Analyzing Exotic Instructions for a Retargetable Code Generator[1]

Thomas M. Morgan and Lawrence A. Rowe

Computer Science Division
Department of Electrical Engineering and Computer Science
University of California
Berkeley, California 94720

Abstract

Exotic instructions are complex instructions, such as block move, string search, and string edit, which are found on most conventional computers. Recent retargetable code generator and instruction set analysis systems have not dealt with exotic instructions. A method to analyze exotic instructions is presented which provides the information needed by a retargetable code generator. The analysis uses source-to-source transformations to prove the equivalence of high-level language operators to exotic instructions. Examples are presented which illustrate the analysis process.

1. Introduction

Exotic instructions are characterized by complex semantics, usually involving looping behavior. These instructions can be thought of as performing a sequence of primitive actions. Exotic instructions are useful because they can often perform operations in less time and space than an equivalent sequence of primitive actions. Examples of exotic instructions are block move instructions, linked list instructions and string edit instructions. Exotic instructions occur on such diverse machines as the VAX-11 [DEC76], the Burroughs B4800 [Burroughs76], the Intel 8086 [Intel79], and the IBM 370 [IBM81].

Recent work in producing retargetable code generators [Glanville78, Cattell80] has focused on generating code for expression evaluation and simple control structures. The code generators which have been produced have used the arithmetic, logical, and branching instructions of the host machine, but have not used the exotic instructions of a machine's repertoire. Similarly, [Oakley79] has formulated a method to analyze a general purpose machine description and provide input to a retargetable code generator. Oakley is able to produce assertions describing the effects of instructions by symbolically executing a procedural description of the target machine. However, his method is unable to deal with exotic instructions due to their greater complexity. This paper describes an approach to analyzing exotic instructions and the features required of a retargetable code generator to generate them.

[1]This work was supported in part by the National Science Foundation under Grant MCS 80-06329.

The analysis of exotic instructions is based on the fact that high-level languages often have operators or runtime routines whose semantics closely match those of exotic instructions. For instance, assignment between strings in Pascal is essentially the same operation as that performed by the block move instruction (**movc3**) on the VAX-11. Thus, the Berkeley Pascal Compiler for the VAX-11 generates **movc3** to implement string assignment.

The semantics of an exotic instruction are often close to those of a language operator but do not match exactly. For example, the Burroughs B4800 has an instruction to search through a linked list of records for a record with a specified field. However, the instruction assumes that the link field of the list is the first field in the record. Thus, the B4800 instruction can only be used to implement a general list search operation if a specific *constraint* is satisfied, namely, that the link field is the first field of the record.

The analysis of exotic instructions must identify which instructions can be used to implement which high-level operators. Moreover, the special conditions under which the use of the instruction is valid must also be identified. This information can then be passed to a retargetable code generator enabling it to generate exotic instructions.

The Exotic Instruction Transformational Analysis System (EXTRA) has been designed and implemented to explore the analysis of exotic instructions. EXTRA uses source-to-source transformations [Loveman76] to prove the equivalence of high-level language operators to exotic instructions. EXTRA takes a description of a high-level language operator and a description of an exotic instruction. The descriptions are transformed until they are equivalent. Transformations are provided to transform descriptions into semantically equivalent descriptions, to uncover constraints on the equivalence, and to *augment* exotic instructions with other code when needed. In the current implementation of EXTRA the user must guide the system by choosing the transformations to be performed and by choosing the code to augment the exotic instruction description. EXTRA verifies that the transformations can be correctly applied and applies them. The ultimate goal is to provide a system that operates with little or no user intervention.

The rest of the paper presents details and examples of the analysis system. Section 2 describes the important characteristics of exotic instructions. The details of the analysis method are presented in section 3. Section 4 presents examples of instructions analyzed by EXTRA, as well as an instruction that could not be analyzed. Section 5 presents our experience with the system including a summary of all the instructions which have been analyzed to date. Compiler support for exotic instructions is discussed in section 6. Finally, section 7 presents our conclusions and ideas for future research.

2. Exotic Instructions

Exotic instructions occur on a wide variety of current computers. In a sample of 6 machines, representing 6 different manufacturers, 67 string and list processing exotic instructions were identified[2]. These instructions perform string operations such as string move, search, compare, and edit, and list operations such as list search, link, and unlink. Statistics on the number of such instructions for each machine are given in table 1.

