Chapter Two

Emulation: Interpretation and Binary Translation

Many virtual machine implementations are based on emulation. We define *emulation* as the process of implementing the interface and functionality of one system or subsystem on a system or subsystem having a different interface and functionality. For example, a VT100 terminal emulator running inside a window on a PC presents to the user an interface and functionality that is nearly identical to that of a real VT100 terminal. In fact, taken in this general sense, one could argue that virtualization itself is simply a form of emulation. In this chapter, however, we give emulation a narrower meaning by applying it specifically to instruction sets.

Instruction set emulation is a key aspect of many virtual machine implementations because the virtual machine must support a program binary compiled for an instruction set that is different from the one implemented by the host processor(s). For example, Intel IA-32 program binaries are more widely available than binaries for any other instruction set. Consequently, a user may wish to use a virtual machine to execute an IA-32 program binary on some other platform, for example, an Apple Macintosh, which incorporates a PowerPC processor. For high-level virtual machines, binary classes (using the Java terminology) incorporate a stack-based, bytecode instruction set that can be emulated on any of a number of different host platforms.

In terms of instruction sets, emulation allows a machine implementing one instruction set, the *target* instruction set, to reproduce the behavior of software compiled to another instruction set, the *source*

instruction set. This is illustrated in **Figure 2.1**. Note that we use the terms *source* and *target* specifically for instruction set emulation, and we use the terms *guest* and *host* when referring to complete virtual machine environments and supporting platforms (which often involve more than just ISAs). The reader should also be aware that the literature often does not use consistent terminology when describing guest/host and source/target relationships.

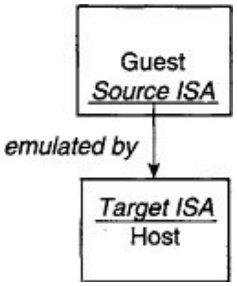


Figure 2.1 Terms Describing the Emulation Process. *Emulation allows a guest to support a source instruction set while running on a host platform executing a target instruction set.*

For many virtual machine applications, it is important that the emulation of the instruction set be performed efficiently. The lower the overhead of emulation, the more attractive the virtual machine will be. This chapter will focus on techniques for emulation of conventional instruction sets that already have hardware implementations rather than those designed specifically for virtual machine implementation, such as the Java bytecode instruction set. The latter *virtual instruction sets* can be implemented with similar emulation techniques as conventional instruction sets, but they also have special properties and therefore can take advantage of other emulation techniques. High-level language VM-specific instruction sets will be discussed in **Chapter 5**.

A complete ISA consists of many parts, including the register set and memory architecture, the instructions, and the trap and interrupt architecture. A virtual machine implementation is usually concerned with all aspects of ISA emulation. In this chapter, however, we focus primarily on emulating the operation of user-level instructions in the absence of exceptions (traps and interrupts). The emula-

tions in the absence of exceptions (traps and interrupts). The emulation of the memory addressing architecture, traps, interrupts, and other features are discussed in this chapter only as is necessary to understand instruction emulation. In **Chapters 3** and **4**, we extend the discussion to emulation of memory architecture, traps, and interrupts in the context of process virtual machines. In later chapters (primarily **Chapters 7** and **8**) we discuss emulation of system instructions and other ISA issues that are specific to system virtual machines.

To some degree, instruction emulation techniques can be applied to every type of VM we discuss in this book. Although we are interested primarily in the case where the source and target instruction sets are different, there are some virtual machine applications where the instruction sets are the same. In these situations, emulation may not be performed, strictly speaking, but the very same techniques are used for other purposes. One such application is same-ISA dynamic binary optimization, where a program binary is optimized at run time on the same platform for which it was originally compiled. Another example is in system VMs where the VMM must control and manage the execution of certain privileged guest OS instructions, and emulation-like techniques are used for this purpose.

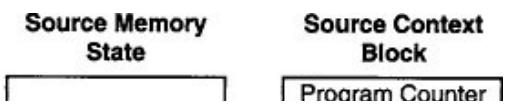
Instruction set emulation can be carried out using a variety of methods that require different amounts of computing resources and offer different performance and portability characteristics. At one end of the spectrum is the straightforward technique of *interpretation,* while on the other is *binary translation.* Interpretation involves a cycle of fetching a source instruction, analyzing it, performing the required operation, and then fetching the next source instruction—all in software. Binary translation, on the other hand, attempts to amortize the fetch and analysis costs by translating a block of source instructions to a block of target instructions and saving the translated code for repeated use. In contrast to interpretation, bi-

nary translation has a bigger initial translation cost but a smaller execution cost. The choice of one or the other depends on the number of times a block of source code is expected to be executed by the guest software. Predictably, there are techniques that lie in between these extremes. For example, *threaded interpretation* eliminates the interpreter loop corresponding to the cycle mentioned earlier, and efficiency can be increased even further by predecoding the source instructions into a more efficiently interpretable intermediate form.

2.1 Basic Interpretation

Interpreters have a long and rich history. Some programming languages, e.g., the functional language LISP, rely on an interpreted implementation. Perl is a widely used language that is commonly implemented through interpretation. The FORTH language is probably better known for its “threaded code” interpretation model than for any other feature. A good reference is **Debaere and Van Campenhout (1990)**. In this chapter, however, we are more interested in applying interpretation techniques to program binaries (machine code) rather than high-level languages.

In our narrower context, an interpreter program emulates and operates on the complete architected state of a machine implementing the source ISA, including all architected registers and main memory (**Figure 2.2**). The image 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.



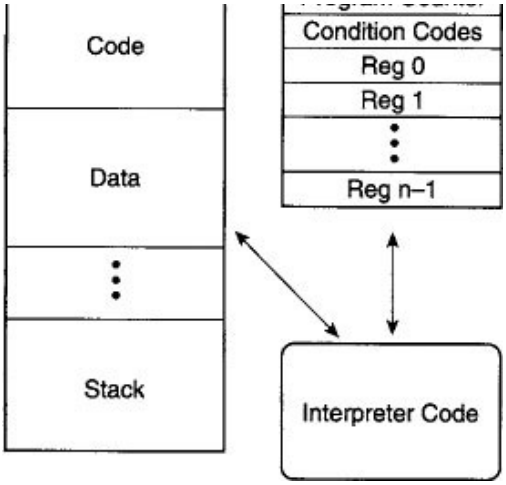


Figure 2.2 Interpreter Overview. *An interpreter manages the complete architected state of a machine implementing the source ISA.*

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 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 lan-

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.

```
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.*

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 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.

```
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.*

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 2.5**a shows native execution

prefer 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.

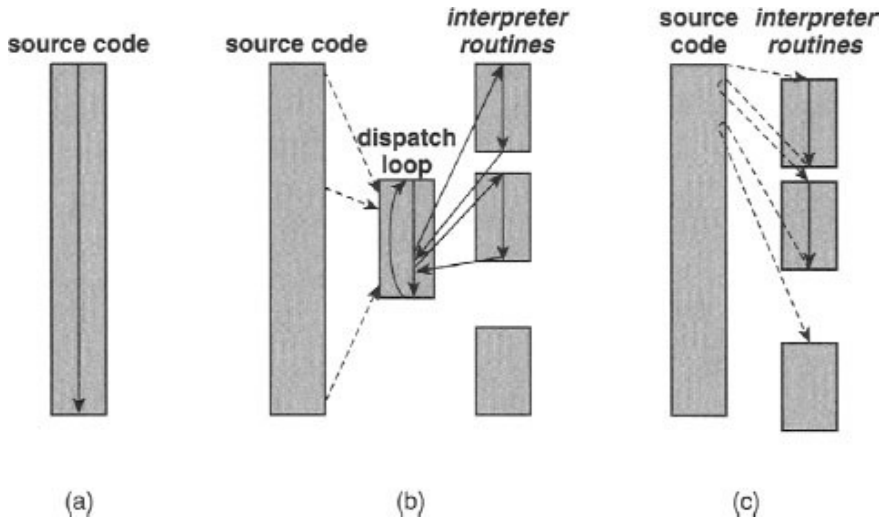


Figure 2.5 Interpretation Methods. *Control flow is indicated via solid arrows and data accesses with dotted arrows. The data accesses are used by the interpreter to read individual source instructions. (a) Native execution; (b) decode-and-dispatch interpretation; (c) threaded interpretation.*

