EMIT "GL" op2 ":";
                EMIT ".long 3, _" op1 ",1" ;;
```

APPENDIX E MACHINE DESCRIPTION TABLE FOR SUN WORKSTATION

```
GCODE       ATTRIBUTES          ACTION CODES

ADD
            'op1_lit_0'     :   V[0]=V[1]; ;
ADD
            'op2_lit_0'     :   ;;
ADD
                            :   $N = ALLOC_SUDO;
                                "addl" V[0] V[1] $N;
                                V[0] = $N; ;
NEG
                            :   $N = ALLOC_SUDO;
                                "negl" V[0] $N;
                                V[0] = $N;;
NOT
            'op1_lit_1'     :   "clrl" V[0];;
NOT
                            :   $N = ALLOC_SUDO;
                                "subl" V[0] V[1] $N;
                                V[0] = $N;;
AND
                            :   $N = ALLOC_SUDO;
                                "andl" V[0] V[1] $N;
                                V[0] = $N;;
OR
                            :   $N = ALLOC_SUDO;
                                "orl" V[0] V[1] $N;
                                V[0] = $N;;
INCR
                            :   "addl #1, " V[0];;
DECR
                            :   "subl #1, " V[0];;
```

```
GCODE      ATTRIBUTES          ACTION CODES

FST
                           :   $N = ALLOC_SUDO ;
                               PUSH_P $N;
                               P[1] = P[0];
                               POP_P;
                               EMIT "movl PS@+, a5";
                               EMIT "movl a5@(4), PS@-";;
SND
                           :   $N = ALLOC_SUDO ;
                               PUSH_P $N;
                               P[1] = P[0];
                               POP_P;
                               EMIT "movl PS@+, a5";
                               EMIT "movl a5@(8), PS@-";;
POPP
                           :   EMIT "addl #4, PS";
                               POP_P;;
POPV
                           :   POP_V;;

COPYP      'arg_0'         :   PUSH_P P[op1];
                               EMIT "movl PS@, PS@-";;

COPYP      'arg_1'         :   PUSH_P P[op1];
                               EMIT "movl PS@(4), PS@-";;
COPYP
                           :   PUSH_P P[op1];
                               EMIT "movl PS@(4*"
                                    op1 "), PS@-";;
COPYV
                           :   PUSH_V V[op1];;

MOVEP      'arg_0'         :   EMIT "movl PS@+, PS@";
                               P[op1 + 1] = P[0];
                               POP_P; ;

MOVEP      'arg_1'         :   EMIT "movl PS@+, PS@(4)";
                               P[op1 + 1] = P[0];
                               POP_P; ;
```

```
GCODE      ATTRIBUTES          ACTION CODES

MOVEP                     :    EMIT "movl PS@+, PS@(4*"
                                    op1 ")";
                               P[op1 + 1] = P[0];
                               POP_P; ;
MOVEV
                          :    V[op1 + 1] = V[0];
                               POP_V;;
ROTP
                          :    EMIT "movl PS@+, a0";
                               EMIT "movl PS@+, a5";
                               EMIT "movl a0, PS@-";
                               EMIT "movl a5, PS@-";
                               $N = ALLOC_SUDO;
                               PUSH_P $N;
                               P[0] = P[2];
                               P[2] = P[1];
                               P[1] = P[0];
                               POP_P ; ;
ALLOC
                          :    $N = ALLOC_SUDO;
                               PUSH_P $N;
                               EMIT "movl GM, PS@-";
                               EMIT "movl #" op1 ", GM@+";
                               EMIT "movl #0, GM@+";
                               EMIT "movl #0, GM@+";;
MK_APP
                          :    POP_P 2;
                               $N = ALLOC_SUDO;
                               PUSH_P $N;
                               EMIT "movl #APPLY, GM@+";
                               EMIT "movl PS@+, GM@+";
                               EMIT "movl PS@+, GM@+";
                               EMIT "movl GM, PS@-";
                               EMIT "subl #12, PS@";;
```