Machine	Number of Exotic Instructions
Intel 8086	6
DG Eclipse	5
Univac 1100	21
IBM 370	7
Burroughs B4800	16
VAX-11	12
Total	67

Table 1: Exotic Instruction Statistics

The exotic instructions on the various machines all have certain features in common. First, the instructions are characterized by complex semantics. This complexity includes:

- The presence of loops in procedural descriptions of the instructions.
- The setting of multiple registers and memory locations by the instructions.
- Special addressing requirements, such as operating on fixed registers.

The complexity of exotic instructions makes them difficult to analyze. The presence of loops in particular means the exotic instructions cannot be symbolically executed. Thus, the analysis methods in [Oakley79] cannot be applied.

Although exotic instructions are complex, their semantics are far from arbitrary. Exotic instructions have a close relationship to operators and data types in high-level languages. There are exotic instructions which manipulate character strings, vectors, and lists, all of which are data types in high-level languages. While exotic instructions exist to manipulate those data types, the operations which the instructions perform rarely match exactly the operations defined in the high-level languages. However, the instructions can be made to match the language operators by:

- *Constraining* the operands of the language operator.
- *Simplifying* an instruction that is too complex by fixing the values of some of its operands.
- *Augmenting* the instruction by adding extra code to the prologue or epilogue of the instruction.

Constraints are important because the language operator may match the exotic instruction in only a conditional way. A common type of constraint deals with storing operands to the instruction in fields of limited size. The size of the field limits the range of values which the instruction can operate on. An example of this type of size constraint is the IBM 370 **mvc** instruction which has a string length operand in a byte field and thus can only move at most 256 characters at a time. Other constraints deal with the addressing of operands. For instance, operands to an exotic instruction may have to be in fixed memory locations or registers. Other restrictions that would be handled by a storage allocator are also possible, as in the case of the B4800 list search instruction described earlier.

[2]These machines also contain other exotic instructions such as decimal and array indexing instructions which are not covered in this paper.

When exotic instructions are too complex to match a language operator, it is sometimes possible to simplify the instruction. For example, an exotic instruction may be more general than a language operator. In which case, the instruction can be simplified by fixing the values of some of its operands. For example, the Intel 8086 string instructions can process strings from low addresses to high or vice versa depending on the setting of the *df* flag. Those instructions can be simplified by forcing *df* to have a fixed value (e.g., setting *df* to 0 so that strings are always processed low addresses to high). Then the instructions can thought of as simpler instructions with one less operand.

While some exotic instructions match a language operator directly, many instructions only match if some extra code is added to the instruction. The code may need to be added to either the prologue or the epilogue of the instruction. With extra code the augmented instruction is able to compute the same results as the language operator. For example, the VAX-11 **locc** instruction searchs a string for a character and returns the address of character if found. However, the PL/1 **index** operator (if used to search for a single character) returns the index of the character in the string, and not the address in memory. Thus, code must be added to **locc** to compute the index from the address if **locc** is to match **index**.

In summary exotic instructions occur on a wide variety of machines. Exotic instructions are complex which has prevented them from being analyzed by previous methods. Because exotic instructions are closely related to operators in high-level languages they can be used to implement the language operators under certain constraints or when simplifications or augments are made to the instructions. The next section shows how exotic instructions with the above characteristics can be analyzed.

3. Analysis of Exotic Instructions

The goal of the analysis is to establish that an exotic instruction can be used to implement a high-level language operator. In addition, the analysis must discover any conditions that control when the implementation is valid.

Both exotic instructions and language operators are represented by descriptions written in a notation similar to ISPS [Barbacci77]. An instruction can be used to implement a language operator if their descriptions are equivalent. The descriptions are equivalent if they can be transformed into a common form through a sequence of source-to-source transformations. The descriptions are in a common form if they are identical except for variable and register names. The application of source-to-source transformations changes the procedural descriptions, but not the results that are computed. For example, the sample transformation in figure 1 reverses the clauses of a conditional, but has no effect on the semantics of the conditional as a whole.

```
if exp                          if not exp
then                            then
  stmt_list1         =>           stmt_list2
else                            else
  stmt_list2                      stmt_list1
end_if                          end_if
```

Fig. 1: Reverse Conditional Transformation

A transformation can be applied whenever the syntactic and data flow conditions that control it are satisfied. Also, some transformations can be applied only when certain conditions on the values of variables in the description are satisfied. These conditions become constraints on the values of the high-level language operator's operands and must be satisfied at compile-time in order for the instruction to be generated.