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

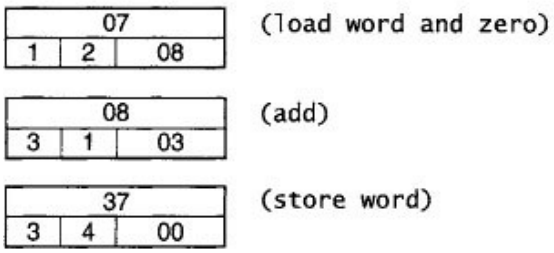
Predecoding involves parsing an instruction and putting it in a form that simplifies interpretation. In particular, predecoding extracts pieces of information and places them into easily accessible fields (**Magnusson and Samuelsson 1994**; **Larus 1991**). For example, in the PowerPC ISA, all the basic 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 opera-

tion 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 2.6**a 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 back to memory. **Figure 2.6**b 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.

```
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.*

The *Load Word and Zero* interpreter routine operating on the prede-

coded 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 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).

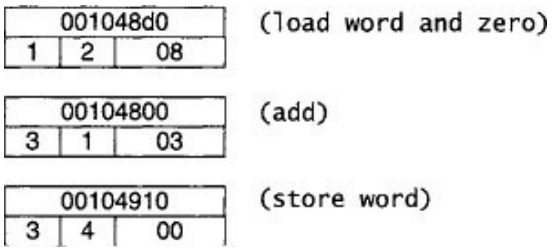


Figure 2.8 Intermediate Form for Direct Threaded Interpretation. *The opcode in the intermediate form is replaced with the address of the interpreter routine.*

```
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.

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 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.

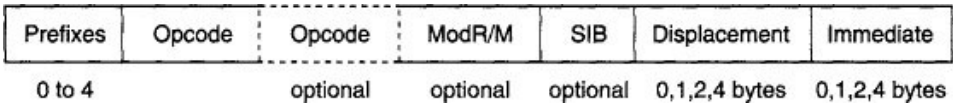


Figure 2.10 General Format for IA-32 Instructions.

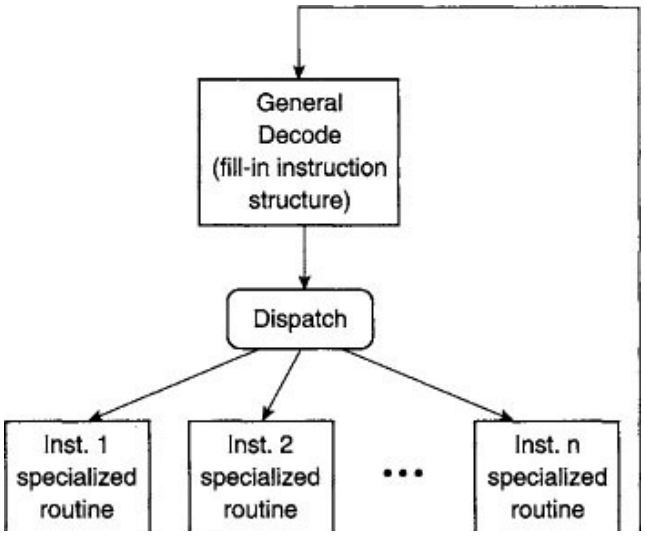




Figure 2.11 Program Flow for a Basic CISC ISA Interpreter.

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.

The code that fills in the template structure is given at the top of **Figure 2.12b**. This code scans an instruction from left to right, first looking for prefix bytes and filling in the prefix mask. It then determines whether the opcode is one or two bytes, and uses the opcode to do a table lookup to find a decode action and a pointer to the routine that will eventually be used to interpret the instruction. The lookup table is the second of the major data structures used for interpretation, shown at the bottom of **Figure 2.12a**. The lookup table

returns a pair: <DecodeAction, InterpreterFunctionPointer>. DecodeAction contains addressing mode information, the presence or absence of immediate values, etc., and the InstructionFunctionPointer is a pointer to a specific interpreter routine. Finally, the operand specifiers are decoded and displacement and immediate values are filled in.

```
struct IA-32instr {
  unsigned short opcode;
  unsigned short prefixmask;
  char ilen; // instruction length.

  InterpreterFunctionPointer execute; // semantic routine for this instr.

  struct {
    // general address computation: [Rbase + (Rindex << shmt) + displacement]
    char mode;     // operand addressing mode, including register operand.
    char Rbase;    // base address register
    char Rindex;   // index register
    char shmt;     // index scale factor
    long displacement;
  } operandRM;

  struct {
    char mode;     // either register or immediate.
    char regname;  // register number
    long immediate;// immediate value
  } operandRI;
} instr;


//
// BIG fetch_decode table indexed by the opcode
//
IA-32OpcodeInfo_t IA-32_fetch_decode_table[] =
{
  { DecodeAction, InterpreterFunctionPointer},
  { DecodeAction, InterpreterFunctionPointer},
  { DecodeAction, InterpreterFunctionPointer},
  ..................

};
```

Figure 2.12a Major Data Structures for an IA-32 Interpreter. *Instruction template and decode table.*

```
IA-32instr
IA-32_FetchDecode(PC){
    fetch_ptr = PC;

    // 1. parse prefixes
    byte = code[++fetch_ptr];
    while (is_IA-32_prefix(byte)) {
        add_prefix_attribute(byte, instr);
        byte = code[++fetch_ptr];
    }

    // 2. parse opcode
    instr.opcode = byte;       // its code[fetch_ptr];
    if (instr.opcode == 0x0f){
```

```
        instr.opcode = 0x100 | code[++fetch_ptr]; // 2 Byte opcode.
    }

    // 3. Table Look up based on opcode to find action and function pointer.
    decode_action = IA-32_fetch_decode_table[instr.opcode].DecodeAction;
    instr.execute =
            IA-
32_fetch_decode_table[instr.opcode].InterpreterFunctionPointer;
    // Semantic routines for IA-32 instrs, e.g., ADD_RX_32b(IA-32instr i);

    // 4. Operand Resolution -- setup the operandRI and operandRM fields above.
    if (need_Mod_RM(decode_action)) {
        parse_Mod_RM_byte(instr);
        if (need_SIB_byte(instr->operandRM.mode)) fetch_SIB_byte(instr);
        if (need_displacement(instr->operandRM.mode)) fetch_displacement(instr);
    }
    if (need_immediate(decode_action)) fetch_immediate(instr);

    // 5. bookkeeping and return.
    instr.ilen = bytes_fetched_for_this_instr;
    return instr;
}

void cpu_loop()
{
    while (!halt) {
        instr = IA-32_FetchDecode(PC);
        if (!IA-32_string_instruction) {
            instr.execute();
        }
        else {
            while(need_to_repeat(instr.prefixmask))
                instr.execute();
                handle_asyn_event();    // i.e. an interrupt
            }
        }
        PC = PC + instr.ilen;
        handle_asyn_event();
    }
}
```

Figure 2.12b Template-Filling Routine and Major Emulation Loop for IA-32 Interpreter.

Instruction function list

```
// ADD: register + Reg/Mem --> register
void
ADD_RX_32b(IA-32instr instr)      // X means: either Reg or Mem{
    unsigned op1_32, op2_32, sum_32;
    op1_32 = IA-32_GPR[instr.operandRI.regname];
    if (mem_operand(instr.operandRM.mode)) {
        unsigned mem_addr = resolve_mem_address(instr);
        op2_32 = virtual_mem[mem_addr];
}
    else {
        op2_32 = IA-32_GPR[instr.operandRM.Rbase];
    }
    sum_32 = op1_32 + op2_32;
    IA-32_GPR[instr.operandRI.regname] = sum_32;
    SET_IA-32_CC_FLAGS(op1_32, op2_32, sum_32, IA-32_INSTR_ADD32);
}


void
ADD_XR_32b(IA-32instr instr)
{
```

```
}


void
ADD_RI_32b(IA-32instr instr)
{


}
```

Figure 2.12c Instruction Interpretation Routines for an IA-32 Decode-and-Dispatch Interpreter.

Figure 2.12c illustrates the instruction function list, with one (of the many) instructions, a 32-bit add instruction, being given in detail. This interpreter routine uses information from the IA-32instr template as it executes. After interpretation is complete, the CPU loop increments the program counter and proceeds to the next instruction.

