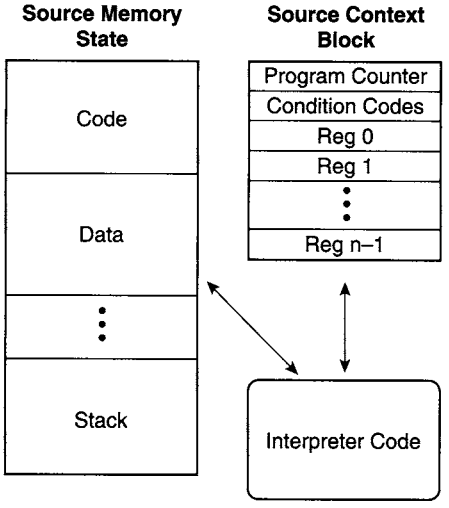


Figure 2.2 Interpreter Overview. *An interpreter manages the complete architected state of a machine implementing the source ISA.*

of the guest memory, including both program code and program data, is kept in a region of memory maintained by the interpreter. The interpreter's memory also holds a table we call the *context block*, which contains the various components of the source's architected state, such as general-purpose registers, the program counter, condition codes, and miscellaneous control registers.

A simple interpreter operates by stepping through the source program, instruction by instruction, reading and modifying the source state according to the instruction. Such an interpreter is often referred to as a *decode-and-dispatch* interpreter, because it is structured around a central loop that *decodes* an instruction and then *dispatches* it to an interpretation routine based on the type of instruction. The structure of such an interpreter is shown in Figure 2.3 for the PowerPC source ISA.

The main interpreter loop is depicted at the top of Figure 2.3, and routines for interpreting the *Load Word and Zero* and *ALU* instructions are shown below the main loop. The *Load Word and Zero* instruction loads a 32-bit word into a 64-bit register and zeroes the upper 32-bits of the register; it is the basic PowerPC load word instruction. Note that in this example routine (and others to follow), for brevity we have omitted any checks for memory addressing errors; these would be included in most VMs. Sections 3.3 and 3.4 describe emulation of the memory-addressing architecture more completely. The *ALU* "instruction" is actually a stand-in for a number of PowerPC instructions that have the same primary opcode but are distinguished by different

```
while (!halt && !interrupt) {
     inst = code[PC];
     opcode = extract(inst,31,6);
     switch(opcode) {
          case LoadWordAndZero: LoadWordAndZero(inst);
          case ALU: ALU(inst);
          case Branch: Branch(inst);
          . . .}
}


Instruction function list

LoadWordAndZero(inst){
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     displacement = extract(inst,15,16);
     if (RA == 0) source = 0;
     else source = regs[RA];
     address = source + displacement;
     regs[RT] = (data[address]<< 32) >> 32;
     PC = PC + 4;
}

ALU(inst){
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     RB = extract(inst,15,5);
     source1 = regs[RA];
     source2 = regs[RB];
     extended_opcode = extract(inst,10,10);
     switch(extended_opcode) {
          case Add: Add(inst);
          case AddCarrying: AddCarrying(inst);
          case AddExtended: AddExtended(inst);
          . . .}
     PC = PC + 4;
}
```

Figure 2.3 Code for Interpreting the PowerPC Instruction Set Architecture. *A decode-and-dispatch loop uses a switch statement to call a number of routines that emulate individual instructions. The extract(inst, i, j) function extracts a bit field of length j from inst, beginning at bit i.*

extended opcodes. For instructions of this type, two levels of decoding (via switch statements) are used. The decode-and-dispatch loop is illustrated here in a high-level language, but it is easy to see how the same routines could be written in assembly language for higher performance.

In Figure 2.3, the architected source program counter is held in a variable called PC. This variable is used as an index into an array that holds the source binary image. The word addressed by this index is the source instruction that needs to be interpreted. The opcode field of the instruction, represented by

the 6-bit field starting at bit 31,[1] is extracted using shift and mask operations contained in the `extract` function. The opcode field is used in a switch statement to determine a routine for interpreting the specific instruction. Register designator fields and immediate data in the instruction are decoded similarly using the `extract` function. The register designator fields are used as indices into the context block to determine the actual source operand values. The interpreter routine then emulates the operation specified by the source instruction. Unless the instruction itself modifies the program counter, as in a branch, the program counter must be explicitly incremented to point to the next sequential instruction before the routine returns back to the decode-dispatch loop of the interpreter.

The example of Figure 2.3 shows that, while the process of interpretation is quite straightforward, the performance cost of interpretation can be quite high. Even if the interpreter code were written directly in assembly language, interpreting a single instruction like the *Load Word and Zero* instruction could involve the execution of tens of instructions in the target ISA.