In some instances, the instruction and the language operator are not equivalent, but the instruction can still be used by the code generator. When they are not equivalent it is sometimes possible that a *simplified instruction* can be constructed by restricting the values of some of the instruction's operands. Alternatively, a variant of the instruction can be constructed. The variant, or *augmented instruction*, is constructed by adding additional code to the prologue or epilogue of the exotic instruction. The simplified or augmented instruction can then be proven equivalent to the language operator and, consequently, the instruction can be used to implement the language operator.

During the matching process, variables in the language operator description are bound to real registers in the instruction description. This binding may result in further constraints on the values of the operands of the language operator. The operands will be constrained to have values in the range determined by the size of the register.

The results of the analysis are passed to a retargetable code generator as part of the instruction repertoire of the machine. The code generator can then generate the augmented instruction when it can satisfy the constraints or verify that they are already satisfied in the original source program.

As was mentioned earlier, both the exotic instruction and the language operator are represented by descriptions written in a notation similar to ISPS. Each description can be broken up into various sections. Each section may contain register and function declarations. Main memory is represented as the array *Mb*. Statements in the language include loop statements (**repeat**), conditionals (**if**), loop exits (**exit_when**), and explicit i/o statments (**input** and **output**). In order to facilitate data flow computations the language has been restricted to eliminate aliasing among registers. For example, only call-by-value parameters are allowed. Also, the addressing calculations associated with operands to the instruction are not represented. The prototype system assumes that addressing calculations can be manipulated by other methods since inclusion of addressing modes would make the descriptions more difficult to deal with.

The next section shows how the source-to-source transformations, constraints, simplifications, and augments have been used to analyze real instructions.

4. Examples

The analysis of exotic instructions can best be illustrated by examples of instructions analyzed by EXTRA. The first example presents a detailed account of how the Intel 8086 **scasb** instruction can be analyzed with respect to the Rigel **index** operator. The second example, shows how an idiosyncrasy on the IBM 370 is handled. Finally, some of the limits of EXTRA are presented by way of an instruction and a language operator that are equivalent, but could not be successfully analyzed.

4.1. Intel 8086 scasb/Rigel index

The Rigel[3] [Rowe81] **index** operator searches a string for a specified character and, if the character is found, the index of the character is returned. However, if the character is not found then the **index** operator returns zero. A description of the Rigel **index** operator is given in figure 2.

[3]Rigel is an experimental language designed for research into the development of interactive data base applications.

```
index.operation := begin

   ** SOURCE.ACCESS **

      Src.Base : integer,              ! string base address
      Src.Index : integer,             ! string index
      Src.Length : integer,            ! string length

      read() : integer := begin
         read ← Mb[Src.Base + Src.Index];
         Src.Index ← Src.Index + 1;
      end

   ** STATE **

      ch : character                   ! character sought

   ** STRING.PROCESS **

      index.execute := begin

         input (Src.Base, Src.Length, ch);
         Src.Index ← 0;
         repeat
            ! exit when string exhausted
            exit_when (Src.Length = 0);
            ! exit if char is found
            exit_when (ch = read());
            Src.Length ← Src.Length - 1;
         end_repeat;
         if Src.Length = 0
         then
            output(0);                 ! char not found
         else
            output(Src.Index);         ! char found
         end_if;

      end
end
```

Fig. 2: Rigel **Index** Operator

The **scasb** instruction on the Intel 8086 also searches a string for a specified character. **Scasb** takes as operands: 1) the address of the string which must be preloaded into register *di*, 2) the length of the string which must be preloaded into register *cx*, and 3) the character sought which must be in register *al*. If the character is found, the *zf* condition code is set and *di* contains the address of the character which follows the character that was found.

There are also several flag operands that control how **scasb** is executed. In order for an entire string to be searched the repeat flag *rf* must be set. The string can be searched from low addresses to high addresses or vice versa, depending on the setting of the direction flag *df*. Besides searching for a character, the **scasb** instruction can be used to scan over all occurrences of a character by setting the *rfz* flag. A description of the **scasb** instruction is given in figure 3.