A basic decode-and-dispatch interpreter as just described is well structured and relatively easy to understand, but it will be quite slow. One reason for this slowness is its generality. That is, it does all the sequential decoding first, covering all possible cases, before it proceeds to an instruction interpretation routine. A more efficient interpreter can be built around the principle: “make the common case fast.” For the IA-32 ISA, the common case is: (1) no prefix bytes, (2) a single-byte opcode, (3) simple operand specifiers, often just registers. Based on these observations, an interpreter can be structured along the lines illustrated in **Figure 2.13**. This interpreter first dispatches to a routine based on the first instruction byte (alternatively, two bytes could be used, with a much larger dispatch table). Then these simpler, common instructions are interpreted immediately via specialized routines that decode the remainder of the instruction bytes. For the less common cases, there are more complex interpretation routines that may share routines for the more involved operations. If the first byte should happen to be a prefix, then its value can be recorded, with control returning to the dispatch code. With this implementation, a sequence of simple IA-32 instructions will be emulated relatively quickly, very much like an

equivalent sequence of RISC instructions.

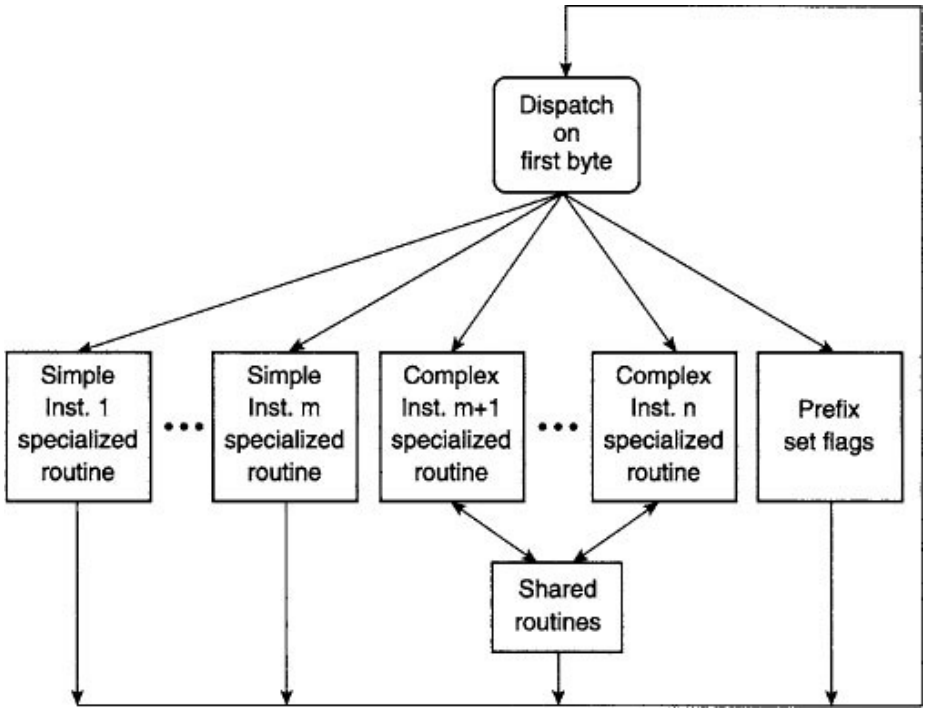


Figure 2.13 Interpreter Based on Optimization of Common Cases.

2.4.2 Threaded Interpretation

Recall that with threaded interpretation, decode-and-dispatch code is appended to the end of every instruction interpretation routine. For a RISC, as we saw earlier, this is a relatively small amount of code and results in the removal of several branch instructions. However, if the complex decode routine for the IA-32 ISA, illustrated in **Figure 2.12b**, were to be appended to every instruction interpretation routine, the interpreter would become very large, and any performance improvement would be relatively small. Consequently, to implement a threaded interpreter for a CISC ISA, one should append a simple decode-and-dispatch routine, optimized for the common cases, to each instruction interpretation routine and use a centralized decode-and-dispatch routine for the more complex cases. This is illustrated in **Figure 2.14**, where simple instructions are threaded from one to the next but where, when a complex case is

encountered, a centralized decode routine is executed, followed by a dispatch to an instruction interpretation routine. In effect this is a hybrid of the decode-and-dispatch and threaded methods.

If predecoding and direct threaded interpretation are to be used for a CISC ISA, there are two significant problems. The first is that a general, fixed-length predecoded instruction format would probably look very similar to the instruction template IA-32instr used in the decode-and-dispatch interpreter. This would lead to a very large predecoded program. In this case, each instruction consumes about six words. An alternative is to use a small number of intermediate predecoded forms with some specialization for instruction types. Or one can predecode a single CISC instruction into multiple, simple, predecoded instruction forms. This is very similar to binary translation to a RISC ISA (binary translation is covered in the next section).

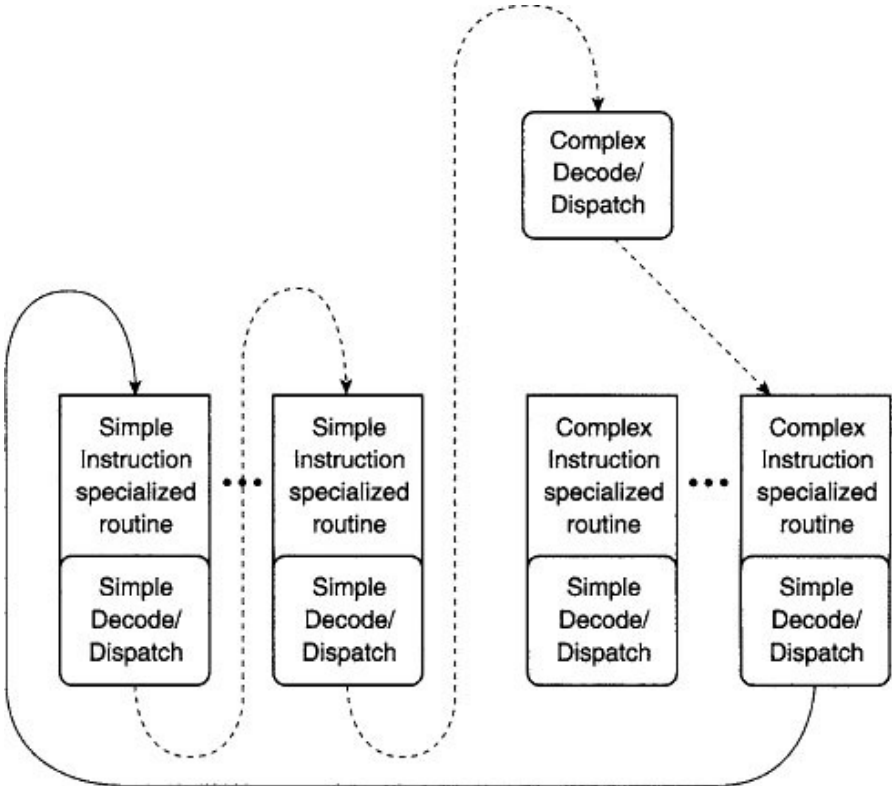


Figure 2.14 Threaded Interpreter for a CISC ISA.

The second significant problem that occurs with predecoding most

CISC ISAs is the need to perform *code discovery*. Ordinarily, the predecoding step operates on a program binary *before* instructions are interpreted. With variable-length instructions, however, it is not always practical, (or may be even possible) to scan a binary and correctly identify all instruction boundaries or, in some cases, to separate data from instructions. The code-discovery problem also occurs with binary translation and is described in more detail in **Section 2.6**. Because of the code-discovery problem, predecoding a conventional CISC ISA becomes an iterative, two-stage process, with simple decode-and-dispatch interpretation first being used to discover instructions, followed by predecoding into an intermediate form and direct threaded interpretation. Overall, this is very similar to the staged emulation methods incorporating binary translation, to be discussed later in this chapter.

Based on the foregoing, one could conclude that predecoding a CISC ISA is so similar to binary translation that one should simply perform binary translation and be done with it. However, predecoding, as opposed to binary translation, has one major advantage: better portability. With binary translation, there must be a code generator specialized for every target ISA, although retargetable binary translators have been developed (**Cifuentes, Lewis, and Ung 2002**; **Scott et al. 2003**). With predecoding, the intermediate form remains largely platform independent, and the interpreter routines can be written in a portable high-level language. Hence, one can combine a simple RISC-like intermediate form with fast direct threaded interpreter routines and a staged interpretation strategy, to arrive at a high-performance yet portable interpreter for a CISC ISA.