2.2 Threaded Interpretation

While simple to write and understand, a decode-and-dispatch interpreter can be quite slow. In this and subsequent sections, we will identify techniques to reduce or eliminate some of its inefficiencies. We begin by looking at *threaded interpretation* (Klint 1981).

The central dispatch loop of a decode-and-dispatch interpreter contains a number of branch instructions, both direct and indirect. Depending on the hardware implementation, these branches tend to reduce performance, particularly if they are difficult to predict (Ertl and Gregg 2001, 2003). Besides the test for a halt or an interrupt at the top of the loop, there is a register indirect branch for the switch statement, a branch to the interpreter routine, a second register indirect branch to return from the interpreter routine, and, finally, a branch that terminates the loop. By appending a portion of the dispatch code to the end of each of the instruction interpretation routines, as shown in Figure 2.4, it is possible to remove three of the branches just listed. A remaining branch is register indirect and replaces the switch statement branch found in the central dispatch loop. That is, in order to interpret the next instruction it is

---

1. PowerPC actually numbers the most significant bit (msb) 0; we use the convention that the msb is 31.

```
Instruction function list

LoadWordAndZero:
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     displacement = extract(inst,15,16);
     if (RA == 0) source = 0;
     else source = regs[RA];
     address = source + displacement;
     regs[RT] = (data(address)<< 32) >> 32;
     PC = PC + 4;
     If (halt || interrupt) goto exit;
     inst = code[PC];
     opcode = extract(inst,31,6);
     extended_opcode = extract(inst,10,10);
     routine = dispatch[opcode,extended_opcode];
     goto *routine;

Add:
     RT = extract(inst,25,5);
     RA = extract(inst,20,5);
     RB = extract(inst,15,5);
     source1 = regs[RA];
     source2 = regs[RB];
     sum = source1 + source2;
     regs[RT] = sum;
     PC = PC + 4;
     If (halt || interrupt) goto exit;
     inst = code[PC];
     opcode = extract(inst,31,6);
     extended_opcode = extract(inst,10,10);
     routine = dispatch[opcode,extended_opcode];
     goto *routine;
```

Figure 2.4 Two Threaded Interpreter Routines for PowerPC Code. *With threaded interpretation, the central dispatch loop is no longer needed.*

necessary to load the opcode of the next instruction, look up the address of the relevant interpreter routine using the dispatch table, and jump to the routine.

Figure 2.5 illustrates the differences in data and control flow between the decode-and-dispatch method and the threaded interpreter technique just described. Figure 2.5a shows native execution on the source ISA, Figure 2.5b shows the decode-and-dispatch method, and Figure 2.5c illustrates threaded interpretation. The centralized nature of the dispatch loop is evident from Figure 2.5b. Control flow continually exits from, and returns to, the central dispatch loop. On the other hand, with threaded interpretation (Figure 2.5c) the actions of the dispatch loop in fetching and decoding the next instruction are replicated in each of the interpreter routines. The interpreter routines are not subroutines in the usual sense; they are simply pieces of code that are "threaded" together.

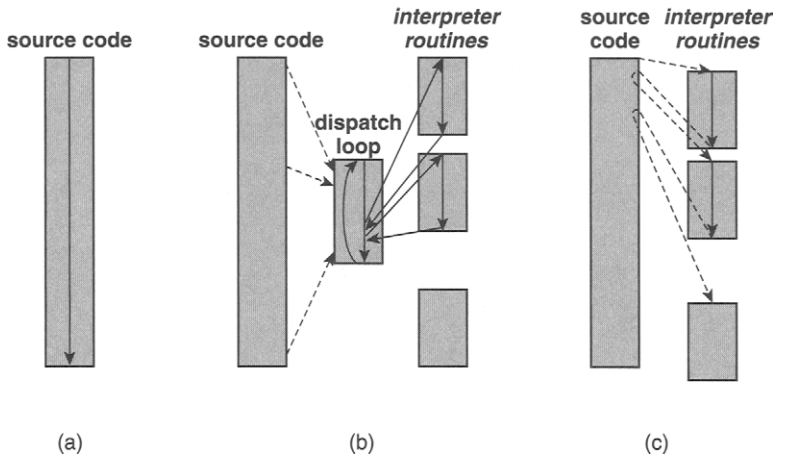


Figure 2.5 Interpretation Methods. *Control flow is indicated via solid arrows and data accesses with dotted arrows. The data accesses are used by the interpreter to read individual source instructions. (a) Native execution; (b) decode-and-dispatch interpretation; (c) threaded interpretation.*

A key property of threaded interpretation, as just described, is that dispatch occurs indirectly through a table. Among the advantages of this indirection is that the interpretation routines can be modified and relocated independently. Because the jump through the dispatch table is indirect, this method is called *indirect* threaded interpretation (Dewar 1975).

