Simple Machine Description Grammars

Jack W. Davidson

Computer Science Technical Report 85-22
November 26, 1985

Simple Machine Description Grammars

Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22903

ABSTRACT

Many recent automatic code generators use descriptions of the target machine's instruction set to simplify development of high-quality back ends. This paper describes the machine descriptions that are used with PO, a retargetable peephole optimizer. These machine descriptions are written in a manner similar to the grammars used to describe the language being compiled. Indeed, they are unique in that conventional parser generators used to process the grammar for the front end can be used to build the back end. These machine descriptions are flexible, easy to write, and produce optimizers that use a small amount of memory.

November 26, 1985

Department of Computer Science

The University of Virginia

Charlottesville, VA 22901

Simple Machine Description Grammars

1. Introduction

Most compilers can be logically divided into two phases. The analysis phase, often called the front end, performs lexical, syntax, and semantic analysis and produces an intermediate representation of the source program. Building a front end is a well understood process and many high-quality tools are available to assist in its development. The synthesis phase, or back end, translates the intermediate representation of the source program into object code for the target machine. In contrast to the front end, building high-quality back ends traditionally has been the most difficult and least understood aspect of compiler construction.

Several approaches have been devised recently to ease the difficulty of developing back ends. One of the most promising techniques involves writing a description of the target machine's instruction set. This technique, known as table-driven code generation, offers several advantages over other methods of code generation [Gana82a, Glan78, Grah80, Grah82]. Conceptually retargeting the back end simply requires presenting a description of the new target machine to the code generator's table construction routines. While the above is seldom the case in practice, table-driven code generators do provide a well-structured and consistent framework for the construction of back ends.

A different approach is taken with compilers developed using PO, a retargetable peephole optimizer [Davi80, Davi84a]. The traditional model of compilation places code optimization before code generation [Aho86]. Compilers developed using PO optimize after code generation [Davi84b]. To enhance portability, the front end emits code for a simple abstract machine. This allows a clean separation between the language and machine aspects of the compiler. A naive code generator expands the abstract machine codes to simple object code sequences that are subsequently improved by PO. A schematic of a retargetable compiler constructed using PO is shown in Figure 1. Compilers developed using PO are particularly easy to retarget and produce object code that is comparable to other automatic code generation techniques.

PO is similar to table-driven code generators in that retargeting is achieved by writing a description of the target machine's instruction set. This paper describes these machine descriptions and how they are written. They are unique in that they are similar to a grammar and actions that describe the syntax and semantics of the source language. For a machine description, a grammar and actions that describe the syntax and semantics of the target machine's instruction set are written. Indeed, the same tools used to process the grammar describing the source language can be used to process the grammar describing the machine's instruction set. This technique has been used to describe the instruction sets of

seven different machines and used in retargetable compilers for three different languages. One major computer manufacturer is using these machine descriptions and PO in a large in-house compiler project.