2.4.3 A High-Performance IA-32 Interpreter

We conclude this section on CISC interpretation with an example that illustrates features of an IA-32 decode-and-dispatch interpreter optimized for speed. The example is based roughly on a description of an IA-32 interpreter used as part of the DEC/Compaq FX!32 sys-

tem (**Chernoff et al. 1998**; **Hookway and Herdeg 1997**). In the FX!32 system, the 32-bit IA-32 architecture is emulated on a 64-bit Alpha platform. This particular interpreter is written in assembly language, which makes it nonportable but allows very highly optimized code. In this example, we use assembly code similar to that described for the FX!32; however, it is written using 64-bit PowerPC code rather than Alpha code. The interpreter uses a decode-and-dispatch style that is specialized for the common cases.

The main loop uses the technique of *software pipelining* (**Lam 1988**) to interleave different phases of program interpretation in the same inner loop in order to reduce execution time. The following are the key registers used in the dispatch loop, along with a brief description of their usage.

r1 and ***r2*** hold a byte stream of prefetched IA-32 instructions; a minimum of 8 bytes of prefetched instructions are kept.

r3 holds the instruction buffer length; this register is used not in the main dispatch loop but in the buffer maintenance routine.

r4, at the top of the loop, holds the length of the current instruction (in bits).

r5 is loaded with the upper two bytes of the next instruction; these two bytes contain sufficient information to determine both the opcode and the length of the next instruction.

r6 holds a pointer to the interpretation routine for the current instruction.

r7 is loaded with a pointer to the interpretation routine for the next instruction.

r8 points to the dispatch table of interpreter routines; each entry is 8

bytes.

r9 points to a table of IA-32 instruction lengths; each entry is 1 byte.

r10 holds an instruction pointer that is used for prefetching from the IA-32 code stream.

r11 is loaded with 8 bytes of prefetched IA-32 instructions; these prefetched instructions are used by buffer maintenance code (not shown).

The decoding of instructions is layered, with the common cases being handled more efficiently. Most instructions in the IA-32 instruction set have 6 bytes or fewer. Hence the dispatch loop is optimized for this case. **Figure 2.15** shows the central dispatch loop. The first two instructions in the routine perform a check on the instruction length; if the length is greater than 6, then it requires special handling and there is a branch to routine long_inst (not shown). The next three instructions of the routine extract the first 2 bytes of the *next* IA-32 instruction to be interpreted and convert it to a double-word (8 bytes) offset placed in r5. As mentioned earlier, the first 2 bytes of an IA-32 instruction contain information about the length of an instruction as well as its opcode.

```
loop:
     cmpwi   cr0,r4,48          ;compare length (in r4) with 48 (bits)
     bgt     cr0, long_inst     ;branch to long_inst if length > 48
     sld     r5,r1,r4           ;shift instruction I+1 into r5
     extrdi  r4,r5,16,0         ;extract upper 2 bytes of I+1 from "buffer"
     sldi    r5,r4,3            ;multiply by 8: convert to double word offset
     lbzx    r4,r4,r9           ;look up instruction length for I+1
     ldx     r7,r5,r8           ;look up interpreter routine for I+1
     ld      r11,0(r10)         ;prefetch next 8 bytes
     mtctr   r6                 ;move I's interpreter routine pointer into ctr
     bctrl                      ;dispatch I; branch to ctr and link
     mr      r6,r7              ;move register; to maintain software pipeline
     b       loop               ;continue loop
```

Figure 2.15 Main Dispatch Loop for High-Performance IA-32 Interpreter Code Written in 64-Bit PowerPC Assembly Language.

The next three instructions perform a series of memory loads. The first two are table lookups to find the length of the next source in-

struction and a pointer to its interpreter routine. In both cases the tables contain 64K entries because they are indexed with the upper 2 IA-32 instruction bytes. This approach leads to some redundancy in the table because 2 bytes are not always required to determine the operation type and instruction length, but a direct table lookup reduces the number of instructions required significantly. The third load instruction prefetches the next 8 instruction bytes, using the instruction prefetch pointer held in r10. This is part of the instruction buffer mechanism described later. After the three loads are issued, the current instruction is dispatched for interpretation via a branch and link instruction. Finally, r7 is copied into r6 to maintain the software pipeline as the "next" instruction becomes the "current" instruction.

The key point in organizing the dispatch loop in a software-pipelined fashion is to overlap the three loads related to future source instructions with the interpretation of the current instruction. If any of these three loads misses in the data cache, then the miss delay will be overlapped with the current instruction interpretation that follows. Perhaps this feature is most critical for prefetching instruction bytes (load of r11), because this particular load has a higher probability of missing in the cache. Note that IA-32 instructions are actually *data* as far as the interpreter code is concerned, and they contend for the data cache along with the actual IA-32 data.

In addition to the dispatch code, there must also be code (not shown) to manage the instruction prefetch buffer, held in registers r1 and r2. This buffer-maintenance code is executed as part of each interpreter routine. For instructions that cause a transfer of control (e.g., a branch or jump), the register buffer is filled with instructions from the target of the control transfer, and the various registers are updated appropriately. For all other instructions the buffer-maintenance code shifts the instruction buffer "up" by the length of the instruction just completed. For any given instruction, this is a

the instruction just completed. For any given instruction, this is a length that is known by the interpretation routine. The buffer shift involves shifting the upper buffer r1 and filling it in from below with instruction bytes from r2. The buffer length (held in r3) is decremented and checked to see if the number of buffered bytes has fallen below 8. If it has, the prefetched instruction word (in r11 from the dispatch loop) is copied into the lower buffer r2, and the buffers are adjusted. Register r10 is maintained as a pointer to the next block of instructions to be prefetched.

With respect to performance, the FX!32 developers report that their hand-optimized decode-and-dispatch interpreter takes an average of 45 Alpha instructions to emulate a single IA-32 instruction (**Chernoff et al. 1998**). Because IA-32 instructions are more complex than the Alpha RISC instructions, they further estimate that it takes about 30 Alpha instructions per IA-32 Pentium Pro micro-operation (the rough equivalent of a RISC instruction).

2.5 Binary Translation

With predecoding, all source instructions of the same type are executed with the same interpretation routine. For example, all *load word and zero* instructions, irrespective of the actual registers used, are executed with the code given in **Figure 2.9**. Performance can be significantly enhanced by mapping each individual source binary instruction to its own customized target code. This process of converting the source binary program into a target binary program is referred to as *binary translation* (**May 1987**; **Sites et al. 1993**). **Figure 2.16** shows an example of binary translation from a small IA-32 program sequence to the PowerPC ISA. Note that in this and following examples, the emulation of IA-32 condition codes is omitted for simplicity. Emulation of condition codes is discussed separately in **Section 2.8.2**.

The architected register values of the IA-32 source instructions are

The architected register values of the IA-32 source instructions are maintained in a register context block held in memory and are fetched into the registers of the target PowerPC ISA. Some of the target registers are permanently assigned to either contain or point to certain important and frequently used source resources. For example, r1 points to the register context block, r2 points to the memory image of the source machine, and the program counter is held in r3. The process of mapping source registers to target registers, as in the case of the program counter, is a very clear example of the guest-to-host *state mapping* that is a basic part of the virtualization process, as discussed at the beginning of **Chapter 1**.

```
addl   %edx,4(%eax)
movl   4(%eax),%edx
add    %eax,4
```

(a)

```
r1 points to IA-32 register context block
r2 points to IA-32 memory image
r3 contains IA-32 ISA PC value

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwzx    r5,r2,r5      ;load operand from memory
lwz     r4,12(r1)     ;load %edx from register block
add     r5,r4,r5      ;perform add
stw     r5,12(r1)     ;put result into %edx
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r5,r4,4       ;add 4 to %eax
lwz     r4,12(r1)     ;load %edx from register block
stwx    r4,r2,r5      ;store %edx value into memory
addi    r3,r3,3       ;update PC (3 bytes)

lwz     r4,0(r1)      ;load %eax from register block
addi    r4,r4,4       ;add immediate
stw     r4,0(r1)      ;place result back into %eax
addi    r3,r3,3       ;update PC (3 bytes)
```

(b)

Figure 2.16 Binary Translation from an IA-32 Binary to a PowerPC Binary. *(a) Intel IA-32 source binary sequence. (b) PowerPC target binary.*

Figure 2.17 compares predecoding (with threaded interpretation)

Figure 2.17 compares predecoding (with threaded interpretation) and binary translation. In both cases, the original source code is converted to another form. But in the case of predecoding, interpreter routines are still needed, while in binary translation the converted code is directly executed.

As shown in **Figure 2.16**, binary translated code may keep the target registers in a register context block held in memory, just as with interpretation. However, because each instruction translation is customized, state mapping can be used to map the registers in the source ISA directly to target ISA registers. By enabling the direct access of registers in the target code, memory accesses to the context block is eliminated. Such state mapping, especially of general-purpose registers, is generally not employed in interpreters, except perhaps for special registers that are implied by the opcode, such as the program counter or a condition code register.