There are several key steps in the analysis of **scasb** that are now presented. As was noted above the **scasb** instruction has several flag operands that control the execution of the instruction. The instruction can be simplified by setting these flags to fixed values and then propagating this information to the places were the flags are used. The information can then be used to simplify the expressions and statements that involve those operands. For instance, by setting the *df* flag to zero the string access is simplified to always search from low to high addresses. The other simplifications on **scasb**

```
scasb.instruction := begin
  ! segment addressing ignored in this description

  ** SOURCE.ACCESS **

      di<15:0>,                      ! source string address
      cx<15:0>,                      ! source string length

      fetch()<7:0> := begin          ! fetch source character
        fetch ← Mb[ di ];
        if df                        ! control direction of fetch
        then
          di ← di − 1;               ! high-to-low addresses
        else
          di ← di + 1;               ! low-to-high addresses
        end_if
      end

  ** STATE **

      rf<>,                          ! repeat flag
      df<>,                          ! direction flag
      rfz<>,                         ! exit condition flag
      zf<>,                          ! last compare zero flag
      al<7:0>                        ! character sought

  ** STRING.PROCESS **

      scasb.execute := begin
        input (rf, rfz, df, zf, di, cx, al);
        if (not rf)
        then ! no repetition
          if (al − fetch()) = 0
          then
            zf ← 1;
          else
            zf ← 0;
          end_if;
        else ! repeat mode
          repeat
            exit_when (cx = 0);
            cx ← cx − 1;
            if (al − fetch()) = 0
            then
              zf ← 1;
            else
              zf ← 0;
            end_if;
            ! exit on condition
            exit_when (rfz and (not zf))
                   or ((not rfz) and zf);
          end_repeat;
        end_if;
        output (zf, di, cx);
      end
  end
```

Fig. 3: Intel 8086 **Scasb** Instruction

involve setting the *rf* flag to 1 and setting the *rfz* flag to 0. Setting *rf* means that **scasb** always loops, and setting *rfz* means that the instruction terminates when the character is found.

The information that the flags have been set to fixed values can be propagated in the description of **scasb**. The description can then be simplified by applying constant folding transformations. For example, the constant folding transformations simplify the exit condition at the end of the loop from:

```
exit_when (rfz and (not zf)) or ((not rfz) and zf);
```

to:

```
exit_when zf;
```

The above simplification of the description is possible because the value of the operand *rfz* has been fixed at 0. The description of **scasb** after all of the simplifications is given in figure 4.

```
scasb.instruction := begin

  ** SOURCE.ACCESS **

      di<15:0>,                      ! source string address
      cx<15:0>,                      ! source string length

      fetch()<7:0> := begin          ! fetch source character
        fetch ← Mb[ di ];
        di ← di + 1;                 ! low-to-high addresses
        end_if
      end

  ** STATE **

      zf<>,                          ! last compare zero flag
      al<7:0>                        ! character sought

  ** STRING.PROCESS **

      scasb.execute := begin
        input (zf, di, cx, al);
        repeat
          exit_when (cx = 0);
          cx ← cx − 1;
          if (al − fetch()) = 0
          then
            zf ← 1;
          else
            zf ← 0;
          end_if;
          ! exit on condition
          exit_when (zf);
        end_repeat;
        output (zf, di, cx);
      end
end
```

Fig. 4: Simplified Intel 8086 **Scasb** Instruction

After the simplifications, the *zf* flag can almost be used to identify what caused the exit from the loop (this is important when augmenting the epilogue to compute the character index from its address). If *zf* is set, the exit was caused by the character being found, otherwise the string is exhausted. However, there is a problem. In the case where the input length of the string is zero, *zf* would never be set and its value at loop exit would be unusable. Therefore, code must be added to the beginning of **scasb** which initially sets *zf* to zero.

An additional prologue augment is needed to save the initial string pointer value. Since **index** requires the index of the sought character to be returned, the generated code must compute it. This computation can be done by subtracting the initial string address from the final address that is produced by **scasb**. Hence, a temporary must be allocated and code must be added to store the initial pointer value.

Code can now be added to the epilogue of **scasb** that checks the condition that caused the loop to exit and then compute the index of the character that was found or return 0 if the character was not found. The code that must be added is:

```
if zf
then
  output( di - temp );
else
  output( 0 );
end_if;
```

where *temp* is the temporary allocated by the previous prologue augment to hold the initial address of the string. The augmented description of **scasb** is given in figure 5.

```
scasb.instruction := begin
  ** SOURCE.ACCESS **

      di<15:0>,                          ! source string address
      cx<15:0>,                          ! source string length

      fetch()<7:0> := begin              ! fetch source character
        fetch ← Mb[ di ];
        di ← di + 1;                     ! low-to-high addresses
        end_if
      end

  ** STATE **

      zf<>,                              ! result of last comparison
      al<7:0>,                           ! character sought
      temp<15:0>                         ! new temporary

  ** STRING.PROCESS **

      scasb.execute := begin
        zf ← 0;                          ! augmented code
        input (di, cx, al);
        temp ← di;                       ! augmented code
        repeat
          exit_when (cx = 0);
          cx ← cx - 1;
          if (al - fetch()) = 0
          then
            zf ← 1;
          else
            zf ← 0;
          end_if;
          ! exit on condition
          exit_when (zf);
        end_repeat;
        if zf                            ! augmented code
        then
          output( di - temp );
        else
          output( 0 );
        end_if;
      end
end
```

