Fast Interpretation of Instruction Sets: Implementation and Applications

Jack W. Davidson

Computer Science Technical Report 85-05
June 4, 1985

Fast Interpretation of Instruction Sets: Implementation and Applications

Jack W. Davidson

Department of Computer Science
University of Virginia
Charlottesville, VA 22901

ABSTRACT

This paper describes a technique for building a fast interpreter to emulate an arbitrary machine's instruction set. The interpreter is automatically constructed from a specification of the instruction set. The instruction set is described by a machine description language based on ISPS, but is at a sufficiently high level that nonessential detail may be omitted. This same language has also been used to simplify the construction of retargetable compilers. The machine description language along with automatically constructed interpreters and retargetable compilers provides a set of tools that can be used to design and evaluate proposed and existing instruction sets, generate address traces for use in the design and evaluation of memory systems, facilitate testing and debugging of software, and automatically build high-level language interpreters. This paper describes the machine description language, the construction of the interpreters, and the use of the interpreters in the above applications.

June 4, 1985

Department of Computer Science

The University of Virginia

Charlottesville, VA 22901

Fast Interpretation of Instruction Sets:

Implementation and Applications

1. Introduction

To effectively evaluate a proposed instruction set, its behavior must be observed during the execution of real programs. This is often difficult because 1) no programs exist that use the instruction set, and 2) the machine probably has not been built. A few methods of evaluating instruction sets address the second problem [Barb81, Crag83], but not the first.

One method to evaluate a proposed instruction set is to implement the instruction set on a microprogrammable machine. If programs can be generated that use the instruction set, this method provides fast execution of the programs. The difficulty and time-consuming nature of microprogramming, however, discourages experimentation. A second approach to instruction set evaluation is to analyze the execution behavior of programs on existing hardware and then extrapolate from this information. While this method overcomes the above mentioned problems, the new design will be contingent on or influenced by previous design decisions. This can stifle architectural innovation. A third approach is the use of hardware description languages to describe and simulate the proposed instruction sets. A number of different systems exist that have this capability. While this method encourages experimentation, it is still difficult to generate and execute real programs that use the proposed instruction set. Even if it is possible to generate a large set of programs to simulate, the simulation may be so slow that only a few, small unrealistic programs can be run.

This paper describes a technique that provides for the construction of fast interpreters of a machine's instruction set. The interpreters are automatically constructed from a high-level description of the machine's instruction set. These same descriptions are used in the PO compiler generation system to build compilers that produce code for the described machine [Davi84b].

Existing programs can then be compiled and used to observe the behavior of the instruction set. Indeed this system has been used to simulate a 3500 source line high-level program which was compiled using the PO system.

The next section of this paper reviews similar work in this area. Section 3 describes the machine description notation. Section 4 describes the construction and operation of the interpreter. The retargetable compiler system is described in Section 5. Section 6 describes the implementation of the interpreter; Section 7 describes some of the possible applications of this system. Section 8 provides a summary.

2. Background

Of the many computer hardware description languages ('CHDLs') available today, ISPS is one of the most popular [Barb81]. ISPS is based on ISP [Bell71]. ISPS, together with a companion ISPS simulator, have been used to evaluate both new and existing architectures [Barb79]. While ISP was originally developed to primarily describe instruction sets, ISPS has a broader range of applications. It has been used to design circuits and to automatically generate diagnostic tests for machines [Oakl79, Park79]. Further, it has also been used extensively in compiler research [Catt80, Davi81, Davi84b], and as a pedagogical tool.

Writing an ISPS description to document an architecture is relatively straightforward. Quite a lot of detail, however, must be included in the descriptions to be able to simulate assembly language programs using the ISPS simulator. For example, the instruction set designer must specify the encodings for all instruction formats and all addressing modes. In addition opcodes must be assigned to all instructions. While not difficult, this can be a time consuming and tedious task. While these tasks are a necessary part of instruction set design, they draw attention away from the primary job at hand – designing and determining the machine's primitive operations. Another problem with ISPS is the description of complex instructions (e.g. block moves or character compare instructions). This type of instruction is not easily handled by the ISPS notation.

Conventional programming languages have also been used to specify instruction sets. Cragon

describes the use of LISP to specify an instruction set [Crag83]. Instructions are implemented as LISP functions. The advantage of using LISP is that the specification of the instructions can be directly executed. To test this idea, the instruction set of the RISC I [Patt81] was described. To demonstrate the executability of the specification, a seven instruction loop was hand coded and run. While the descriptions themselves are concise and readable, the instruction simulation was somewhat slow. Cragon reported being able to execute 143 instructions per second on a DECsystem-2060.

3. The Machine Description Language

The machine descriptions used in this research are based on ISPS. They are, however, at a sufficiently high level that such detail as opcode assignment and instruction format specification can be ignored. This allows decisions about opcodes and formats to be postponed, resulting in machine descriptions that are easily written and modified. The high level of the description makes it possible to specify complex instructions.

A machine description is a grammar for syntax-directed translation [Aho77] between ISP register transfers and a machine's assembly language. Simplicity was one of the qualities that was desired in the machine description language. This simplicity, however, was gained at the expense of rigorousness. The machine descriptions eschew detail for simplicity and brevity. Thus, they could not be considered a formal specification of the machine. They are, however, quite descriptive and easy to write. Typically a machine description takes a day or two to write by someone familiar with the machine and the description notation.