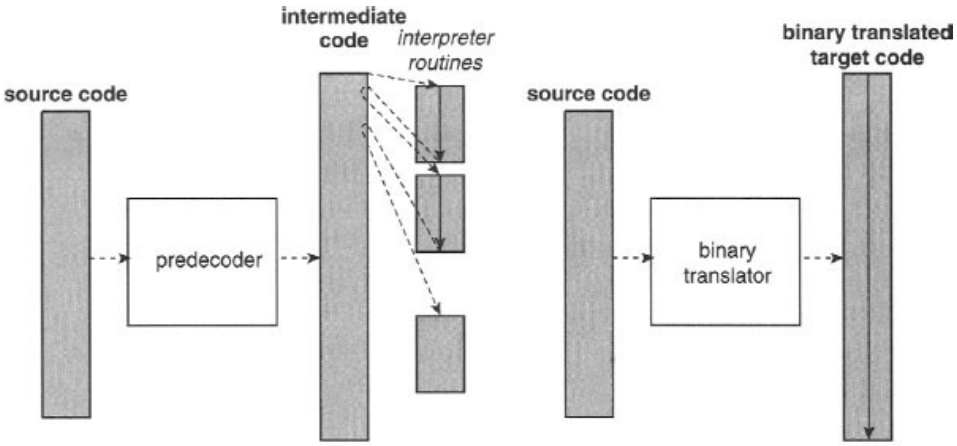
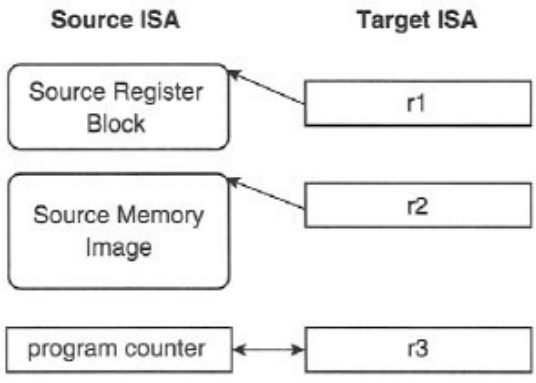


Figure 2.17 Threaded Interpretation, Using Intermediate Code, and Binary Translation.



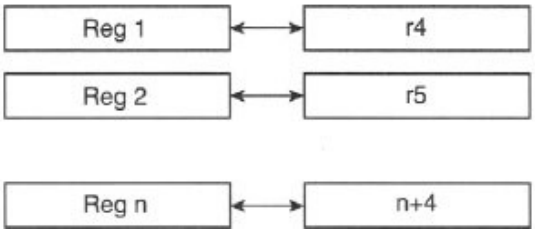


Figure 2.18 State Mapping from Target to Source Registers.

Figure 2.18 illustrates state mapping. Here, some of the target ISA registers point to the memory image and register block of the source ISA (as before). In addition, some of the target registers are mapped directly to some of the source registers; that is, source values are maintained in target registers. Other portions of the source state, such as the program counter and the stack pointer, may also be held in target registers. After mapping is performed, some target registers should be left over for use by the emulator code. Further discussion of register state mapping is in **Section 2.8.1**.

```
r1 points to IA-32 register context block
r2 points to IA-32 memory image
r3 contains IA-32 ISA PC value
r4 holds IA-32 register %eax
r7 holds IA-32 register %edx


addi    r16,r4,4        ;add 4 to %eax
lwzx    r17,r2,r16      ;load operand from memory
add     r7,r17,r7       ;perform add of %edx
addi    r16,r4,4        ;add 4 to %eax
stwx    r7,r2,r16       ;store %edx value into memory
addi    r4,r4,4         ;increment %eax
addi    r3,r3,9         ;update PC (9 bytes)
```

Figure 2.19 Binary Translation with State Mapping Maintained over Translation Block Boundaries.

Binary translation with register state mapping is illustrated in **Figure 2.19**, where the three IA-32 instructions in **Figure 2.16** are translated to seven PowerPC instructions, only one of which is involved in updating the source program counter. The speed of execution of the translated code now starts becoming comparable to the original source code. Furthermore, this sequence can be reduced

even more through optimization (as will be described in **Chapter 4**); for example, the second instance of the common subexpression "addi r16, r4, 4" can be eliminated.

2.6 Code Discovery and Dynamic Translation

2.6.1 The Code-Discovery Problem

From the preceding discussion one might infer that it is possible to predecode or binary translate a program in its entirety before beginning emulation. This is referred to as *static predecoding* or *static translation* because only static (pre-execution) program information is used. There are many situations, however, where such a static approach is difficult, if not impossible, to implement (**Horspool and Marovac 1980**).

Consider the following scenario. A sequence of instructions is being translated and an indirect jump instruction is encountered. The target of such a jump is held in a register. With static predecoding or static translation, it is difficult (or practically impossible) to determine the register contents (and the target of the jump instruction) because the register isn't assigned a value until run time. Furthermore, if an attempt is made to proceed with predecoding or translating the instructions immediately following the jump, there is no guarantee that the locations immediately following the jump contain valid instructions. In some cases, further analysis of other code may allow one to conclude that the location following the jump is a valid instruction, but the results of such an analysis are not always available.

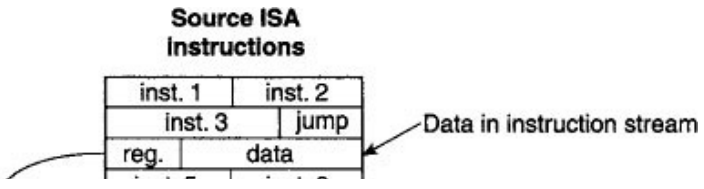
It would appear reasonable to assume that the compiler and linker keep instructions contiguous to one another and separate from all

data items. However, this is sometimes not the case. Some compilers (and assembly language programmers) intersperse data with code. For example, some compilers may provide a (read-only) mask that specifies which registers have been saved by a procedure caller at the time of the call. This mask may be placed immediately after the jump-and-link instruction that implements the call. Also, a compiler may "pad" the instruction stream with unused bytes in order to align branch or jump targets on word or cache line boundaries for performance reasons. Whatever the reason for interspersing data in code sections, it poses difficulties in statically identifying the starting points of all instruction sequences in a given region of memory.

Despite the foregoing observations, static predecoding and binary translation may still seem fairly straightforward from the simple examples given in the previous subsections because, in those examples, we were considering RISC instructions with fixed-length encodings. Instructions are of variable-length in CISC ISAs such as the IA-32. Hence, sequences of IA-32 instructions can begin at any byte boundary rather than on a word boundary as in common RISC ISAs. Even if it is known that a certain location in memory is part of a section of code, it is usually difficult in a CISC ISA to determine whether or not an arbitrary byte is at the start of a new instruction sequence. In **Figure 2.20**, for example, it is not obvious whether the byte beginning with 8b marks the start of a sequence beginning with a movl instruction, or ends some previous instruction, with the next byte, b5, starting a mov instruction.

```
                mov  %ch,0  ??
31 c0 | 8b | b5 00 00 03 08 8b bd 00 00 03 00
          movl  %esi, 0x08030000(%ebp)??
```

Figure 2.20 Finding IA-32 Instruction Boundaries in an Instruction Stream.



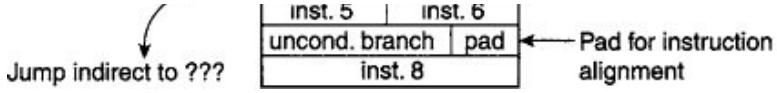


Figure 2.21 Causes of the Instruction Discovery Problem.

To summarize, in real code using conventional ISAs, especially CISC ISAs, code discovery can be a problem because of variable-length instructions, register indirect jumps, data interspersed with the instructions, and pads to align instructions; see **Figure 2.21**.