```
GCODE     ATTRIBUTES          ACTION CODES

MK_VAL
                          :     EMIT "movl #VAL, GM@+";
                                EMIT V;
                                EMIT "movl VS@+, GM@+";
                                EMIT "movl #0, GM@+";
                                EMIT "movl GM, PS@-";
                                EMIT "subl #12, PS@";
                                PUSH_P V[0];;
MK_PR
                          :     POP_P 2;
                                $N = ALLOC_SUDO;
                                PUSH_P $N;
                                EMIT "movl #PAIR, GM@+";
                                EMIT "movl PS@+, GM@+";
                                EMIT "movl PS@+, GM@+";
                                EMIT "movl GM, PS@-";
                                EMIT "subl #12, PS@";;
PUSH_LIT
                          :     $N = ALLOC_SUDO;
                                $N.cnst = op1;
                                PUSH_V $N;;
PUSHCONST
                          :     $N = ALLOC_SUDO ;
                                PUSH_P $N;
                                EMIT "movl #ML" op1
                                ", PS@-";;
PUSHGLOBAL
                          :     $N = ALLOC_SUDO ;
                                PUSH_P $N;
                                EMIT "movl #GL"  op1
                                    ", PS@-" ;;

CALLGLOBFUN 'if_import'   :     EMIT "movl #GL" op1 ", a5";
                                EMIT "movl a5@(4), a5";
                                EMIT "jsr a5@";;
CALLGLOBFUN
                          :     EMIT "jsr _ML" op1 ;;
DEF_FUN
                          :     EMIT ".text" ;
                                EMIT "_ML" op1 "  :";;
```

```
GCODE     ATTRIBUTES          ACTION CODES

JFUN
                        :     EMIT "jmp _MLL" op1 ;;
JMP
                        :     EMIT "jmp _MLL" op1 ;;
JNOT_NEG
                        :     EMIT V;
                              EMIT "jge _MLL" op1 ;;
JNEG
                        :     EMIT V;
                              EMIT "jle _MLL" op1 ;;
JNOT_ZERO
                        :     EMIT V;
                              EMIT "jne _MLL" op1 ;;
JZERO
                        :     EMIT V;
                              EMIT "jeq _MLL" op1 ;;

JGLOBFUN 'if_import'    :     EMIT "movl #GL" op1 ", a5";
                              EMIT "movl a5@(4), a5";
                              EMIT "jsr a5@";;
JGLOBFUN
                        :     EMIT "jsr _ML" op1 ;;
J_NOT_PTR
                        :     EMIT "movl PS@, a5";
                              EMIT "cmpl #APPLY, a5@";
                              EMIT "jne _MLL" op1;
                              EMIT "cmpl #PAIR, a5@";
                              EMIT "jne _MLL" op1;;
J_IF_PTR
                        :     EMIT "movl PS@, a5";
                              EMIT "cmpl #APPLY, a5@";
                              EMIT "jeq _MLL" op1;
                              EMIT "cmpl #PAIR, a5@";
                              EMIT "jeq _MLL" op1;;
```

```
GCODE      ATTRIBUTES          ACTION CODES

UPDATE     'arg_1'       :     P[op1] = P[0];
                               POP_P;
                               EMIT "movl PS@+, a0";
                               EMIT "movl PS@, a5";
                               EMIT "movl a0@, a5@";
                               EMIT "movl a0@(4), a5@(4)";;
UPDATE
                         :     P[op1] = P[0];
                               POP_P;
                               EMIT "movl PS@(4*" op1
                                    "), a5";
                               EMIT "movl PS@+, a0";
                               EMIT "movl a0@, a5@";
                               EMIT "movl a0@(4), a5@(4)";;
UPDATE_PR
                         :     P[op1] = P[0];
                               POP_P 2;
                               EMIT "movl PS@(4*" op1
                                    "), a5";
                               EMIT "movl #PAIR, a5@";
                               EMIT "movl PS@+, a0";
                               EMIT "movl a0, a5@(4)";
                               EMIT "movl PS@+, a0";
                               EMIT "movl a0, a5@(8)";;
RET
                         :     EMIT "rts";;
RET_INT
                         :     PUSH_P V[0];
                               EMIT "movl PS@, a5";
                               EMIT "movl #VAL, a5@";
                               EMIT "movl VS@+, a5@(4)";
                               EMIT "rts";;
EVAL
                         :     EMIT "jsr _eval";;
```

```
GCODE       ATTRIBUTES          ACTION CODES

FUN                       :     PUSH_P op1;
                                EMIT "ML" d_label "   :" ;
                                EMIT ".long 3, _ML" op2
                                     "," op1;;
INT
                          :     EMIT "ML" d_label "   :" ;
                                EMIT ".long 1," op1 ",0";;
LABEL
                          :     EMIT "_MLL" op1 "   :" ;;
EXPORT
                          :     EMIT ".globl _" op1 ;
                                EMIT "_" op1 "   :" ;
                                EMIT "jsr _ML" op2;
                                EMIT "rts";;
IMPORT
                          :     EMIT "GL" op2 "   :";
                                EMIT ".long 3, _" op1
                                     ",1" ;;
```

APPENDIX F MACHINE DESCRIPTION TABLE FOR INTEL 286/310