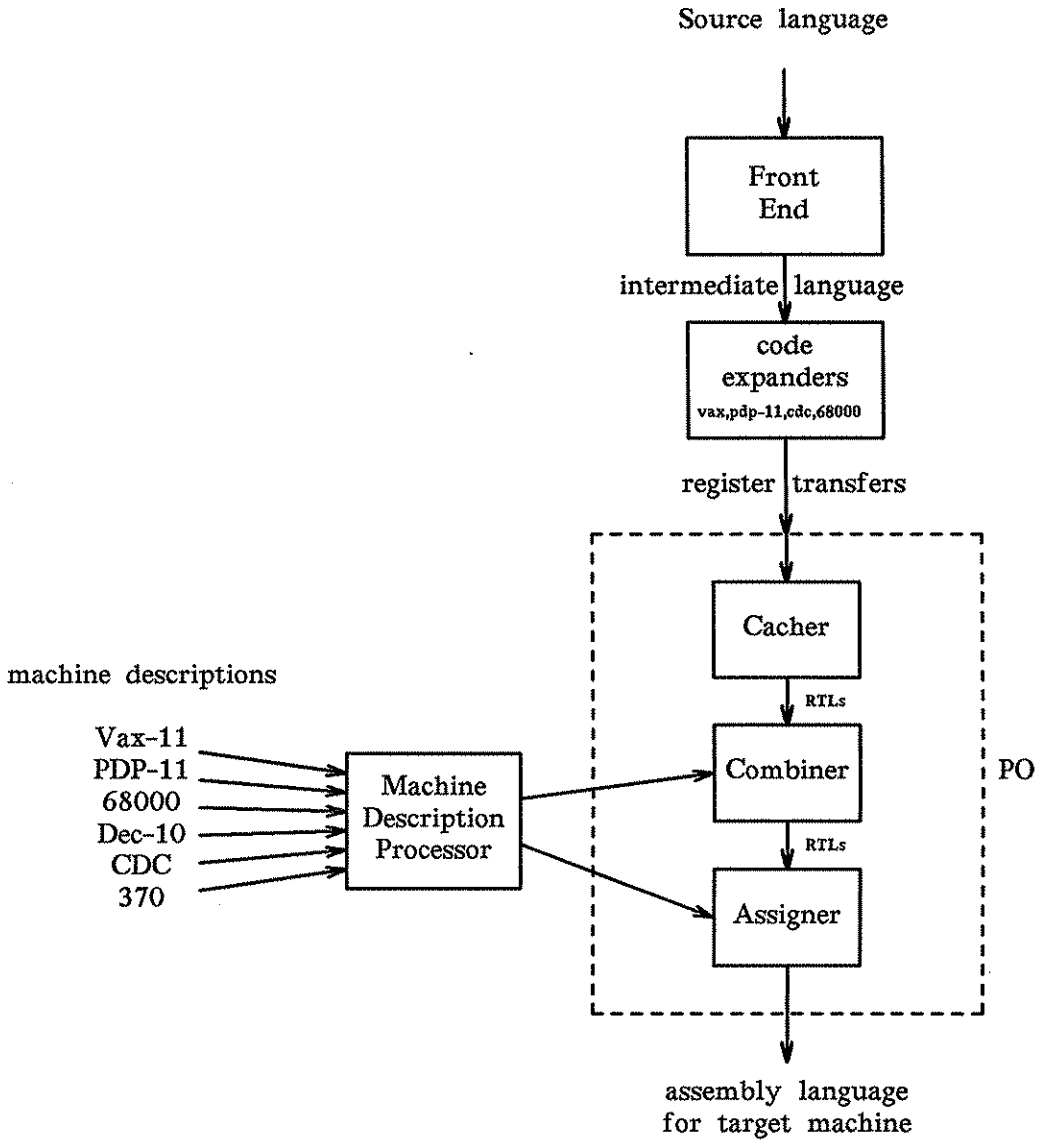

Figure 1. Schematic of a Retargetable Compiler Developed using PO

2. PO

PO is made up of three distinct phases called Cacher, Combiner, and Assigner (see Figure 1). Each phase operates on register transfer lists ('RTLs') which describe an instruction's effect. Cacher eliminates common subexpressions, identifies dead variables, and computes the logical adjacency of instructions. Combiner symbolically simulates logically adjacent pairs of instructions to learn their combined effect. It then determines if a single instruction

exists with the same effect. If it finds one, it replaces the original instruction with the equivalent singleton. After Combiner has finished, Assigner performs register assignment and translates the now optimized RTLs to assembly language. Other documents offer more complete treatments of the implementation and operation of PO [Davi81] and its use in building retargetable compilers [Davi84a].

A target machine description is used by PO in two ways. It is used to build the recognizer used by Combiner to identify legal instructions and the transducer used by Assigner to translate RTLs to assembly language. The correct operation of the recognizer and transducer is essential to the correct operation of PO. In addition, the time to write and debug a new machine description is a substantial portion of the time required to retarget the compiler. Consequently, the ability to quickly and accurately describe a machine's instruction set and correctly convert the description to a recognizer is extremely important.