A machine description is divided logically into two parts. The first part describes the machine's addressing modes, and the second part describes the instructions. This division provides a natural way to describe machines. The addressing modes are described without regard for the instructions that use them. The machine operations and the addressing modes are combined to describe the machine's instruction set. Machine descriptions structured this way are concise and understandable.

To illustrate the machine description language, portions of the machine description for the

VAX–11/780 are used. Some irrelevant detail has been omitted here for clarity and brevity. The descriptions for the VAX–11/780 longword–addressing modes are:

```
STK     :=    m[--r[14]]                          :=    -(r14)

REG     :=    r[RN1]                              :=    rRN

LWORD   :=    m[IDENT]                            :=    IDENT
              m[r[RN1] + IDENT]                   :=    IDENT(rRN1)
              m[r[RN1]++]                         :=    (rRN1)+
              m[--r[RN1]]                         :=    -(rRN1)
              m[r[RN1]]                           :=    (rRN1)
              m[m[r[RN1]++]]                      :=    *(rRN1)+
              m[m[r[RN1] + IDENT]]                :=    *IDENT(rRN1)
              m[m[IDENT]]                         :=    *IDENT
              m[m[r[RN1]]]                        :=    *(rRN1)
              m[r[RN1] << 2 + IDENT]              :=    IDENT[rRN1]
              m[r[RN1] << 2 + r[RN2]]             :=    (rRN2)[rRN1]
              m[r[RN1] << 2 + r[RN2] + IDENT]     :=    IDENT(rRN2)[rRN1]
              m[r[RN1] << 2 + r[RN2]++]           :=    (rRN2)+[rRN1]
              m[r[RN1] << 2 + --r[RN2]]           :=    -(rRN2)[rRN1]
              m[r[RN1] << 2 + m[r[RN2]++]]        :=    *(rRN2)+[rRN1]
              m[r[RN1] << 2 + m[r[RN2] + IDENT]]  :=    *IDENT(rRN2)[rRN1]

LWADDR  :=    IDENT                               :=    IDENT
              r[RN1] + IDENT                      :=    IDENT(rRN1)
              r[RN1]++                            :=    (rRN1)+
              --r[RN1]                            :=    -(rRN1)
              r[RN1] << 2 + r[RN2]                :=    (rRN2)[rRN1]
              r[RN1] << 2 + r[RN2] + IDENT        :=    IDENT(rRN2)[rRN1]
              r[RN1] << 2 + r[RN2] - IDENT        :=    -IDENT(rRN2)[rRN1]
              r[RN1] << 2 + r[RN2]++              :=    (rRN2)+[rRN1]
              r[RN1] << 2 + --r[RN2]              :=    -(rRN2)[rRN1]
              r[RN1] << 2 + m[r[RN2]++]           :=    *(rRN2)+[rRN1]
              r[RN1] << 2 + m[r[RN2] + IDENT]     :=    *IDENT(rRN2)[rRN1]

RL      :=    ==                                  :=    eql
              !=                                  :=    neq
              >=                                  :=    geq
              <=                                  :=    leq
              <                                   :=    lss
              >                                   :=    gtr
```

The first column of the description contains the name of the addressing mode. The second column contains the register transfer syntax that describes the addressing mode. The third column defines the assembly language syntax for the addressing mode. RN1 and RN2 are constants that specify legal register numbers. IDENT is a constant that specifies legal addressing constants. REG and LWORD are the names that can be used to refer to register addressing mode and to the longword addressing modes respectively. LWADDR is the name to refer addressing modes that return addresses. The

description of the addressing modes for other addressable units (e.g. bytes, 16–bit words, float, etc.) would be similar.

The final part of the address definitions groups the previously defined addressing modes into classes that can be used to describe the machine's operations.

```
DSTL    :=      REG|STK|LWORD
SRCL    :=      REG|STK|LWORD
```

The above description defines two classes: DSTL and SRCL which are destinations and sources for longword instructions. A DSTL can be either a register, a reference to the stack, or a longword. Using the definitions of the addressing mode classes allows the instruction definitions to be equally concise. For example, below are definitions for several VAX–11 instructions.

```
NZ = DSTL ? 0;                                  :=      tstl    DSTL
NZ = DSTL ? SRCL;                               :=      cmpl    DSTL,SRCL
NZ = DSTL & SRCL ? 0;                           :=      bitl    DSTL,SRCL
DSTL = 0;NZ = 0 ? 0;                            :=      clrl    DSTL
STK = SRCL;NZ = SRCL ? 0;                       :=      pushl   SRCL
STK = LWADDR;NZ = LWADDR ? 0;                   :=      pushal  LWADDR
DSTL = SRCL;NZ = SRCL ? 0;                      :=      movl    SRCL,DSTL
DSTL = DSTL + 1;NZ = DSTL + 1 ? 0;              :=      incl    DSTL
DSTL = DSTL - 1;NZ = DSTL - 1 ? 0;              :=      decl    DSTL
DSTW = -SRCL;NZ = -SRCL ? 0;                    :=      mnegl   SRCL,DSTL
DSTW = ~SRCL;NZ = ~SRCL ? 0;                    :=      mcoml   SRCL,DSTL
PC = LABEL;                                     :=      jbr     LABEL
PC = NZ RL 0 -> LABEL | PC;                     :=      jRL     LABEL
DSTL = SRCL1 + SRCL2;NZ = SRCL1 + SRCL2 ? 0;    :=      {
   !strcmp(DSTL, SRCL1): addl2 SRCL2,DSTL
   addl3 SRCL2,SRCL1,DSTL
   }
DSTL = SRCL1 / SRCL2;NZ = SRCL1 / SRCL2 ? 0;    :=      {
   !strcmp(DSTL, SRCL1): divl2 SRCL2,DSTL
   divl3 SRCL2,SRCL1,DSTL
   }
```