Fig. 5: Augmented Intel 8086 **Scasb** Instruction

Although the descriptions of **index** in figure 2 and **scasb** in figure 5 are now equivalent, further transformations are required to verify the equivalence. These transformations do not change the semantics of either description, but rather just change the descriptions to an equivalent form. For instance, it must be shown that the method of comparison used in **scasb** is equivalent to that used in **index**. Also, the transformations must show that the use of the flag *zf* in **scasb** is equivalent to the method in **index** that does not use the flag. EXTRA has in its repertoire sufficient transformations to allow it to express these equivalent implementations.

After the analysis is complete, there is a final step that must be performed to allow the **scasb** instruction to be generated. The code that was added to the prologue and epilogue of **scasb** must now have real Intel 8086 code generated for it. This requires the use of the code generator of the target compiler. This process was done by hand for **scasb** and resulted in the following Intel 8086 code being produced to represent the augmented instruction:

```
       ; operands already loaded:
       ;    di...string address
       ;    cx...string length
       ;    al...character sought

       mov bx,di    ; save initial address
       mov si,0     ; clear si to use in resetting zf
       cmp si,1     ; reset zero flag zf
       cld          ; reset direction flag df
       rep          ; set rf and reset rfz
       scasb        ; search string
       jz l1        ; jump if not found
       sub di,bx    ; compute index of char if found
       jmp l2
   l1: mov di,0     ; return zero if not found
   l2:              ; final result stored in di
```

This augmented instruction is then bound to an intermediate language operator representing **index**. The augmented instruction is passed to a retargetable code generator.

Constraints due to location sizes are also passed to the code generator. In the example, since the *Src.Length* variable of the **index** description is bound to the *cx* register of **scasb**, a constraint is developed that the string length must fit into 16 bits. This constraint is a trivial one to satisfy on the Intel 8086 since the word size of the machine is 16 bits. On the VAX-11, however, string lengths are also limited to 16 bits which produces a non-trivial constraint since the word size is 32 bits.

The analysis of **scasb** with respect to **index** requires 73 total transformation steps. The **scasb** instruction had to be both simplified and augmented in order to match the **index** operator. The result of the analysis is an augmented instruction for the Intel 8086 which is customized to implement the **index** operator.

4.2. IBM 370 mvc/Pascal string assignment

The **mvc** instruction on the IBM 370 can be analyzed with respect to the string assignment operator (**sassign**) of Pascal[4]. The analysis requires 105 transformation steps which is the greatest number of any instruction analyzed to date. This section shows how a quirk in **mvc** is handled by the analysis system.

The **mvc** instruction moves a sequence of bytes from a source to a destination field. The number of bytes is specified by an 8 bit length field in the instruction format. The number of characters moved by **mvc** is the value of the length field plus one. Hence, a length value of zero means that one character is to be moved and a length of one means that two characters are to be moved and so forth[5]. On the other hand, the specification of **sassign** contains a length operand that contains the actual number of characters to be moved. Thus, EXTRA must show how the two encodings of the string length can be made equivalent.

The equivalence can be shown through the use of a transformation that creates a coding constraint on the length of the Pascal string. The coding constraint is actually a directive to the compiler to decrement the length of the **sassign** operand before using it as an operand to **mvc**. Furthermore, the decrement becomes part of the description of **mvc**.

Once the coding constraint is introduced into the description of **mvc** the decrement is integrated with the rest of the description. The description is then transformed into a form that moves exactly the number of characters as is specified in the length field. Finally, other transformations are used to show that **mvc** and **sassign** match.

[4]The string assignment operator **sassign** is actually present only in the compiler internal form and not in the Pascal language.

[5]This type of encoding is not unique to the IBM 370, but also occurs on at least one other machine (the Burroughs B4800).