3. Machine Descriptions

A machine description ('MD') is a grammar and actions that describe the syntax and semantics of a machine's instruction set. Writing a MD is similar to writing a lexical analyzer and parser for a programming language. Using MDs that are written and processed in a manner similar to grammars and semantic actions for programming languages has two benefits. First, it permits the compiler writer to conceptualize building the front end and back end as similar tasks. To build the front end, a grammar and semantic actions that describe the source language are written. Similarly, to build the back end, a grammar and semantic actions that describe the target machine's instruction set are written. Second, because they are similar tasks the same theory and tools can be used to construct both phases. Most compiler writers are very familiar with the theory and implementation of parsers and the use of a parser generator. The MDs described here are processed using the familiar tools, Lex [Lesk79] and Yacc [John78].

In abstract terms to write a MD we must write a grammar, M, that generates a language, $L(M)$, that contains only the RTL strings (sentences) that represent legal instructions for the target machine. Due to the simple syntactic structure of a well-formed RTL string that represents a machine instruction, a MD grammar is usually much easier to write than a grammar for a programming language. Similarly, the actions to check the semantics of a machine instruction are particularly simple.

Just as the front end of a compiler consists of a lexical analyzer and a parser, a recognizer for the instruction set consists of a lexical analyzer to break a RTL string into tokens and a parser to analyze the syntactic structure of the string. Each of these parts is built by writing a specification that is fed into a program generator that automatically constructs the lexical analyzer and the parser. These specifications form the MD. The following sections describe these specifications.

3.1 Lexical Specification

The lexical analyzer for the RTL language is built using the program generator Lex. Lex was chosen for several reasons. A lexical analyzer built using Lex interfaces nicely to a parser that is constructed using the program generator Yacc. Both of these programs are popular, easy to use tools available under most implementations of Unix. In addition, it is simple to modify a Lex specification to recognize the tokens of a new machine's instruction set. Typically, the lexical specification describes the form of identifiers, labels, and special machine symbols such as registers and the program counter. In addition, it identifies the operators used to describe common machine operations such as addition, subtraction, multiplication, etc., and the operators used to denote more complex operations such as pushing an operand on a stack, normalizing a floating point number, or performing an indexed jump.

The simplest form of a Lex specification consists of a set of rules. A rule consists of a regular expression and an action. An action is a program fragment that is executed whenever the regular expression matches some portion of the input string. The actions for a MD specification perform two operations. It installs the string matched in a string table, and returns the token number that denotes the type of token recognized. A sample of a rule from the MD for the Ridge 32 [Ridg84] is:

```
[a-zA-Z_][a-zA-Z0-9_]*  {
   yylval = install(yytext);
   return(ID);
   }
```

The above rule and action specifies an assembly language identifier on the Ridge. The value returned by `install` is assigned to the variable `yylval` to allow the parser constructed by Yacc to examine the string. This is often necessary because the semantic actions of Yacc must check the context-sensitive aspects of a RTL.

Typically a lexical specification is quite small. In addition, the lexical structure of the RTL language varies little from machine to machine. Consequently, the lexical specification for one machine can often be used for a new machine with little or no modification. For instance, the lexical specification for the Ridge is 97 lines, while the lexical specification for the PDP-11 is slightly larger consisting of 110 lines. However, 80 percent of the two specifications is identical.

3.2 Syntax Specification

The parser to recognize legal machine instructions is built using the parser generator Yacc. Yacc has one quality that makes it useful for building a recognizer for a machine's instruction set. It permits the controlled use of ambiguous grammars. These ambiguities are resolved using a set of disambiguating rules. The use of ambiguous grammars allows a grammar for machine's instruction set to be written in a perspicuous style.

A Yacc specification for a machine's instruction set is divided into three sections. These sections describe the addressing modes, the individual effects of instructions, and finally the instructions. The following sections describe the three logical parts of a MD grammar using

excerpts from the MD for the PDP-11 as examples.