PC and NZ are the symbols for the program counter and condition codes. Splitting the machine description into a section for addressing modes and operations allows concise descriptions of instructions. For example, the VAX has several conditional jumps. By defining RL in the addressing mode section, the instruction definition section needs only one line to define all relational jumps.

The last two instruction definitions deserve special attention. These describe the longword addition and division instructions respectively. The VAX has two variants of these instructions: a

two–address version and a three–address version. One of the functions of the retargetable peephole optimizer, described in Section 5, is to determine the best instruction to use in a given situation. The above code directs the optimizer to use the shorter two–address variants of the addition and division instructions if possible. The definitions for subtraction, multiplication, and logical operations are similar.

Awkward or complex instructions can be described very simply using the notation. Shortly we shall describe how these instructions are simulated. A good example of some complex instructions are the byte instructions found on the DECsystem–10. These instructions allow the programmer to pack or unpack bytes of any size anywhere within a word. They use and manipulate a pointer that specifies the size of the byte (up to 36 bits), the position of the byte within a word, and the address of the word containing the byte. These instructions can be described using a functional notation. For example, the load byte, increment byte pointer, and increment and load byte instructions are described by:

```
R ← LDB(M)          :=    ldb    R,M        /* load byte */
M ← IBP(M)          :=    ibp    M          /* inc. byte pointer */
R ← LDB(IBP(M))     :=    ildb   R,M        /* increment and load byte */
```

R and M are the definitions for a register and a memory address respectively. Using the load byte instruction as an example, it is only necessary to describe that the LDB function is applied to a memory location indicated by M (the byte pointer) resulting in a byte being extracted and loaded into a register R. Details of how the pointer is explicitly manipulated need not be included in the description. How the these instructions are simulated is described in the next section.

Because of the simplicity of the descriptions, modifications to existing instructions or the addition of new ones is quite easy. The functional notation allows the designer to specify complex instructions without having to worry about implementation details. The details are postponed until the design has stabilized and reached the implementation phase.

4. The Interpreter

Figure 1 shows the structure of the register transfer list ('RTL') interpreter. The purpose of the machine description processor ('MDP') is to produce C [Kern78] language subroutines that implement the addressing modes and instructions described by the machine description.† The machine description simulator library ('MDSIM') contains routines to perform common operations such as addition, subtraction, multiplication, etc. The MDP constructs code that makes calls on these subroutines. As the MDP processes each instruction, the machine dependent instruction library is searched for instruction definitions.