The **mvc** example demonstrates the idiosyncratic nature of exotic instructions and how EXTRA is able to deal with it. **Mvc** and **sassign** are semantically very close but are not equivalent unless a specific condition is identified and satisfied through the coding constraint. The example also shows how constraints can represent various types of directives to the compiler besides simple conditions on the values of operands.

4.3. VAX-11 movc3/Pascal string assignment

An example of an analysis that cannot be performed by EXTRA is the analysis of the VAX-11 **movc3** instruction with respect to the Pascal string assignment operator. **Movc3** moves a source string to a destination and guards against overlap in the strings. If the source address is less than the destination address then the string is moved from high addressed bytes to low, otherwise the low addressed bytes are moved first. For example, if the source address were 10 and the destination address were 12 the string "abc" would be moved by moving the "c" then the "b" and finally "a", since moving in the opposite way would result in the destination string being "aba" not "abc". Because the description of **movc3** reflects the overlap protection the description is more complex than one might at first think.

In the Pascal language it is impossible to have strings that overlap. Hence, the string assignment operator **sassign** can have a simple description that always moves the string from low addresses to high addresses. Because Pascal strings cannot overlap, **movc3** and **sassign** are equivalent. The problem is that the descriptions are equivalent only under this condition and EXTRA has no way to represent it.

One way to represent no overlap would be as the complicated constraint below:

(Src.Base + Src.Length ≤ Dst.Base)
or (Dst.Base + Dst.Length ≤ Src.Base)

This condition would guarantee that the strings would not overlap, since it says that either the source string ends before the destination string begins or that the destination string ends before the source string begins. However, the current version of EXTRA has no ability to deal with complicated constraints that involve more than one operand.

EXTRA can only deal with constraints that are of the form: 1) an operand is constrained to have a certain value, 2) an operand is constrained to be in a certain range, or 3) an operand is to be offset by a value (as was done with **mvc** above). EXTRA can only deal with simple constraints since we originally thought that only simple constraints would occur in real instructions. However, what this example points out is that there is a need to deal with more complicated constraints that are due to the characteristics of the source language. The no-overlap condition is a property of Pascal and can never be violated by any Pascal program[6]. Thus, the analysis system is the appropriate place to deal with it.

5. Current Status of EXTRA

The EXTRA system is written in Franz Lisp [Foderaro80] and runs under the UNIX[7] operating system on a VAX-11/780. The system monitor allows the user to move through the language operator and machine instruction descriptions in much the same way as a Lisp-oriented structure editor. Transformations are applied in the same way that the the descriptions would be edited, by positioning the cursor at the point of interest and then specifying the transformation to be applied.

[6]However, some Pascal extensions do allow overlap and use different semantics for copy, and thus would require a differant treatment.

[7]UNIX is a Trademark of Bell Laboratories.

The current implementation of EXTRA includes 75 transformations in the transformation library. The transformations themselves utilize various types of data flow information that is used to determine whether a transformation is valid at a particular point. The transformations can be categorized based on the functions they perform and the type of information they require. The seven categories are:

- *Local transformations* which manipulate the descriptions based on local properties. These transformations include arithmetic and logical identities.
- *Code motion transformations* which move statements with respect to one another, such as reversing the order of two statements or moving one statement into the body of another when possible.
- *Loop transformations* which manipulate the loops in the descriptions. These are especially necessary to manipulate the counting loops for string oriented instructions.
- *Global transformations* which must look at potentially the entire description. For instance, *copy propagation* and *dead variable elimination* both use information that may be a long distance textually from where it is used.
- *Routine structuring transformations* which change how a description is structured into different routines. For instance, a routine with several calls may be changed into several routines each with a single call.
- *Constraint and assertion transformations* which manipulate constraints and assertions in the descriptions. These transformations allow constraints and auxiliary assertions to be created and manipulated by transformations like any other part of the description text.
- *Augment producing transformations* that produce prologue and epilogue augments to the descriptions. The user specifies the augment, and the system guarantees the interface of the augment code to the exotic instruction.

Although the transformation library is in no sense exhaustive (other transformation libraries can be found in [Standish76, Barstow77]), the set used seems to form a kernel of necessary transformations to deal with exotic instructions. A complete list and descriptions of the transformations can be found in [Morgan82].

In order to test our theories, experiments have been performed to see how well EXTRA can analyze real instructions. To date, the system has been used to analyze 8 different exotic instructions on the IBM 370, VAX-11, and Intel 8086. The instructions have been analyzed with respect to operators or runtime routines in Pascal, PL/1, Rigel, and CLU. Table 2 shows the instructions and the language operators that have been successfully analyzed. Each analysis required from 21 to 105 transformation steps.