3.2.1 Addressing Modes

The addressing modes for the PDP-11 are defined by the following Yacc rules:

```
%token   LB RB      /* left and right bracket */

/* source addressing modes */
src  :   dst
     |   expr                                  { $$ = addr(ANY, IMMED, $1, NULL, NULL); }
     ;

/* destination addressing modes */
dst  :   REG                                   { $$ = addr(ANY, REGISTER, $1, NULL, NULL); }
     |   mt LB expr RB                         { $$ = addr($1, DIRECT, $3, NULL, NULL); }
     |   mt LB REG '+' expr RB                 { $$ = addr($1, INDEX, $3, $5, NULL); }
     |   mt LB REG PLUSPLUS RB                 { $$ = addr($1, AUTOINC, $3, NULL, NULL); }
     |   mt LB MINUSMINUS REG RB               { $$ = addr($1, AUTODEC, $4, NULL, NULL); }
     |   mt LB REG RB                          { $$ = addr($1, INDIR, $3, NULL, NULL); }
     |   mt LB 'w' LB REG PLUSPLUS RB RB       { $$ = addr($1, AUTOINCDFR, $5, NULL, NULL); }
     |   mt LB 'w' LB MINUSMINUS REG RB RB     { $$ = addr($1, AUTODECDFR, $6, NULL, NULL); }
     |   mt LB 'w' LB REG '+' expr RB RB       { $$ = addr($1, INDXDFR, $5, $7, NULL); }
     |   mt LB 'w' LB expr RB RB               { $$ = addr($1, DIRECTDFR, $5, NULL, NULL); }
     |   mt LB 'w' LB REG RB RB                { $$ = addr($1, INDIRDFR, $5, NULL, NULL); }
     ;

/* assembly-time expressions */
expr :   expr '+' expr                         { $$ = addr(ANY, EXPR, '+', $1, $3); }
     |   expr '-' expr                         { $$ = addr(ANY, EXPR, '-', $1, $3); }
     |   '-' expr %prec '*'                    { $$ = addr(ANY, EXPR, '-', $2, NULL); }
     |   ID                                    { $$ = addr(ANY, ID, $1, NULL, NULL); }
     |   CON                                   { $$ = addr(ANY, CON, $1, NULL, NULL); }
     |   '1'                                   { $$ = addr(ANY, CON, $1, NULL, NULL); }
     |   '0'                                   { $$ = addr(ANY, CON, $1, NULL, NULL); }
     ;

mt   :   'w'                                  { $$ = WORD; }
     |   'b'                                   { $$ = BYTE; }
     ;
```

The actions for the addressing modes calls the routine `addr` that builds a record that contains information about the addressing mode. A pointer to the record constructed is returned as the value associated with the nonterminal or rule.

By factoring the grammar it is often possible to write concise MDs. In the above example, by factoring out whether the addressing mode is referencing a word or byte (denoted by 'w' or 'b'), one set of rules defines both the word and byte addressing modes of the PDP-11.

3.2.2 Instruction Effects

The second section describes the individual effects of instructions. The individual effects for some of the arithmetic instructions on the PDP-11 are shown below.

```
/* add instruction effects */
addi :   dst '=' dst '+' src ';'              { $$ = addsubi($1, $3, $5); } ;
addn :   NZ '=' dst '+' src '?' '0' ';'       { $$ = addsubn($3, $5); } ;

/* complement instruction effects */
comi :   dst '=' '~' dst ';'                  { $$ = unopi($1, $4); } ;
comn :   NZ '=' '~' dst ';'                   { $$ = record($4, NULL, NULL); } ;

/* increment instruction effects */
inci :   dst '=' dst '+' '1' ';'              { $$ = unopi($1, $3); } ;
incn :   NZ '=' dst '+' '1' '?' '0' ';'       { $$ = record($3, NULL, NULL); } ;

/* subtract instruction effects */
subi :   dst '=' dst '-' src ';'              { $$ = addsubi($1, $3, $5); } ;
subn :   NZ '=' dst '-' src '?' '0' ';'       { $$ = addsubn($3, $5); } ;
```

In the above examples, there are two rules for each instruction. One rule describes the computation of the result, while the second rule describes the setting of the condition codes.

The actions for the individual effects perform two operations. They check the semantics of the instruction if necessary. For example, since the PDP-11 is a two-address machine, the action for the `addi` rule must check that both occurrences of `dst` denote the same location. It also insures that the operands `dst` and `src` denote word operands. The second operation that is performed by all individual effect rules is to produce a semantic record that contains information about the operands and operation to be performed.