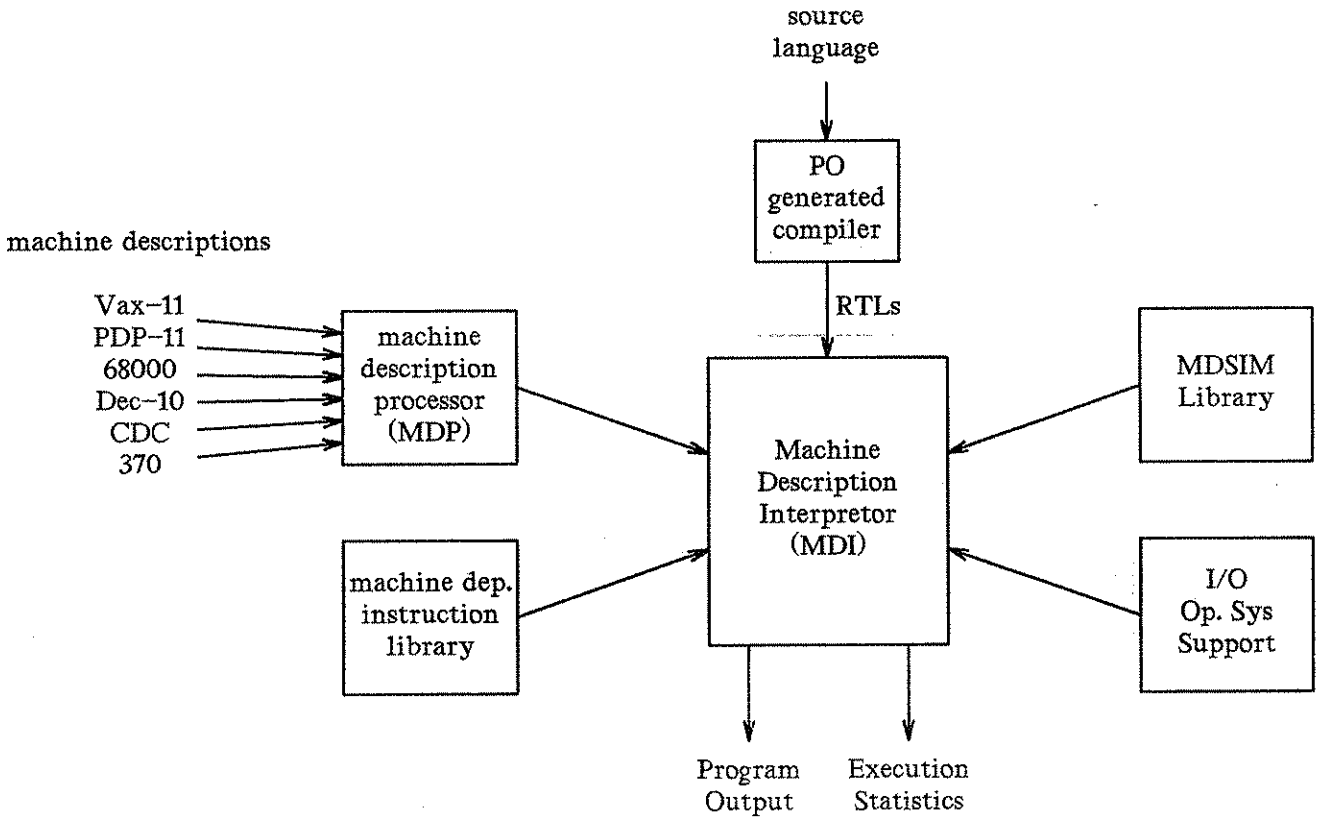

Figure 1. Schematic of the RT Interpreter

This library contains the hand–coded routines that implement the "hard" instructions that cannot be explicitly described using the description notation. The byte instructions for the DECsystem–10

†The choice of the C programming language was arbitrary. The techniques for constructing the subroutines could be applied to other high-level languages such as Ada, Pascal, or Modula-2.

would be in this class. Typically these routines are very short. For instance, the subroutine that implements the VAX–11 move character instruction, is 8 lines of C code. When this phase of the process is complete, C routines exist to execute all the instructions defined by the machine description. The machine description interpreter ('MDI') uses these routines to simulate an RTL program.

The first step of the simulation process is to read in the RTL program produced by the PO generated compiler. After the program has been read in, all jumps and subroutine calls in the program are located. The destinations of these jumps are found and a link is created between the jump and its destination. This is to avoid searching for the destination of a jump when the program is running.† Locating subroutine calls allows all routines that are called but are not part of the RTL program to be identified. At this point, the I/O and operating system support library is searched to see if any of the missing routines are contained in this library. If so they are loaded along with the RTL program. This allows the RTL program to call system routines for I/O and operating system service. At this point, the program can be executed.

Execution of the program is controlled by the user. Breakpoints can be set, and memory locations can be examined. Currently the user interface in the prototype simulator is crude. A clean user interface is planned for the production version of the simulator. The user can direct the simulator to collect various statistics. Currently only the number of instructions executed and the total number of memory references is recorded. Other statistics, such as execution counts for each instruction and for each addressing mode are planned for the production version.

The prototype simulator executes programs very fast. Running on the VAX–11/780 under UNIX and emulating the VAX–11/780 instruction set, the interpreter simulated 500 instructions per second. The number of instructions executed per second depends on the complexity of the machine being simulated. A less complicated machine was clocked at 650 instructions per second. The simulator owes its speed to the use of threaded code [Bell73]. Its flexibility is due to its general model of memory. These are discussed in the following sections.

†As a practical matter the destinations of all jumps cannot be determined at 'load' time. For example, the target of indexed jumps must be computed at execution time.

4.1 Threaded Code

As the MDI reads in a program, each instruction is identified, and a link to the subroutine implementing the instruction (previously constructed by the MDP) is created. In addition, the addressing modes used by the instruction are identified and links to the subroutines implementing these are created as well. At execution time, the interpreter follows the links to each subroutine. For example, the structure constructed for the program fragment below (taken from the VAX–11) is illustrated in Figure 2.

Execution of the program becomes essentially a series of subroutine calls. This threaded code structure has several advantages. Because the subroutine to implement an instruction is shared by all occurrences of that instruction, memory usage is reduced. Indeed, an entire compiler (3500 lines of source code) has been executed using the prototype interpreter. The interpreter is fast, which allows real programs to be simulated in a reasonable amount of time. As noted earlier, the prototype interpreter can execute approximately 500 instructions per second on a VAX–11/780 running UNIX. It is expected that an execution rate of 1000 instructions per second can be obtained once the interpreter is tuned and code included to facilitate the debugging of the interpreter is removed. The structure of the threaded code makes it easy to collect statistics about how many times a particular instruction or addressing mode is used.

4.2 Memory Model

The model of memory used by the interpreter is quite simple. Values are stored in a large associative memory that is accessed by a key. Normally a key is a combination of the address and the type of memory being accessed. This allows registers and main memory locations to be accessed using the same mechanism. Static memory is allocated and initialized as the program is read in by the interpreter. All other memory that is requested during program execution is allocated on demand. This, again, reduces the memory requirements necessary for the interpreter.