2.6.2 The Code-Location Problem

As discussed earlier, translated code is accessed with a target program counter (TPC), which is different from the architected source program counter (SPC). This creates a problem when there is an indirect control transfer (branch or jump) in the source code. The destination address of the control transfer is held in a register and is a source code address, even though it occurs in the translated code. During emulation, then, there must be some way to map an SPC address to a TPC address. The code shown in **Figure 2.22** will not work properly because the target code cannot jump to a source code location. This problem is referred to as the *code-location problem*.

```
mov1   %eax, 4(%esp)      ;load jump address from memory
jmp    %eax               ;jump indirect through %eax
```

(a)

```
addi   r16,r11,4          ;compute IA-32 address
lwzx   r4,r2,r16          ;get IA-32 jump address from IA-32 memory image
mtctr  r4                 ;move to count register (ctr)
bctr                      ;jump indirect through ctr
```

(b)

Figure 2.22 Translation of an Indirect Jump Code Sequence. *The value in the jump register is a source code address in both cases; consequently, this code translation will not work correctly. (a) IA-32 source code; (b) PowerPC target code.*

In general, the code-discovery and code-location problems require sophisticated solutions that will be discussed in the next section. However, there are special situations where the solutions are sim-

pler. One of these we have already seen: instruction sets with fixed-length (typically 32-bit) instructions that are always aligned on fixed boundaries, as is typical in RISC ISAs. Another special situation occurs with source instruction sets that are explicitly designed to be emulated, such as Java bytecodes. These virtual instruction sets do not allow data to be interspersed with code (other than immediate values associated with instructions), and they have restricted control flow instructions (branches and jumps) that enable easy code discovery.

2.6.3 Incremental Predecoding and Translation

For an arbitrary ISA, code discovery is a problem both for predecoding and for binary translation. In both cases, a general solution is to translate the binary while the program is operating on actual input data, i.e., *dynamically,* and to predecode or translate new sections of code *incrementally* as the program reaches them. Because the overall process is more or less the same for both predecoding and binary translation, we will simply refer to both as *translation.*

The overall process is illustrated in **Figure 2.23**. High-level control is provided by an *emulation manager* (EM), which is part of the runtime support. Other major components include an interpreter and the binary translator. The interpreter may be either a decode-and-dispatch interpreter or a simple threaded interpreter as described in **Section 2.2**. The important point is that the interpreter operates on the original source binary code.

As blocks of code are translated (or predecoded), they are placed into a region of memory reserved for them. As more and more code is translated, this memory region could become quite large, however, which is potentially wasteful for code that is rarely used. Therefore, to reduce the memory space that holds translated code, it is typically organized as a *code cache* (**Deutsch and Schiffman 1984**). The objective of the code cache is to hold the more recently

used blocks of translated code. We defer details of code caching and code cache management to **Chapter 3**. For the remainder of this chapter, the reader can simply assume that the code cache is always large enough to hold the translated code.

Finally, a *map table* associates the SPC for a block of source code with the TPC for the corresponding block of translated code. The map table essentially provides a way of indexing into the code cache, and it is typically implemented as a hash table. The SPC comes from the interpreted or translated program, and the TPC points to the beginning of the translated block in the code cache. If the EM wants to find a block of translated code (or determine if it has yet been translated), the SPC is applied to the map table. The corresponding TPC value (pointing into the code cache) is produced if the code block has been translated (i.e., if there is a *hit* in the code cache). Otherwise the map table indicates that there is a code cache *miss,* and additional translation is required.

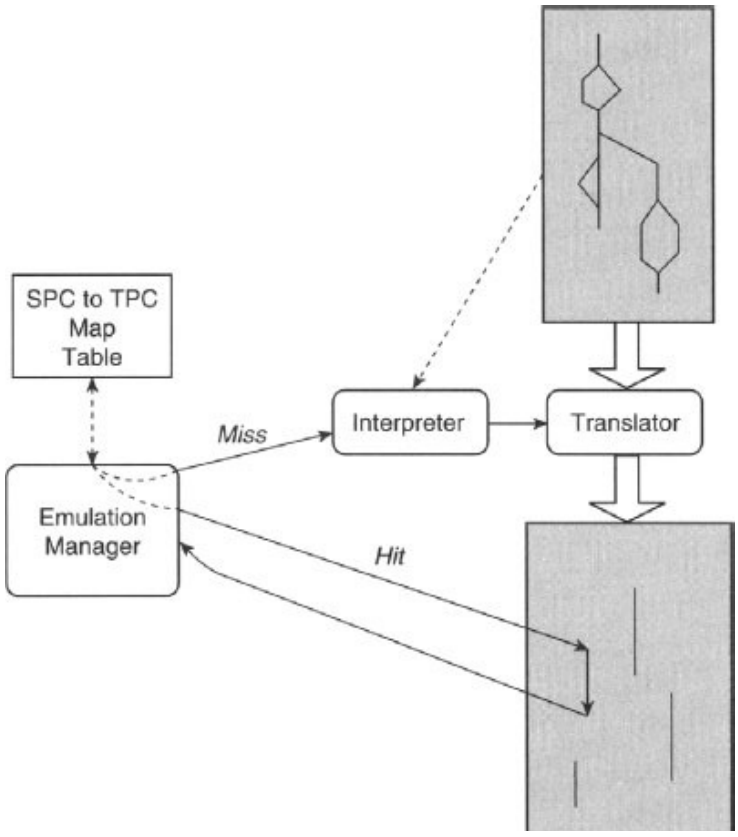


Figure 2.23 Overview of Dynamic Translation System. *Dotted lines indicate data accesses; solid lines indicate flow of control.*

The system translates one block of source code at a time. In a simple translation scheme, the natural unit for translation is the *dynamic basic block.* A dynamic basic block is slightly different from a conventional basic block, determined by the static structure of a program (see **Figure 2.24**a). A *static basic block* of instructions contains a sequence with a single entry point and a single exit point. In essence, static basic blocks begin and end at all branch/jump instructions and branch/jump targets.

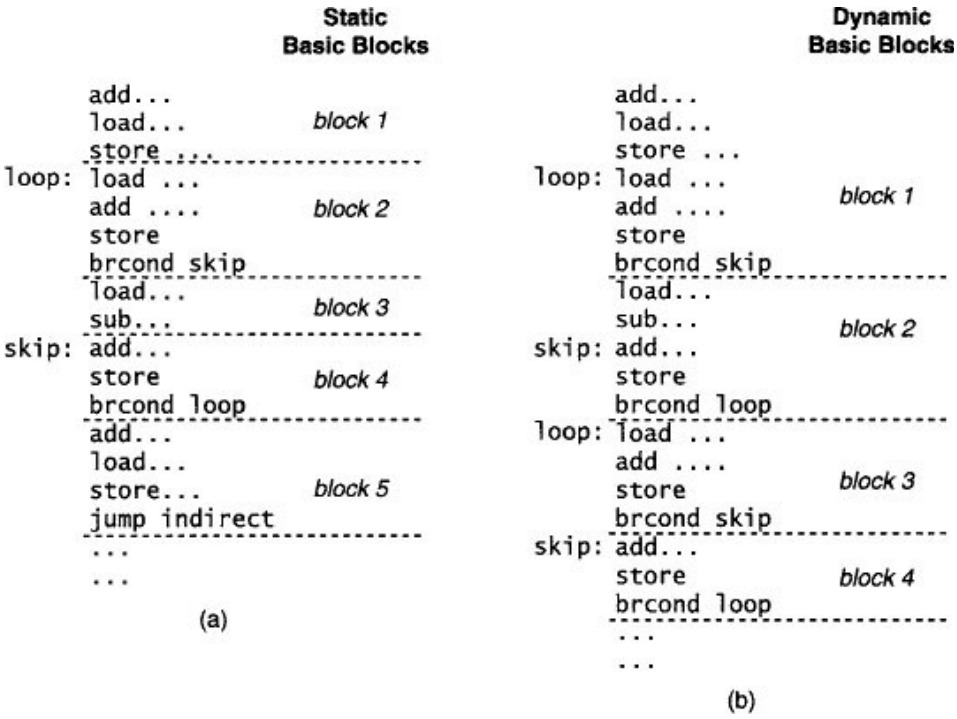


Figure 2.24 Static Versus Dynamic Basic Blocks. (*a) Static basic blocks are code sequences with one entry point and one exit point. They begin and end with control transfer instructions or targets of control transfer instructions. (b) Dynamic basic blocks are often larger than static basic blocks and are determined by the actual flow of control at run time.*

A dynamic basic block is determined by the actual flow of a program as it is executed. A dynamic basic block always begins at the instruction executed immediately after a branch or jump, follows the sequential instruction stream, and ends with the next branch or jump. In **Figure 2.24**b, when the static loop shown is first entered

from the top, the instruction at loop happens to be the target of a branch, yet it does not terminate the dynamic basic block. The dynamic basic block continues until the first conditional branch is encountered. Dynamic basic blocks tend to be larger than static basic blocks. Note that the same static instruction may belong to more than one dynamic basic block. For example, the add instruction at label skip belongs to dynamic basic block 2 as well as the shorter dynamic basic block 4. In the remainder of this book the term *basic block* will mean "dynamic basic block" unless stated otherwise. Translation methods in this chapter will operate on one dynamic basic block at a time. A translation unit larger than a single dynamic basic block can often be beneficial, however. Such larger translation units are discussed in **Chapter 4**.