Again by noting the similarity of instructions, it is possible to reduce the amount of code that must be written to implement the semantic actions. For example, the add and subtract instructions share the same semantic action routines. Similarly, all the unary instructions share the same semantic action routines.

3.2.3 Instructions

The third section of a MD grammar composes the previously defined individual instruction effects into instructions. The left hand side of the first grammar rule in this section is the *start* symbol for the grammar. For example, using the individual effects defined in the previous section, the instruction definitions for the PDP-11 are:

```
inst :   addi addn                            { binop("add", $1, $2); }
     |   addn addi                            { binop("add", $2, $1); }
     |   addi                                 { binop("add", $1, NULL); }
     |   addn                                 { binop("add", NULL, $1); }
     |   comi comn                            { unop("com", $1, $2); }
     |   comn comi                            { unop("com", $2, $1); }
     |   comi                                 { unop("com", $1, NULL); }
     |   comn                                 { unop("com", NULL, $1); }
     |   inci incn                            { unop("inc", $1, $2); }
     |   incn inci                            { unop("inc", $2, $1); }
     |   inci                                 { unop("inc", $1, NULL); }
     |   incn                                 { unop("inc", NULL, $1); }
     |   subi subn                            { binop("sub", $1, $2); }
     |   subn subi                            { binop("sub", $2, $1); }
     |   subi                                 { binop("sub", $1, NULL); }
     |   subn                                 { binop("sub", NULL, $1); }
     ;
```

For instructions that have several effects, all possible combinations of the effects must be described. Using the add instruction with two effects as an example, the four possible combinations that must be described are:

```
1.  dst = dst + src; NZ = dst + src ? 0;
2.  NZ = dst + src ? 0; dst = dst + src;
3.  dst = dst + src;  (NZ dead)
4.  NZ = dst + src ? 0;  (dst dead)
```

While there are a few instructions with three effects (15 combinations), fortunately we have not encountered any instructions that required four individual effects.

The actions for an instruction definition have two functions. During the optimization phase when instructions are combined, the actions must check and enforce instruction semantics. During the register assignment phase, the actions convert the RTL to the corresponding assembly language instruction.

The optimization phase actions normally perform two types of semantic checks. The first type of semantic check insures that the context sensitive requirements of an instruction are satisfied. For example, the semantic action for the add instruction, `binop`, must check that the portion of the input that matched the nonterminal `dst` in the first effect is identical to the input that matched `dst` in the second effect. The second type of semantic check insures that when only one effect of an instruction appears that the left hand side of the missing effect appears on the dead variable list.

3.3 Ambiguous Grammars

One of the qualities that distinguishes Yacc from other conventional parser generators is that it allows the controlled use of ambiguous grammars. This allows the MD grammar to be written in a straightforward natural style. To handle ambiguous grammars, Yacc invokes two disambiguating rules by default.

1. In a shift/reduce conflict, the default action is to do the shift.
2. In a reduce/reduce conflict, the default action is to reduce by the grammar rule appearing first in the input specification.

It is up to the grammar writer to determine if the default action chosen by Yacc is the correct action.

Our experience with writing MDs has been that it is possible to avoid reduce/reduce conflicts entirely. In the case of shift/reduce conflicts, the default action, shift, is usually the correct one. In the few cases where the obvious description of an instruction resulted in erroneous shift/reduce conflict resolution, it was a simple matter to modify the grammar to avoid the conflict.

4. Experience

The MDs described here were developed as a replacement to the MDs previously used with PO [Davi81, Davi84a]. The major motivation for the development of these new MDs were problems and limitations encountered with the old descriptions [Crow82, Hanc85]. The following section compares the new MDs to the old MDs.