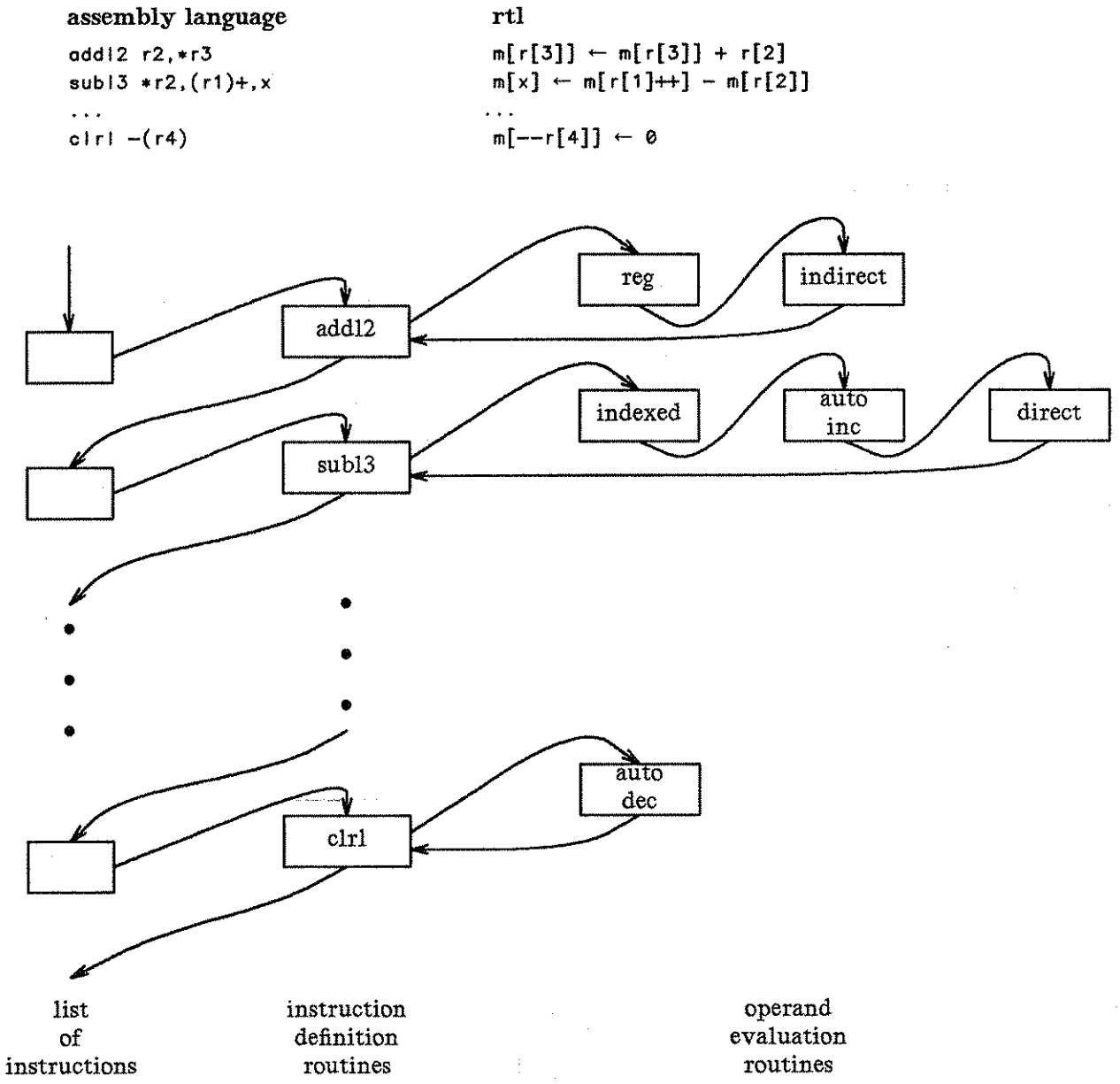

Figure 2. Threaded Code to Interpret Instructions

This simple model of memory can, however, create a problem when the same memory location can be accessed using two different names. For example, many contemporary architectures can access a byte at a certain address as well as a word starting at that same address. To handle this "aliasing" of memory locations, main memory is always accessed in some standard unit. A main memory access is converted to memory requests made up of these standard units. This models, in some sense, the way real architectures work.

The interpreter provides fast execution allowing machine code for real programs processing real data to be simulated. However, it must also be possible to translate high–level language programs into code that uses the instruction set. Ideally we would like to be able to use a variety of high–level languages. The machine description notation along with a retargetable code generator, PO, provides this capability.

5. PO

PO currently runs on a VAX–11/780 running UNIX. Using PO, cross–compilers have been built for seven machines, some in as few as three person days. The seven machines currently supported are the VAX–11, the PDP–11, the DEC–10, the CDC Cyber series machines, the Motorola 68000, the IBM 370, and the Intel 8080. Figure 3 shows a diagram of the relationship of PO to the other parts of the compiler.

Like many retargetable compilers [Korn80, Newe72], front ends for use with PO compile source code into an abstract machine code. Using an abstract machine code keeps the front end machine–independent. These intermediate languages ('ILs') are designed to be as simple and concise as possible. They include only the most general source language operations. For example, special–case instructions such as increment or decrement, are not included in the IL. Only the more general add and subtract operations are present.