2.3 Predecoding and Direct Threaded Interpretation

Although the centralized dispatch loop has been eliminated in the indirect threaded interpreter, there remains the overhead created by the centralized dispatch table. Looking up an interpreter routine in this table still requires a memory access and a register indirect branch. It would be desirable, for even better efficiency, to eliminate the access to the centralized table.

A further observation is that an interpreter routine is invoked every time an instruction is encountered. Thus, when the same source instruction is interpreted multiple times, the process of examining the instruction and extracting its various fields must be repeated for each dynamic instance of the instruction. For example, as shown in Figure 2.3, extracting instruction fields takes several interpreter instructions for a *Load Word and Zero* instruction. It would appear to be more efficient to perform these repeated operations just once, save away the extracted information in an intermediate form, and then reuse it each time the instruction is emulated. This process, called *predecoding*, is

discussed in the following subsections. It will be shown that predecoding enables a more efficient threaded interpretation technique, *direct threaded interpretation* (Bell 1973; Kogge 1982).

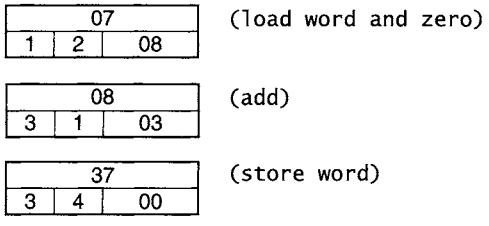
2.3.1 Basic Predecoding

Predecoding involves parsing an instruction and putting it in a form that simplifies interpretation. In particular, predecoding extracts pieces of information and places them into easily accessible fields (Magnusson and Samuelsson 1994; Larus 1991). For example, in the PowerPC ISA, all the basic ALU instructions, such as `and`, `or`, `add`, and `subtract`, are specified using a combination of the opcode bits and extended opcode bits. They all have the same primary opcode (31) and are distinguished by the extended opcode bits that appear at the low-order end of the instruction word, far away from the opcode. Predecoding can combine this information into a single operation code. Also, register specifiers can be extracted from the source binary and placed into byte-aligned fields so that they may be accessed directly with byte load instructions.

Basic predecoding of the PowerPC ISA is illustrated in Figure 2.6. Figure 2.6a contains a small PowerPC code sequence. This sequence loads a data item from memory and adds it to a register, accumulating a sum. The sum is then stored

```
lwz     r1, 8(r2)       ;load word and zero
add     r3, r3, r1      ;r3 = r3 + r1
stw     r3, 0(r4)       ;store word
```

(a)



(b)

Figure 2.6 Predecoded PowerPC Instructions. *The extended opcode and opcode of the add instruction are merged into a single predecoded opcode. (a) PowerPC source code. (b) PowerPC program in predecoded intermediate form.*

back to memory. Figure 2.6b is the same code, in a predecoded intermediate form. This predecoded format uses a single word to encode the operation, found by combining the opcode and function codes as discussed earlier. Consequently, these codes need *not* be the same as the source ISA opcodes, and in the example given they are not. A second predecode word is used for holding the various instruction fields, in a sparse, byte-aligned format. When immediate or offset data are given, a 16-bit field is available. Overall, this yields an intermediate instruction format that is less densely encoded than the original source instructions but more easily accessed by an interpreter.

The *Load Word and Zero* interpreter routine operating on the predecoded intermediate code of Figure 2.6 is given in Figure 2.7. Here, we predecode into an array of `instruction` structs, which is adequate for the example instructions but would be more elaborate for the full PowerPC ISA. In this example, the interpreter routines for the predecoded intermediate form are slightly simpler than the corresponding routines given earlier in Figure 2.3, and the benefits of predecoding for the PowerPC ISA appear to be relatively small. However, for CISC ISAs, with their many varied formats, the benefits can be greater. In addition, predecoding enables an additional performance optimization, direct threading, to be described in the next subsection.

```
struct instruction {
     unsigned long op;
     unsigned char dest;
     unsigned char src1;
     unsigned int src2;
     } code [CODE_SIZE]

     .
     .
     .
Load Word and Zero:
     RT = code[TPC].dest;
     RA = code[TPC].src1;
     displacement = code[TPC].src2;
     if (RA == 0) source = 0;
     else source = regs[RA];
     address = source + displacement;
     regs[RT] = (data[address]<< 32) >> 32;
     SPC = SPC + 4;
     TPC = TPC + 1;
     If (halt || interrupt) goto exit;
     opcode = code[TPC].op;
     routine = dispatch[opcode];
     goto *routine;
```

Figure 2.7 Threaded Interpreter Code for PowerPC *Load Word And Zero* Instruction After Predecoding.