A simple, incremental translation process works as follows. After the source binary is loaded into memory, the EM begins interpreting the binary using a simple decode-and-dispatch or indirect threaded method. As it proceeds, the interpreter dynamically generates the intermediate code or translated target binary. This translated code is placed in the code cache, and the corresponding SPC-to-TPC mapping is placed into the map table. When a branch or jump is encountered, the interpreter has completed translation of a dynamic basic block.

The EM then follows the path of the source program's control flow (using the map table) and either directly executes the next block if it is already translated (there is a hit in the table) or begins translating the next dynamic basic block if it hasn't been previously translated (a miss in the table). Incrementally, more of the program is discovered and translated, until eventually only translated code is being executed (except for the emulation manager code that connects the control flow from one translated block to the next—**Figure 2.25**). The entire process is summarized in the flowchart given in **Figure 2.26**.

An apparent complication to the translation process occurs when a branch goes into the middle of a block that is already translated. The destination translation block could be divided into two pieces at that point. However, in order to discover such a situation, it would be necessary to maintain an additional data structure to keep track of address ranges of translated code blocks and then to search the structure whenever a miss occurs in the map table. This apparent complication does not occur when dynamic basic blocks are used (a major reason for using dynamic basic blocks). A new translation is always started when there is a miss in the map table, even if it leads to replicated sections of translated code.

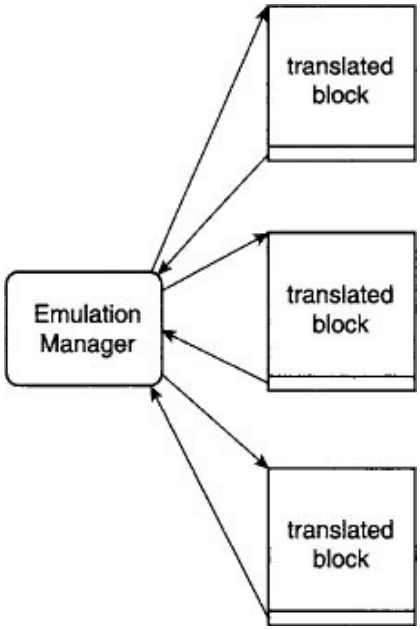
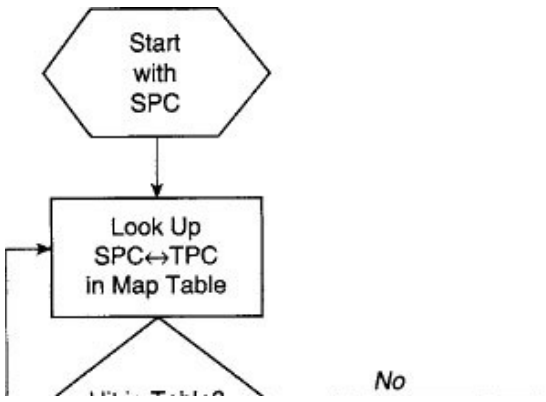


Figure 2.25 Flow of Control Involving Translated Blocks. *The emulation manager handles control transfers from one translated block to the next.*



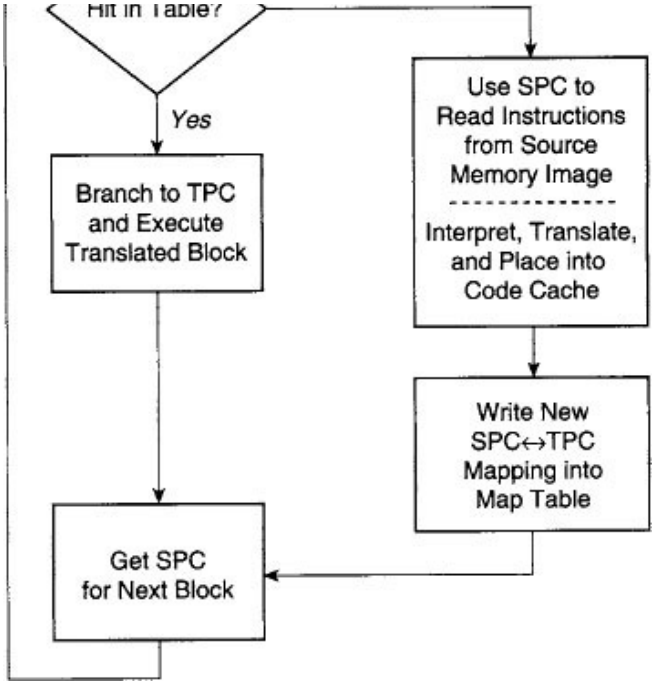


Figure 2.26 Dynamic Translation Flowchart.

Tracking the Source Program Code

It is important that the translation system keep track of the SPC value at all times while emulation is taking place. In the translation system, control is shifted as needed between the interpreter, the EM, and translated blocks in the code cache, and each of these components must have a way of tracking the SPC. First, the interpreter uses the SPC directly as it fetches source instructions. When the interpreter transfers control to the EM at the end of a basic block, it passes the next SPC to the EM. Similarly, when a block of translated code finishes executing, the value of the next SPC must be made available to the EM. One way of doing this is to map the SPC to a register on the host platform, with the register being updated either at each translated instruction or at the end of each translated block (as in **Figure 2.19**). Another possibility is shown in **Figure 2.27**. Here the value of the next SPC is placed in a "stub" at the end of the translated block. When the translated block finishes, control is transferred back to the EM using a jump-and-link (JAL) instruction. The link register can then be used by the EM to access the SPC from the

end of the translated code block (**Cmelik and Keppel 1994**).

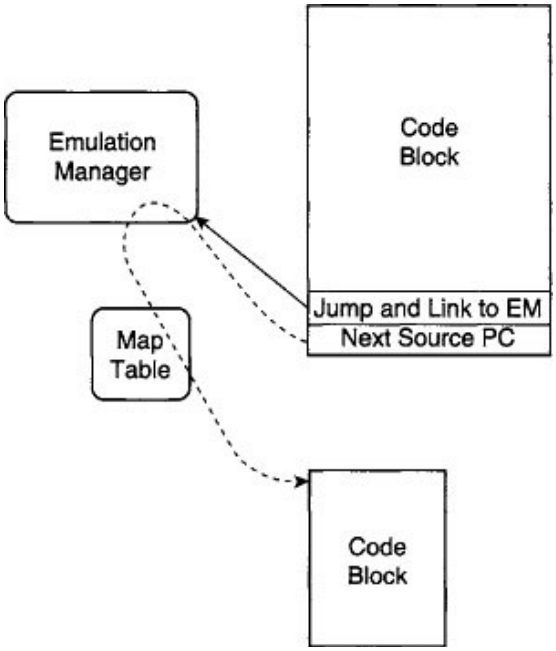
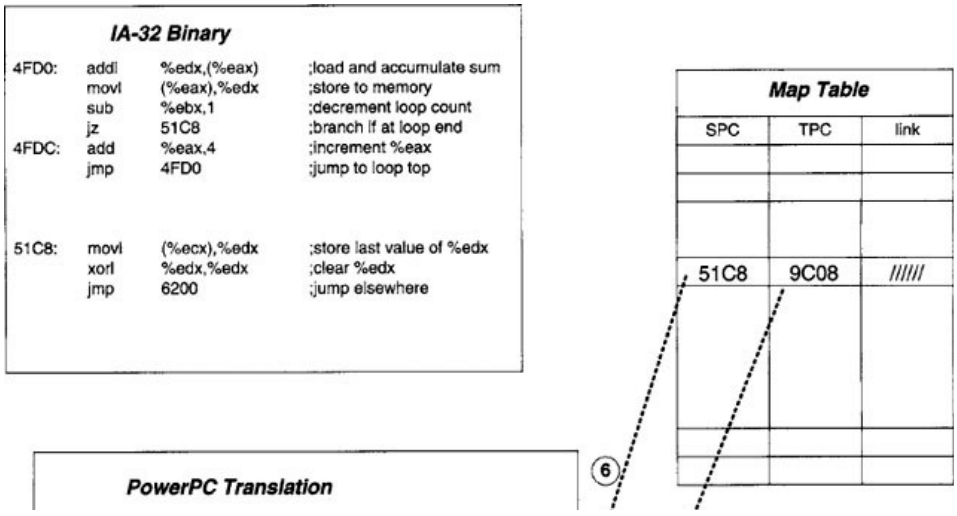


Figure 2.27 Linking Translated Blocks via a Stub.