The process of mapping the IL to the target machine is called code expansion. This stage expands the IL into register transfers ('RTLs'). For each target machine a code expander must be written that translates the source–dependent IL to machine–dependent RTLs. Writing code expanders is an ad hoc process. For traditional expanders, the quality of the final code depends on the ingenuity and expertise of the person writing the code expander. This is much less true for expanders used with PO. PO corrects inefficiencies, turning poor code into good code. Depending on the size of the source language IL, it takes a day or two to implement a code expander for a target architecture.

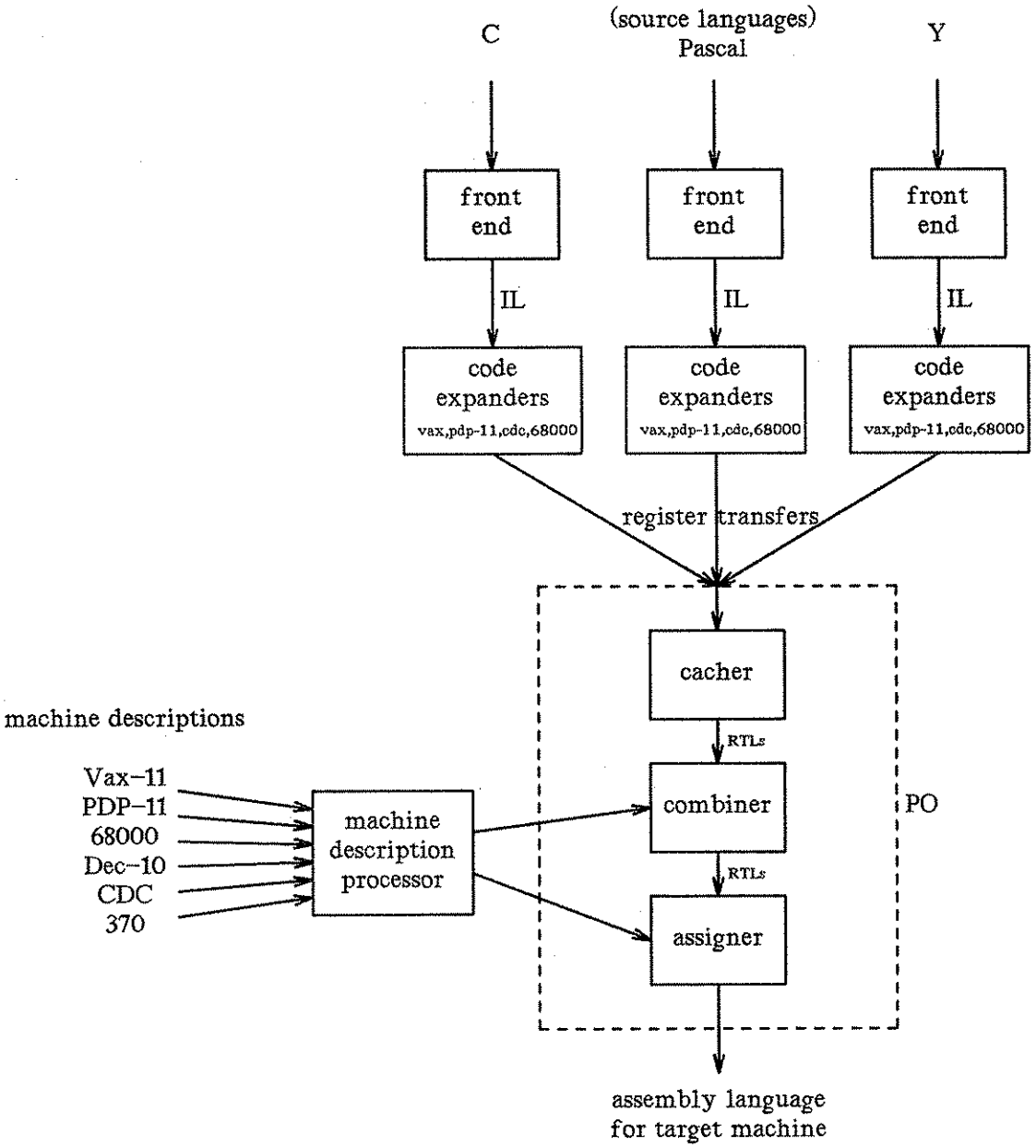

Figure 3. Schematic of the PO Compiler System

The last stage of the compiler is PO. It is comprised of three individual phases called "Cacher", "Combiner", and "Assigner". The "Cacher" phase eliminates common subexpressions in register transfers; the "Combiner" phase replaces sequences of register transfer instructions with equivalent singletons; and the "Assigner" phase translates them into assembly language. When the PO generated compiler is used to produce programs for the machine interpreter, the Assigner phase is

omitted. For more details on the operation of PO, see [Davi80, Davi84a, Davi84b].

Currently front-ends for Y [Hans81], a subset of C [Kern78], and a subset of Pascal [Jens75] are available. Thus programs written in those languages can be used to test the proposed instruction set. A front-end for Modula-2 [Wirt82] is planned. The ability to simulate code produced by compiling various high-level languages allows realistic simulations to be performed. For most programming environments, a computer must support a variety of high-level languages. Relying on simulations of code produced for just one language may produce an instruction set that is nicely tailored to that language but can not easily be used to implement or efficiently support other languages [Ditz80].