The descriptions that have been analyzed have come from a variety of sources to eliminate bias caused by a single style in writing the descriptions. The descriptions for the machine instructions were derived from flow charts in machine reference manuals and on-line ISPS descriptions available from Carnegie-Mellon University. Descriptions for the language operators were derived from the code for run-time routines and descriptions in the language manuals. The descriptions were translated in a straightforward manner into the description language used by EXTRA. Thus, we feel that the ability of EXTRA to analyze exotic instructions is not dependent on a certain style of writing the descriptions.

Machine	Instruction	Language	Operation	Steps
Intel 8086	movsb	Pascal	string move	52
Intel 8086	movsb	PL/1	string move	66
Intel 8086	scasb	Rigel	string search	73
Intel 8086	scasb	CLU	string search	86
Intel 8086	cmpsb	Pascal	string compare	79
VAX-11	movc3	PC2[8]	block copy	21
VAX-11	movc5	PC2	block clear	26
VAX-11	locc	Rigel	string search	33
VAX-11	locc	CLU	string search	32
VAX-11	cmpc3	Pascal	string compare	47
IBM 370	mvc	Pascal	string move	105

Table 2: Exotic Instruction Analysis Summary

There have been some failures of the analysis system. The problem with the VAX-11 **movc3** instruction and the Pascal **sassign** operator has already been mentioned. Another problem was with the Data General Eclipse [DG75]. String instructions on the Eclipse can move strings from low addresses to high or vice versa like the instructions on the Intel 8086. However, instead of encoding the direction in a specific flag the direction is encoded in the length operand for each string. If the length is greater than zero then the string is processed from low addresses to high. Otherwise, the string is processed in the reverse order. The problem is that the length operand is now used for two unrelated purposes and it is difficult to formulate transformations to separate the two functions. This points out a general problem. Instructions that use a *clever coding trick* make analysis difficult or impossible.

Experience with EXTRA has also indicated some problems with the mechanics of the analysis process. Many of the transformations are at too low a level and thus the user gets involved in a mass of detail. For example, the simplifications mentioned earlier can require many steps. Also, there is no way to structure the matching process. The user cannot match parts of the descriptions independently. The entire descriptions must be matched as a whole.

Finally, the results of EXTRA have been compared to an analysis done by hand to generate exotic instructions in Pascal and Rigel compilers at Berkeley. The comparison revealed obscure bugs in the use of VAX-11 instructions in each compiler. Thus, while exotic instructions can be used without a formalized analysis process, there is a chance of bugs due to their complexity.

6. Compiler Support

EXTRA produces a binding between exotic instructions and high-level language operators, as well as constraints on when the binding is valid. In order to use this binding information the compiler must have an internal form that allows high-level language operators to be represented explicitly. The code generator can then generate an exotic instruction when a high-level operator is encountered in the internal form and any constraints can be satisfied. If there is no exotic instruction to implement a high-level operator, or if the constraints can not be satisfied, then the compiler must include *decomposition rules* to transform the high-level operator into a sequence of low-level operations for which code can be generated.

[8]PC2 is the Berkeley Pascal runtime system (written in C).

Data flow information can often be used by the compiler to show that constraints on the values of operand are already satisfied in the source program. Constraints can also be satisfied by *constraint satisfaction rewriting rules*. These rules rewrite the language operator to put it in a context where the constraints are satisfied. For example, a string move operator that is constrained to move strings of at most 65K bytes can be rewritten to move consecutive substrings of size less than or equal to 65K.

Once it has been ascertained that an exotic instruction can be used and that the constraints can be satisfied, then it is often possible to perform optimizations to improve the quality of the code. Three optimizations that have been identified are:

- *Integration* of rewriting rules with augment code. Integration is necessary since augments and rewriting rules are developed independently of one another. Thus, when the two types of added code are put together there is often unnecessary or inefficient computations present.
- *Constant value optimizations* such as constant propagation and constant folding can be used to simplify the code after the rewriting rules are applied.
- *Intelligent register allocation* to make use of the special addressing modes sometimes present in exotic instructions. This optimization is especially important when registers are used for dedicated purposes such as on the VAX-11. Since the VAX-11 string instructions leave string addresses in fixed registers those same registers should be used to hold the initial string addresses. Then, if exotic instructions are cascaded or put in loops, additional loads of the registers are not necessary.