4.1 Comparison with Old Machine Descriptions

The old MDs were also used to construct a recognizer and transducer for RTLs. The recognizer and transducer were built by processing the machine specification with a SNOBOL4 [Gris71] program that produced Lex input specifications. These specifications were converted by Lex into set of finite state automatons that recognized legal RTLs for the described machine. The finite state automata implementation was chosen because measurements indicated that the speed of the optimizer would hinge on the speed of the instruction recognition process. The use of a Lex-constructed finite state automaton, however, was the source of several problems. First, the use of a lexical analyzer limited the size of the MD that could be processed. While machines with relatively small instruction sets could be handled (e.g., the PDP-11 or the CDC Cyber series machines), machines with large instruction sets could not be completely described. A second problem was that most all MDs have constructs that can only be described using context-free grammars. For example, most MDs contain a specification of the syntax of assembly-time expressions. Consequently regular expressions were not powerful enough to accurately describe the machine. The result was that it was possible to write rules that incorrectly identified a RTL string as representing a legal instruction. In these cases, it is necessary to rewrite the MD to prevent these erroneous optimizations. The use of a MD based on the theory of context-free languages and employing a parser generator like Yacc to process the MD specification solves both of these problems.

Another advantage to the Yacc-based MDs is that the user can write powerful semantic action routines. These actions can make decisions and perform optimizations that were difficult to generate automatically. One of the problems with the old MDs and MD processor was that it was often possible to have a RTL that represented a legal instruction, yet it was difficult to determine the proper instruction to emit.† For example, consider the following RTL for the VAX-11/780

```
NZ = r[2] + r[3] ? 0;
```

This RTL specifies that the condition codes (NZ) are set according to the sum produced by the addition of register two and register three. If register two appears on the dead-variable list for this instruction, the optimizer can emit the instruction:

†Thanks to Terry Crowley for this example.

```
addl2   r3,r2
```

If, however, register two is not on the dead-variable list it is more difficult to determine the proper instruction to emit. If any other registers appear on the dead-variable list, the optimizer may generate a three-address add instruction that sets any one of the dead registers. For example, if register four appears on the dead-variable list, the optimizer may emit:

```
addl3   r3,r2,r4
```

If there are no registers on the dead-variable list, the semantic action could request a free register and temporarily use it for the result. While the above logic could be generated automatically by a MD processor, it would be difficult. On the other hand, the code to perform the above actions is relatively simple to write by hand. Indeed, the code to make such decisions is part of the MD for the VAX-11/780.

It is also possible to write semantic actions that perform additional machine-specific optimizations. For example, the semantic action routine for the multiply instruction on the Concurrent Computer 3200 includes a routine that produces the best code sequence for an integer multiply by a constant. This MD also includes semantic actions that emit special idiomatic multi-instruction sequences that are more efficient than equivalent single instructions.

The Yacc-based MDs enjoy several other advantages over the original MDs. One of the motivations for the original implementation was the desire for high-speed. With the introduction of HOP [Davi84c], the requirement for high-speed instruction recognition became less of an issue. Surprisingly, it turns out that the Yacc-based instruction recognizer runs as fast as the Lex-based implementation. The Yacc-based optimizers are considerably smaller than the Lex-based optimizer. An optimizer for the PDP-11 built using a Lex-based MD occupies 92,388 bytes, while the Yacc-based optimizer occupies 59,560 bytes, a savings of 36 percent. Table 1 presents data on the size of four MDs produced using Yacc.

Machine Description	VAX-11	PDP-11	Ridge	Concurrent 3200
Grammar Symbols	91	80	65	309
Productions	150	136	72	596
Tables				
States	405	370	149	1342
S/R conflicts	12	9	1	7
R/R conflicts	0	0	0	0
Size (bytes)	6250	4240	1928	14924
Total Size (bytes) (including semantic routines)	9050	5248	2232	20876

Table 1. Statistics on Grammar Size

The other advantage of the Yacc-based MDs is that they take much less time to process. It takes approximately 8.5 minutes of CPU time on a VAX-11/780 to process a Lex-based MD and produce an optimizer. A functionally identical Yacc-based MD for the PDP-11 requires only 1.5 minutes of CPU time to produce an optimizer. This faster processing

time is particularly important when the MD is being developed and debugged. It is also important when the MDs are being used to specify, test and evaluate experimental instruction sets [Davi85].