```
GCODE       ATTRIBUTES      ACTION CODES

ADD                     :   EMIT "mov ax , [di]";
                            EMIT "add [di-2] , ax";
                            EMIT "sub di , 2";;

MOD                     :   EMIT "mov ax , [di]";
                            EMIT "div [di-2]";
                            EMIT "mov [di-2] , ax";
                            EMIT "sub di , 2";;

NEG                     :   EMIT "neg [di]";;

INCR                    :   EMIT "add [di] , WORD PTR 1";;

GET_FST
                        :   EMIT "mov si , [bx-2*" op1 "]";
                            EMIT "mov ax , [si+2]";
                            EMIT "mov [di+2] , ax";
                            EMIT "add di , 2";;
FST
                        :   EMIT "mov si , [bx]";
                            EMIT "mov ax , [si+2]";
                            EMIT "mov [bx] , ax";
                            EMIT "mov ax , [si]";
                            EMIT "mov [di+2] , ax";
                            EMIT "add di , 2";;
GET_BYTE
                        :   EMIT "mov [di+2] , WORD PTR " op1;
                            EMIT "add di , 2";;
```

```
GCODE     ATTRIBUTES     ACTION CODES

POPP                  :   EMIT "sub bx , 2";;

POPV                  :   EMIT "sub di , 2";;

MOVEP                 :   EMIT "mov ax , [bx]";
                          EMIT "mov [bx-2*(1+" op1 ")] , ax";
                          EMIT "sub bx , 2";;
MOVEV
                      :   EMIT "mov ax , [di]";
                          EMIT "mov [di-2*(1+" op1 ")] , ax";
                          EMIT "sub di , 2";;
COPYP
                      :   EMIT "mov ax , [bx-2*" op1 "]";
                          EMIT "mov [bx+2] , ax";
                          EMIT "add bx , 2";;
COPYV
                      :   EMIT "mov ax , [di-2*" op1 "]";
                          EMIT "mov [di+2] , ax";
                          EMIT "add di , 2";;
LABEL
                      :   EMIT "LL" op1 ":";;
ALLOC
                      :   EMIT "mov si , G_mem_ptr";
                          EMIT "mov [bx+2] , si";
                          EMIT "add G_mem_ptr , 6";
                          EMIT "add bx , 2";;
MK_VAL
                      :   EMIT "mov si , G_mem_ptr";
                          EMIT "mov [si] , WORD PTR BAS";
                          EMIT "mov ax , [di]";
                          EMIT "mov [si+2] , ax";
                          EMIT "mov [si+4] ,WORD PTR NIL";
                          EMIT "mov [bx+2] , si";
                          EMIT "sub di , 2";
                          EMIT "add bx , 2";
                          EMIT "add G_mem_ptr , 6";;
```

```
GCODE      ATTRIBUTES      ACTION CODES

MK_APP
                       :   EMIT "mov si , G_mem_ptr";
                           EMIT "mov [si] , WORD PTR APPLY";
                           EMIT "mov ax , [bx]";
                           EMIT "mov [si+2] , ax";
                           EMIT "mov ax , [bx-2]";
                           EMIT "mov [si+4] , ax";
                           EMIT "mov [bx-2] , si";
                           EMIT "sub bx , 2";
                           EMIT "add G_mem_ptr , 6";;
MK_PR
                       :   EMIT "mov si , G_mem_ptr";
                           EMIT "mov [si] , WORD PTR PR";
                           EMIT "mov ax , [bx]";
                           EMIT "mov [si+2] , ax";
                           EMIT "mov ax , [bx-2]";
                           EMIT "mov [si+4] , ax";
                           EMIT "mov [bx-2] , si";
                           EMIT "sub bx , 2";
                           EMIT "add G_mem_ptr , 6";;
PUSH_LIT
                       :   EMIT "mov [di+2] , WORD PTR " op1;
                           EMIT "add di , 2";;
PUSHCONST
                       :   EMIT
                           "mov [bx+2] , OFFSET _DATA:FD" op1;
                           EMIT "add bx , 2";;
PUSHGLOBAL
                       :   EMIT
                           "mov [bx+2] , OFFSET _DATA:GD" op1;
                           EMIT "add bx , 2";;
DEF_FUN
                       :   EMIT "L" op1 ":" ; ;
JFUN
                       :   EMIT "jmp L" op1 ;;
JMP
                       :   EMIT "jmp L" op1 ;;
```