Thus, a retargetable code generator can generate exotic instructions if it 1) has a high-level internal form to allow the use of the binding information produced by EXTRA, and 2) has methods to satisfy constraints. Once the constraints have been satisfied, there are optimizations that can be applied to improve the quality of the code. We are currently working on interfacing EXTRA directly to the current version of the Graham-Glanville retargetable code generator as described in [Henry81, Graham82].

7. Conclusions

Initial experience with EXTRA indicates that exotic instructions can be successfully analyzed. EXTRA enables the compiler writer to verify that an exotic instruction can be used to implement a language operator and under what conditions the implementation is valid. The general compiler features necessary to support the generation of exotic instructions have also been identified.

There are several directions for future research. First, EXTRA should be extended to understand source language characteristics such as overlap that result in complex constraints. These language characteristics seem easy for people to understand, but difficult to formalize and use. Second, the compiler interface needs to be completed. This includes the exact form of the information given to a retargetable code generation system as well as support in the analysis system for the compiler optimizations that were mentioned earlier. Finally, methods should be developed to structure the analysis and to help the user in deciding how the analysis should proceed.

Acknowledgements

We would like to thank Kurt Shoens and Robert Henry for their comments on an earlier version of this paper.

8. References

[Barbacci77] M. R. Barbacci, G.E. Barnes, R.G. Cattell, and D.P. Siewiorek, "The ISPS Computer Description Language,," Technical Report, Computer Science Department, CMU, Pittsburgh, PA (August, 1977).

[Barstow77] D. R. Barstow, "Automatic Construction of Algorithms and Data Structures Using a Knowledge Base of Programming Rules," PhD Dissertation, AIM-308, Stanford, CA (November, 1977).

[Burroughs76] Burroughs, *B4800/B3800/B2800 Systems Reference Manual*, Burroughs Corporation (October, 1976).

[Cattell80] R. G. Cattell, "Automatic Derivation of Code Generators from Machine Descriptions," *Transactions on Programming Languages and Systems* **2**(2) pp. 173-190 (April, 1980).

[DEC76] DEC, *VAX-11/780 Architecture Handbook*, Digital Equipment Corporation (1976).

[DG75] DG, "Programmer's Reference Manual: Eclipse Line Computers," 015-000024-04, Data General Corporation (March, 1975).

[Foderaro80] J. K. Foderaro, *The Franz Lisp Manual*, Computer Science Division, University of California, Berkeley (1980).

[Glanville78] R. S. Glanville and S. L. Graham, "A New Method for Compiler Code Generation," *Proc. 5th ACM Symp. Principles of Programming Languages*, (January, 1978).

[Graham82] S. L. Graham and R. R. Henry, "An Experiment in Table Driven Code Generation," *Proc. SIGPLAN Symposium on Compiler Construction (this issue)*, (June, 1982).

[Henry81] R. R. Henry, "The Code Generator Generator's Work Station: Experiments with the Graham Glanville Machine Independent Code Algorithms for Code Generation," Master's Project Report, Electronics Research Laboratory, University of CA, Berkeley (1981).

[IBM81] IBM, "IBM System/370 Principles of Operation," (GA22-7000-8), IBM Corporation Manual (October 1981).

[Intel79] Intel, *The 8086 Family User's Manual*, Intel Corporation (October 1979).

[Loveman76] D. B. Loveman, "Program Improvement by Source to Source Transformation," *Proc. 3rd ACM Symp. on Principles of Programming Languages*, pp. 140-152 (January, 1976).

[Morgan82] T. M. Morgan, "Instruction Set Analysis and Retargetable Code Generation in the Presence of Exotic Instructions," PhD Dissertation, Computer Science Division, University of California, Berkeley, CA (June, 1982).

[Oakley79] J. D. Oakley, "Symbolic Execution of Formal Machine Descriptions," PhD Dissertation, Computer Science Department, CMU, Pittsburgh, PA (April, 1979).

[Rowe81] L. A. Rowe, J. R. Cortopassi, D. P. Doucette, and K. A. Shoens, *Rigel Language Specification*, Computer Science Division, University of California, Berkeley (June, 1981).

[Standish76] T. A. Standish, D. C. Harriman, D. F. Kibler, and J. M. Neighbors, "The Irvine Program Transformation Catalogue," Technical Report, Department of Information and Computer Science, University of California, Irvine, CA (1976).