6. Implementation

Currently all the software described here runs on a VAX-11/780 running UNIX. The peephole optimizer, PO, and its retargetable compilers have been in use for approximately four years. Details of their implementation can be found in [Davi81, Davi84b].

The machine description processor that builds the interpreter is written in SNOBOL4 [Gris71]. It is 391 lines of code. It takes approximately five minutes of CPU time to process a 175 line machine description. The output is C routines that implement the addressing modes and instructions defined in the machine description. Some effort was taken to make the resulting routines portable so they could be run on other systems.

The MDSIM library contains routines that implement common operations. Currently the library only supports 32-bit and 16-bit arithmetic. Work is in progress to handle other word sizes. 64-bit and 8-bit arithmetic routines are planned.

Much of the implementation effort at this point is concentrated on improving the execution performance of the interpreter. Timing measurements of the interpreter show that a large percentage of the execution time is spent in the memory access routines. To reduce the time spent accessing the simulated machine's memory, we are planning to borrow an idea used in real memory systems — cache memories. The cache memory will hold the portions of the simulated memory in the

heaviest demand.

The other area of implementation is the user interface to the interpreter. Currently the user can set breakpoints and examine/change memory. A more sophisticated user interface is planned. This interface would allow the user to set conditional breakpoints, trace of subroutine calls and returns, and watch memory locations.

7. Applications

7.1 Instruction Set Design

The main application for this system is the design and evaluation of instruction sets. To effectively evaluate a proposed instruction set, the execution behavior must be observed for real programs. This is possible with this system as compilers can easily be built that generate code for the proposed instruction set. As Wulf notes:

> The problem arises from a "semantic clash" between the language and high-level instructions; by giving too much semantic content to the instruction, the machine designer has made it possible to use the instruction only in limited contexts [Wulf81].

This "clash" occurs because the instruction set designer often has a narrow view of how an instruction will be used. We believe this system will help designers develop better instruction sets.

The prototype system has been used by its developer to design an instruction set for a proposed machine. An iterative design technique was used. An instruction set was proposed, compilers for several languages were constructed, and a set of test programs were executed using the simulator. Based on the statistics gathered by the simulator and other analysis tools, the instruction set was modified. The process was repeated until the design was considered complete. An iteration of the design process took one to four days depending on the extent of the modifications to the instruction set. This fast cycle time is due both to the high level of the machine descriptions and the ability to quickly construct a compiler that used the experimental instruction set.

7.2 Software Testing

Just as with ISPS, this tool has other applications as well. It can be used in software testing. For example, one way to increase confidence in software is by thorough testing. A program can hardly be described as thoroughly tested if there are some parts of the code that, in fact, have never been executed. Through the use of the interpreter, it would be simple to determine which sections of code have or have not been executed. This could be done by marking each instruction as it is executed. After the program terminates, the list of instructions comprising the program could be scanned to locate sections of code that were not executed. Based on this information, new test cases could be devised that would cause those sections of code to be executed.

7.3 Performance Evaluation

Currently the machine description language does not have a facility for specifying the size or the timing of instructions. If this capability were added, the simulator could be used to obtain detailed timing information about a program. If size information about the instructions and addressing modes were available, the size of a program could be obtained. These measures are useful when comparing different instruction sets. Specification of the size and timing of instructions should be optional so that the user is required to specify this information only if statistics about program size and speed are desired.

7.4 Cache Memory Design

The interpreter could also be used to generate address traces. Address traces are used in trace-driven simulations to evaluate the behavior of cache memories [Smit82]. One way to generate an address trace is to interpretively execute a program and record every main memory location referenced by the program. The addresses are usually tagged to indicate the type of reference (e.g. read or write), and the size of the reference (e.g. byte, word, etc.). The use of the interpreter would permit interaction between the design of the memory system and the instruction set. This will become more important as more machines include special hardware and instructions to enhance the performance of the memory system [Patt83, Radi82]. In order for the proposed interpreter to gen-

erate address traces, the information about the size of instructions discussed in the preceding section would be necessary.

7.5 Retargeting Compilers

PO has dramatically simplified the construction of retargetable compilers. The use of the machine description interpreter can further reduce the time and expense of retargeting a compiler. Often when retargeting a compiler to a new machine, the new machine is not readily accessible. This may be because the machine is not located physically close to the site where the retargeting work is being done, or as is often the case with new machines, the machine is being used for other tasks and is not available for software development. With the use of the machine interpreter, much of the retargeting and testing of a compiler for a new machine can be done without access to the target machine. This can save time and development costs.

7.6 Automatic Generation of High-Level Language Interpreters

Many languages run in an interpreted environment. For example, many Pascal compilers generate Pcode, the abstract machine assembly language for Pascal [Berr78]. There are interpreted versions of many other high-level languages as well. Often it is desirable to have both a compiler and interpreter for a language. The interpreter is used for program development and debugging because it provides fast compilation at the expense of fast execution, while the compiler is used for production programs where fast execution is desired. These abstract machine interpreters are generally hand coded, and depending on the complexity of the abstract machine, they can require substantial development effort.

By writing a machine description of the abstract machine, the machine description processor can automatically construct an interpreter. For example, it would be quite easy to write a machine description for Pcode. By running this description through the machine description processor, we would have a Pcode interpreter that could be used with the Pcode compiler. This would save development time and costs. In addition, an automatically generated interpreter would be more reliable and easier to maintain.