Because the intermediate code exists separately from the original source binary, a separate target program counter (TPC) is added for sequencing through the intermediate code. However, the source ISA program counter (SPC) is also maintained. The SPC keeps the correct architected source state, and the TPC is used for actually fetching the predecoded instructions. In general, with CISC variable-length instructions, the TPC and SPC values at any given time may bear no clear relationship, so it is necessary to maintain them both. With fixed-length RISC instructions, however, the relationship can be relatively easy to calculate, provided the intermediate form is also of fixed length.

2.3.2 Direct Threaded Interpretation

Although it has advantages for portability, the indirection caused by the dispatch table also has a performance cost: A memory lookup is required whenever the table is accessed. To get rid of the level of indirection caused by the dispatch table lookup, the instruction codes contained in the intermediate code can be replaced with the actual addresses of the interpreter routines (Bell 1973). This is illustrated in Figure 2.8.

Interpreter code for direct threading is given in Figure 2.9. This code is very similar to the indirect threaded code, except the dispatch table lookup is removed. The address of the interpreter routine is loaded from a field in the intermediate code, and a register indirect jump goes directly to the routine. Although fast, this causes the intermediate form to become dependent on the exact locations of the interpreter routines and consequently limits portability. If the interpreter code is ported to a different target machine, it must be regenerated for the target machine that executes it. However, there are programming techniques and compiler features that can mitigate this problem to some extent. For example, the gcc compiler has a unary operator (&&) that takes the address

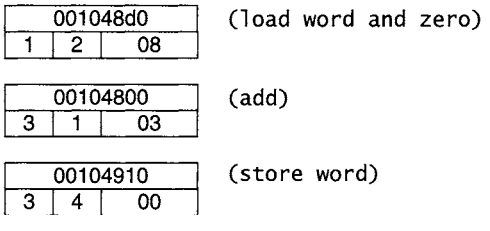


Figure 2.8 Intermediate Form for Direct Threaded Interpretation. *The opcode in the intermediate form is replaced with the address of the interpreter routine.*

```
Load Word and Zero:
     RT = code[TPC].dest;
     RA = code[TPC].src1;
     displacement = code[TPC].src2;
     if (RA == 0) source = 0;
     else source = regs[RA];
     address = source + displacement;
     regs[RT] = (data[address]<< 32) >> 32;
     SPC = SPC + 4;
     TPC = TPC + 1;
     If (halt || interrupt) goto exit;
     routine = code[TPC].op;
     goto *routine;
```

Figure 2.9 Example of Direct Threaded Interpreter Code.

of a label. This operator can then be used to generate portable direct threaded code by finding the addresses of the labels that begin each of the interpreter routines and placing them in the predecoded instructions. The interpreter can also be made relocatable by replacing the absolute routine addresses with relative addresses (with respect to some routine base address).

2.4 Interpreting a Complex Instruction Set

Thus far, when describing basic interpretation techniques, it has been useful to center the discussion on fairly simple instruction sets. A RISC ISA, the PowerPC, was used in our examples. Similarly, virtual instruction sets, such as Pascal P-code and Java bytecodes — to be discussed in Chapter 5 — are designed specifically for emulation and can be interpreted using the techniques described above in a straightforward manner. In practice, however, one of the most commonly emulated instructions sets is not a RISC or a simple virtual ISA; rather it is a CISC — the Intel IA-32. In this section we consider the additional aspects (and complexities) of interpretation that are brought about by a CISC ISA, using the IA-32 as the example.

One of the hallmarks of a modern RISC ISA such as the PowerPC is the regular instruction formats. That is, all instructions have the same length, typically 32 bits, and the instruction formats are fairly regular, for example, the register specifiers usually appear in the same bit positions across instruction formats. It is this regularity that makes many of the steps of interpretation straightforward. For example, the interpreter can extract the opcode and then immediately dispatch to the indicated instruction interpreter routine. Similarly, each of the instruction interpretation routines can extract operands and complete the emulation process simply.