4.2 Machine Description Database

The one disadvantage of the Yacc-based MDs is that they are not as concise as the Lex-based MDs. For instance, the Lex-based MD for the VAX-11/780 is 165 lines. The Yacc-based MD is 780 lines (280 lines for the lexical and syntax specifications, and 505 lines of C code for the semantic actions). Many of the semantic checks automatically produced by the MD processor must be coded explicitly in the Yacc-based implementation. Fortunately, writing a Yacc-based MD for a particular machine can still be done relatively quickly. The reason for this is while the assembly language syntax (e.g., the opcode mnemonics, notation for addressing modes, etc.) for different machines varies considerably, the RTL description of a particular operation varies little from machine to machine. For example, a RTL description of a two-address addition for many machines is

```
dst '=' dst '+' src ';'
```

where `dst` and `src` specify the addressing modes.

This similarity of RTLs across machines and the existence of a database of MDs for seven different machines makes writing a new MD comparatively easy. When developing a MD for a new machine, it is usually possible to consult the database for an existing MD that describes a similar instruction set or similar architectural feature and extract and modify parts of that description. For instance, the MD for the VAX-11/780 was developed in two days by the author by modifying the MD for the PDP-11, a closely related machine. Similarly, a MD for the Prime 9950 was developed using the MD for the DECsystem-10. As more machines are described, the MD database becomes more useful.

4.3 Other Experience

The largest and most complete MD has been written by Concurrent Computers to describe its 3200 series processor. The MD describes the entire instruction set which consists of 182 instructions and 8 addressing modes. The MD grammar consists of 600 grammar rules and 1300 lines of C code for the semantic actions. The semantic actions includes code to perform many machine specific operations such as converting a integer multiplication by a constant to a series of shifts and adds and emitting more efficient multi-instruction sequences in place of single instructions. This MD was developed in a month by some one unfamiliar with PO and the operation of Lex and Yacc. The general feeling is that subsequent MDs would require substantially less time to develop.

5. Discussion

The idea of producing a parser from a machine description is not new. This idea forms the basis of the Graham-Glanville style code generators [Gana82b, Glan78]. A MD in the Graham-Glanville system consists of a set of productions where the left-hand side specifies the results of an operation and the right-hand side specifies the operation. A target machine instruction computing the right-hand side is supplied with each production. The code generator emits instructions for an abstract machine called the *Compiler Writer's Virtual Machine* (CWVM). The parser produced from the MD converts the CWVM code to the assembly language of the target machine. Much of the recent research with the Graham-Glanville code generators has been concerned with speeding up the parser table construction algorithms, automatically handling looping conditions and syntactic blocks, and reducing the size of the parser tables [Aigr84, Grah82].

While PO and the Graham-Glanville technique present two different methods for developing retargetable compilers, they are similar in that they both rely on MD grammars. Consequently, it is worthwhile to compare the two description techniques. The most fundamental difference is in what is being described. In the Graham-Glanville system, a MD is a grammar that defines a mapping between the instruction set of the CWVM and the target machine. This requires knowledge of the particular parsing technique used and of the instruction sets of both the CWVM and the target machine. Several specialized programs are used to assist with the development of a MD and the construction of the parser tables.

In contrast, because PO only requires a recognizer that accepts legal target machine instructions and rejects those that are not, the job of writing a PO MD is conceptually and pragmatically simpler. Consequently, we are able to use conventional parsing algorithms and an existing parser generator to process the grammar. The resulting parser tables are quite small (see Table 1).

The simplicity of the PO MDs is gained at the expense of requiring code generation to be performed by a separate phase (the Code Expander, see Figure 1). This phase is normally written by hand. As noted in a previous paper [Davi84a], several benefits could be realized if the table-driven code generator were combined with the table-driven peephole optimizer. The responsibility for the production of high-quality code could then be divided between the code generator and the peephole optimizer, reducing the sometimes large size of the code generation tables and the sometimes slow execution speed of the the optimizer. A simple, unified MD that would drive both the code generator and the optimizer should be possible. The development of PO MDs that are processed in a manner similar to the Graham-Glanville MDs is a first step towards developing such a MD.