7.7 Non–Applications

It is worthwhile to note some limitations of the machine descriptions and interpreter. The machine descriptions do not contain sufficient detail to be used to generate hardware, nor can they be considered a formal specification of an architecture. Our view is that this description notation should be used during the early stages of the design process. At that point it is important to be able to experiment and make changes quickly and easily. After the design stabilizes, the description can easily be converted into ISPS so that its capabilities can be used.

Also, because of the structure of the interpreter, it is not possible to simulate self–modifying code. This is seen as no great restriction as modern programming practices and principles discourage writing self–modifying code. Operating system and interrupt code can not be accommodated either. The machine descriptions do not describe the interrupt mechanism of the machine or the memory protection mechanisms.

8. Summary and Acknowledgements

This paper has presented a technique for the fast interpretation of machine descriptions. These descriptions are at a sufficiently high–level that they are easy to write, and simple to change. The machine description notation, the interpreter, and the compiler generation system provide an integrated environment for the design of instruction sets. While this is the major application of this system, it has a variety of other potential uses such as software testing and automatic generation of interpreters.

Walter Hansen provided many useful comments on the presentation of this work.

9. References

[Aho77] A. V. Aho and J. D. Ullman, *Principles of Compiler Design*, Addison–Wesley, Reading, MA, 1977.

[Barb79] M. R. Barbacci, W. B. Dietz and L. J. Szewerenko, Specification, Evaluation, and Validation of Computer Architectures Using Instruction Set Processor Descriptions, *Proceedings of the 4th International Symposium on Computer Hardware Description Languages*, Palo Alto, CA, October 1979, 14–20.

[Barb81] M. R. Barbacci, Instruction Set Processor Specifications (ISPS): The Notation and Its Application, *IEEE Transactions on Computers C-30*, 1 (January 1981), 24–40.

[Bell71] C. G. Bell and A. Newell, *Computer Structures: Readings and Examples*, McGraw–Hill, 1971.

[Bell73] J. R. Bell, Threaded Code, *Communications of the ACM 16*, 6 (June 1973), 370–372.

[Berr78] R. E. Berry, Experience with the Pascal P–Compiler, *Software - Practice & Experience 8*, 5 (September 1978), 617–627.

[Catt80] R. G. G. Cattell, Automatic Derivation of Code Generators from Machine Descriptions, *ACM Transactions on Programming Languages and Systems 2*, 2 (April 1980), 173–190.

[Crag83] H. Cragon, Executable Instruction Set Specification, *Computer Architecture News 11*, 1 (March 1983), 25–43.

[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, Register Allocation and Exhaustive Peephole Optimization, *Software - Practice and Experience 14*, 9 (September 1984), 857–866.

[Davi84b] 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.

[Ditz80] D. R. Ditzel and D. A. Patterson, Retrospective on High–Level Language Computer Architecture, *Proceedings of the Seventh Annual Symposium on Computer Architecture*, La Baule, France, May 1980, 97–104.

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

[Hans81] D. R. Hanson, The Y Programming Language, *SIGPLAN Notices 16*, 2 (February 1981), 59–68.

[Jens75] K. Jensen and N. Wirth, *Pascal User Manual and Report*, Springer–Verlag, New York, NY, 1975.

[Kern78] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice–Hall, Englewood Cliffs, NJ, 1978.

[Korn80] P. Kornerup, B. B. Kristen and O. L. Madsen, Interpretation and Code Generation Based on Intermediate Languages, *Software - Practice & Experience 10*, 8 (August 1980), 635–647.

[Newe72] M. C. Newey, P. C. Poole and W. M. Waite, Abstract Machine Modelling to Produce Portable Software, *Software - Practice & Experience 2*, (1972), 107–136.

[Oakl79] J. D. Oakley, *Symbolic Execution of Formal Machine Descriptions*, PhD Dissertation, Carnegie–Mellon University, Pittsburg, PA, 1979.

[Park79] A. C. Parker, D. E. Thomas, S. Crocker and R. G. G. Cattell, ISPS: A Retrospective View, *Proceedings of the 4th International Symposium on Computer Hardware Description Languages*, Palo Alto, CA, October 1979, 21–27.

[Patt81] D. A. Patterson and C. H. Sequin, RISC I: A Reduced Instruction Set VLSI Computer, *Proceedings of the Eighth Annual Symposium on Computer Architecture*, , May 1981, 443–457.

[Patt83] D. A. Patterson, P. Garrison, M. Hill, D. Lioupis, C. Nyberg, T. Sippel and K. V. Dyke, Architecture of a VLSI Instruction Cache for a Risc, *Proceedings of the 10th Annual International Symposium on Computer Architecture*, Stockholm, Sweden, June 1983, 108–116.

[Radi82] G. Radin, The 801 Minicomputer, *Proceedings of the Symposium on Architectural Support for Programming Languages*, Palo Alto, CA, April 1982, 39–47.

[Smit82] A. J. Smith, Cache Memories, *Computing Surveys 14*, 3 (September 1982), 473–530.

[Wirt82] N. Wirth, *Programming in Modula-2*, Springer–Verlag, New York, NY, 1982.

[Wulf81] W. A. Wulf, Compilers and Computer Architecture, *IEEE Computer 14*, 7 (July 1981), 41–38.