Many CISC instruction sets, on the other hand, have a wide variety of formats, variable instruction lengths, and even variable field lengths. In some ISAs the variability in instruction formats was intended to increase code density and "orthogonality" of the instruction set. The VAX ISA is a good example (Brunner 1991); in the VAX, every operand can be specified with any of the many addressing modes. In other ISAs, the variability reflects the evolution of the instruction set over time, where a number of extensions and new features have been added, while maintaining compatibility with older versions. The IA-32 is a good example of this evolutionary process. The IA-32 started as an instruction set for 16-bit microcontroller chips with physical addressing and dense instruction encodings and eventually evolved into a high-performance, 32-bit general-purpose processor supporting virtual memory. This evolutionary process continues, and it has recently been further extended to 64 bits.

2.4.1 Interpretation of the IA-32 ISA

Figure 2.10 illustrates the general form of an IA-32 instruction. It begins with from zero to four prefix bytes. These indicate if there is repetition for string instructions and/or if there are overrides for addressing segments, address sizes, and operand sizes. Then after the prefix byte(s) (if any), there is an opcode byte, which may be followed by a second opcode byte, depending on the value of the first. Next comes an optional addressing-form specifier ModR/M. The specifier is optional, in the sense that it is present only for certain opcodes and generally indicates an addressing mode and register. The SIB byte is present for only certain ModR/M encodings, and it indicates a base register, an index register, and a scale factor for indexing. The optional, variable-length displacement field is present for certain memory-addressing modes. The last field is a variable-length immediate operand, if required by the opcode.

Because of all the variations and the presence or absence of some fields, depending on the values in others, a straightforward approach to interpreting a CISC ISA, and the IA-32 ISA in particular, is to divide instruction interpretation into two major phases, as illustrated in Figure 2.11. The first phase scans and decodes the various instruction fields. As it does so, it fills in fields of a

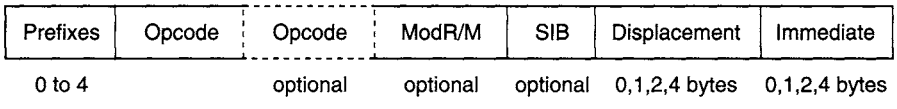


Figure 2.10 General Format for IA-32 Instructions.

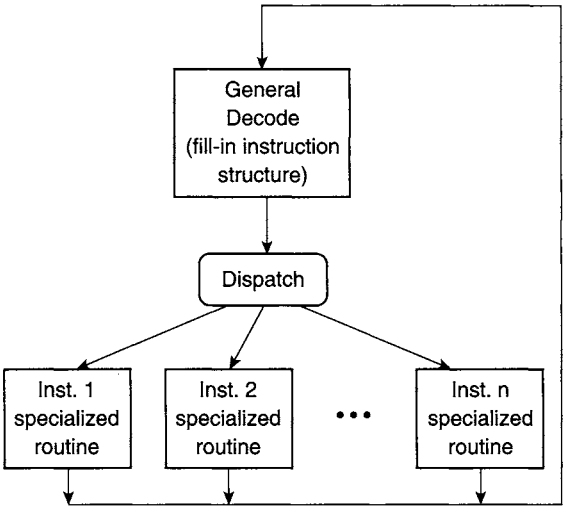


Figure 2.11 Program Flow for a Basic CISC ISA Interpreter.

general instruction template. This template, in essence, contains a superset of the possible instruction options. Then there is a dispatch step that jumps to routines specialized for each instruction type. These routines emulate the specified instruction, reading values from the relevant instruction template fields as needed.

Figure 2.12 is a three-page figure that contains pseudo C code for such an interpreter, styled after the approach used in the Bochs free software IA-32 interpreter (Lawton). Not all procedures used in the example are given, but where their code is not present, they are given mnemonic names that summarize their function. Interpretation is focused on an instruction template, the structure `IA-32instr`, which is defined at the top of Figure 2.12a. The major CPU interpreter loop is at the bottom of Figure 2.12b. This loop begins interpreting an instruction by filling in the instruction template. Included in the instruction template is a pointer to an instruction interpreter routine. After the template is built, the CPU loop uses the pointer to jump to the indicated routine. Some instructions can be repeated (based on a prefix byte), and this is determined by the "`need_to_repeat`" test.

The `IA-32instr` structure consists of the opcode (up to two bytes), a mask that collects the prefixes (up to a total of 12 possible prefixes), a value that contains the instruction length and a pointer to the instruction interpretation routine. Then there are a number of substructures used for collecting operand information. As is the case with the structure as a whole, these are defined to contain a superset of the operand information for all the instructions. The total size of the structure is on the order of 6 words.