6. Summary

This paper has described a novel technique for specifying the instruction sets of machines. These machine descriptions are part of a powerful peephole optimizer that can be used to simplify the development of high-quality retargetable compilers. This technique is appealing for several reasons. It permits the compiler writer to conceptualize the development of a back end as a task similar to the development of the front end. Indeed, the tasks are so similar that existing well known theories and tools that are used to develop the front end can also be used to develop the back end. The optimizers developed using this technique are substantially smaller and run no slower than optimizers developed using a previous description method. While the machine descriptions are larger than a previous description method, they actually take no longer to develop due to the existence of a machine database that describes most architectural features found on contemporary machines.

7. References

[Aho86] A. V. Aho, R. Sethi and J. D. Ullman, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[Aigr84] P. Aigrain, S. L. Graham, R. R. Henry, M. K. McKusick and E. Petegri-Llopart, Experience with a Graham-Glanville Style Code Generator, *Proceedings SIGPLAN Notices '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, 13-24.

[Crow82] T. R. Crowley, *Combining Table-Driven Effect Selection and Description-Driven Peephole Optimization for Automatic Code Generation*, M.S. Thesis, Massachusetts Institute of Technology, Boston, MA, September 1982.

[Davi80] J. W. Davidson and C. W. Fraser, The Design and Application of a Retargetable Peephole Optimizer, *ACM Transactions on Programming Languages and Systems 2*, 2 (April 1980), 191-202.

[Davi81] J. W. Davidson, *Simplifying Code Generation Through Peephole Optimization*, PhD Dissertation, University of Arizona, December 1981.

[Davi84a] J. W. Davidson and C. W. Fraser, Code Selection through Object Code Optimization, *Transactions on Programming Languages and Systems 6*, 4 (October 1984), 7-32.

[Davi84b] J. W. Davidson and C. W. Fraser, Register Allocation and Exhaustive Peephole Optimization, *Software - Practice and Experience 14*, 9 (September 1984), 857-866.

[Davi84c] J. W. Davidson and C. W. Fraser, Automatic Generation of Peephole Optimizations, *Proceedings of the SIGPLAN Notices '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984, 111-116.

[Davi85] J. W. Davidson, Fast Interpretation of Instruction Sets: Implementation and Applications, *Proceedings of the 7th Annual Symposium on Computer Hardware Description Languages and their Application*, Tokoyo, Japan, August 1985, 179-191.

[Gana82a] M. Ganapathi and C. N. Fischer, Description-Driven Code Generation using Attribute Grammars, *Conference Record of the Ninth Annual Symposium on Principles of Programming Languages*, Albuquerque, NM, January 1982, 108-119.

[Gana82b] M. Ganapathi, C. N. Fischer and J. L. Hennessy, Retargetable Compiler Code Generation, *Computing Surveys 14*, 4 (December 1982), 573-592.

[Glan78] R. S. Glanville and S. L. Graham, A New Method for Compiler Code Generation, *Conference Record of the Fifth Annual Symposium on Principles of Programming Languages*, Tucson, AZ, January 1978, 231-240.

[Grah80] S. L. Graham, Table-Driven Code Generation, *IEEE Computer 13*, 8 (August 1980), 25-34.

[Grah82] S. L. Graham, R. R. Henry and R. A. Schulman, An Experiment in Table Driven Code Generation, *Proceedings SIGPLAN Notices '82 Symposium on Compiler Construction*, Boston, MA, June 1982, 32-42.

[Gris71] R. E. Griswold, J. F. Poage and I. P. Polonsky, *The Snobol4 Programming Language*, Prentice-Hall, Inc, Englewood Cliffs, NJ, 1971.

[Hanc85] J. K. Hancock, *Experience Using a Retargetable Peephole Optimizer to Achieve Compiler Portability*, M. S. Thesis, University of Colorado, Boulder, CO, January 1985.

[John78] S. C. Johnson, Yacc: Yet Another Compiler-Compiler, *Unix Programmer's Manual 2B*, Section 19 (July 1978), 1-34.

[Lesk79] M. E. Lesk, Lex - A Lexical Analyzer Generator, *Unix Programmer's Manual 2B*, Section 20 (January 1979), 1-13.

[Ridg84] *Ridge Processor Reference Manual*, Ridge Computers, Santa Clara, CA, 1984.