```
GCODE      ATTRIBUTES      ACTION CODES

JGLOBFUN   'if_import'
                        :   EMIT "mov si , OFFSET _DATA:GD" op1;
                            EMIT "mov ax , [si+4]";
                            EMIT "call dx";;
JGLOBFUN
                        :   EMIT "jmp L" op1 ;;
JNOT_ZERO
                        :   EMIT "jne LL" op1 ;;
JZERO
                        :   EMIT "je LL" op1 ;;
JNOT_NEG
                        :   EMIT "jae LL" op1 ;;
JNEG
                        :   EMIT "jb LL" op1 ;;
J_NOT_PTR
                        :   EMIT "mov ax , [di]";
                            EMIT "sub di , 2";
                            EMIT "sub ax , WORD PTR APPLY";
                            EMIT "jb LL" op1 ;;
J_IF_PTR
                        :   EMIT "mov ax , [di]";
                            EMIT "sub di , 2";
                            EMIT "sub ax , WORD PTR APPLY";
                            EMIT "jae LL" op1 ;;
CALLGLOBFUN 'if_import'
                        :   EMIT "mov si , OFFSET _DATA:GD" op1;
                            EMIT "mov ax , [si+4]";
                            EMIT "call dx";;
CALLGLOBFUN
                        :   EMIT "mov cx , [di]";
                            EMIT "inc cx";
                            EMIT "add bx , 2";
                            EMIT "mov si , bx";
                            EMIT "LT" op1 ":";
                            EMIT "mov ax , [si-2]";
                            EMIT "mov [si] , ax";
                            EMIT "sub si , 2";
                            EMIT "loop LT" op1 ;
                            EMIT "mov [si] , WORD PTR NIL";
                            EMIT "call L" op1 ;;
```

```
GCODE      ATTRIBUTES      ACTION CODES

UPDATE
                        :   EMIT "mov cx , di";
                            EMIT "mov si , [bx-2*" op1 "]";
                            EMIT "mov di , [bx]";
                            EMIT "mov ax , [di]";
                            EMIT "mov [si] , ax";
                            EMIT "mov ax , [di+2]";
                            EMIT "mov [si+2] , ax";
                            EMIT "mov ax , [di+4]";
                            EMIT "mov [si+4] , ax";
                            EMIT "mov di , cx";
                            EMIT "sub bx , 2";;
UPDATE_PR
                        :   EMIT "mov si , [bx-2*" op1 "]";
                            EMIT "mov [si] , WORD PTR PR";
                            EMIT "mov ax , [bx]";
                            EMIT "mov [si+2] , ax";
                            EMIT "mov ax , [bx-2]";
                            EMIT "mov [si+4] , ax";
                            EMIT "sub bx , 4";;
RET
                        :   EMIT "mov ax , [bx]";
                            EMIT "mov [bx-2] , ax";
                            EMIT "sub bx , 2";
                            EMIT "ret";;
RET_INT
                        :   EMIT "mov si , [bx]";
                            EMIT "mov [si] , WORD PTR BAS";
                            EMIT "mov ax , [di]";
                            EMIT "mov [si+2] , ax";
                            EMIT "mov ax , [bx]";
                            EMIT "mov [bx-2] , ax";
                            EMIT "sub bx , 2";
                            EMIT "ret";;
```

```
GCODE      ATTRIBUTES      ACTION CODES


EVAL
                           :   EMIT "call _eval";;
INIT
                           :   EMIT "_DATA SEGMENT WORD";;
GCODE
                           :   EMIT "_DATA ENDS";;
FUN
                           :   EMIT "FD" d_label " dw 1,"
                                op1 ",OFFSET _TEXT:L" op2;;
INT
                           :   EMIT "FD" d_label " dw 1,"
                                op1 "," op2;;
EXPORT
                           :   EMIT "L" op1 ":";
                               EMIT "jmp L" op2;
                               EMIT "ret";;
IMPORT
                           :   EMIT "GD" d_label " dw 1,"
                                op1 ",OFFSET _TEXT:L" op2;;
```

APPENDIX G: TARGET MACHINE INSTRUCTION TABLE

The following is a list of format numbers that are used to define the VAX 11/780 instructions.

Format 1	r0
Format 2	r0 , r1
Format 3	r0, r1, r2

r0, r1, r2 represent target machine registers. In the above formats results are stored in the last register.

The following table contains the instructions and their format numbers.

VAX 11/780 instruction	Format no.
incl	1
mnegl	1
decl	1
clrl	1
addl2	2
subl2	2
mull2	2
divl2	2
addl3	3
subl3	3
mull3	3
divl3	3
bitl3	3
bisl3	3