Example

Figure 2.28 is an extended example that illustrates all the parts of **Figure 2.27**. In this example two basic blocks of IA-32 code have been binary translated to blocks of PowerPC code. The following sequence takes place as one translated block finishes and control is transferred to the next translated block, via the emulation manager.



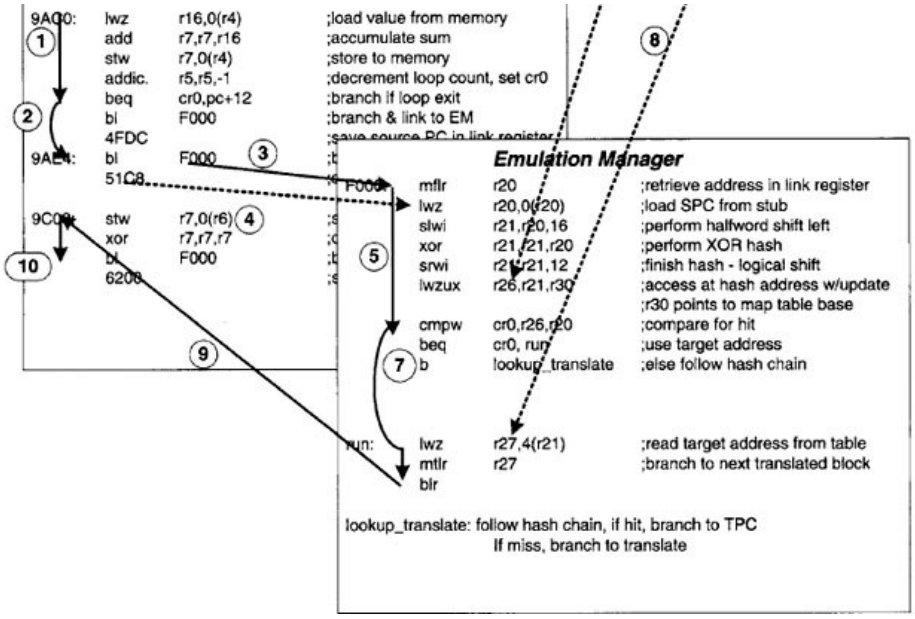


Figure 2.28 Example of Binary Translation.

1. Translated basic block is executed.
2. Branch is taken to stub code.
3. Stub does branch and link to emulation manager entry point.
4. EM loads SPC from stub code, using link register.
5. EM hashes SPC to 16 bits and does lookup in map table.
6. EM loads SPC value from map table; comparison with stub SPC succeeds.
7. Branch to code that will transfer code back to translation.
8. Load TPC from map table.
9. Jump indirect to next translated basic block.
10. Continue execution.

Other issues

There are other issues related to instruction set emulation and dynamic translation that will be covered in **Chapters 3** and **4**. Here are the more important of these.

Self-modifying code—Although in many applications it is uncommon, programs occasionally perform stores into the code area; i.e., the code is self-modifying. When this happens, translated code held in the code cache may no longer correspond to the modified source code. Consequently, mechanisms must be in place to invoke retranslation. The handling of self-modifying code is covered in **Section**

3.4.2.

Self-referencing code—Here, the program performs loads from the code area. When this happens, the data that is read must correspond to the original source code, not the translated version. Handling of self-referencing code is also covered in **Section 3.4.2**.

Precise traps—If the translated code should experience an exception condition (an interrupt or trap), the correct state corresponding to the original source code, including the SPC of the trapping instruction, must be produced. Providing precise state at the time of a trap or interrupt is an important recurring topic throughout both **Chapters 3** and **4**.

2.6.4 Same-ISA Emulation

Although it may seem like an odd idea at first, there are important VM applications where the source and target ISAs are the same and where the emulation techniques described in this chapter are used. Of course, there is no logical reason why an instruction set can't be used for emulating itself. For example, using interpretation techniques, instructions from a source binary can be discovered, parsed, and emulated regardless of whether the target and source ISAs are the same or different. And, naturally, "binary translation" is greatly simplified when the source and target ISAs are identical.

The important, useful aspect of same-ISA emulation is that the emulation manager is always in control of the software being emulated. Emulation software discovers and inspects each source instruction and, in the process, can identify details concerning operations to be performed by that specific instruction. Furthermore, it can monitor the execution of the source program at any desired level of detail. This monitoring, or *code management,* capability is the key to several applications of same-ISA emulation. One application is simulation, where dynamic program characteristics are collected during

the simulation process. Simulation is the primary application of the Shade System, discussed as a case study in **Section 2.9**. A second application is operating system call emulation in virtual machines where the ISAs are the same but the host operating system differs from the guest operating system. By using emulation techniques, all operating system calls in a guest's source binary can be detected and can be translated to host operating system calls. A third application is the discovery and management of certain privileged operations that require special handling in some system VMs; these will be discussed in **Chapter 8**. A fourth application, discussed in **Chapter 10**, is "program shepherding" where the targets of control transfer and other instructions are monitored to ensure that they do not exploit security holes. Finally, there is the application to same-ISA dynamic binary optimization where runtime information is used to help optimize a program binary as it executes, even though no ISA translation is performed. This final application can be used as an end in itself, but, perhaps more importantly, it allows the other applications to be implemented while mitigating any resulting performance losses.

Same-ISA interpretation requires no special discussion; the techniques are exactly as described earlier. For binary "translation"[2] (without optimization), the simplest technique is just to copy the code, using exactly the same code in the target as in the source (except where there are operating system calls or instructions requiring special handling). Same-ISA dynamic binary optimization is discussed in **Section 4.7**.

2.7 Control Transfer Optimizations

Using the simple translation method described thus far, every time a translation block finishes execution, the EM must be reentered and an SPC-to-TPC lookup must occur. There are a number of optimizations that reduce this overhead by eliminating the need to go

through the EM between every pair of translation blocks. These optimizations are described in the following subsections.

2.7.1 Translation Chaining

Chaining in binary translators is the counterpart of threading in interpreters. Instead of branching to the emulation manager at the end of every translated block, the blocks can be linked directly to each other. This is illustrated in **Figure 2.29**.

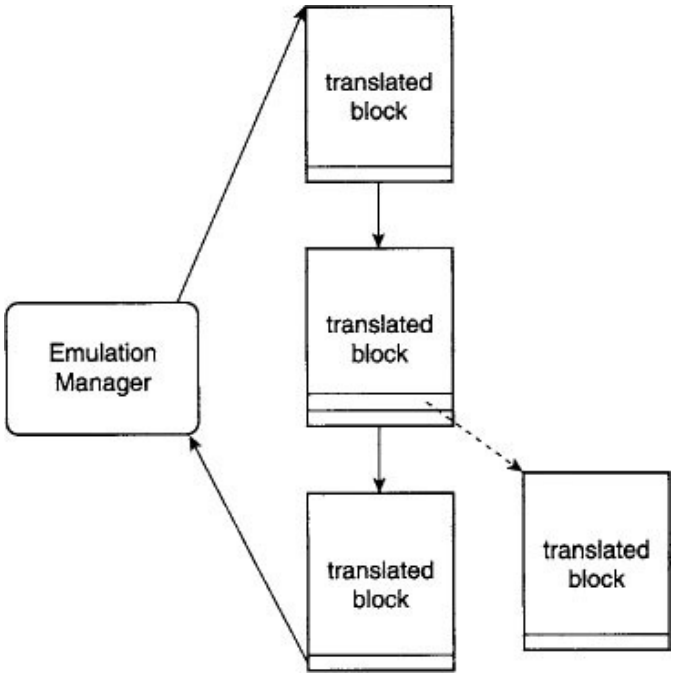


Figure 2.29 Chaining of Translation Blocks Avoids Indirect Transfers Through the Emulation Manager.

In this scheme, blocks are translated one at a time, but they are linked together into chains as they are constructed. Linking is accomplished by replacing what was initially a jump and link (or branch and link, *bl*, in PowerPC) back to the EM, as shown in **Figure 2.27**, with a direct branch to the successor translation block. The address of the successor block is determined by using the SPC value to access the map table and find the corresponding TPC (if the corresponding basic block has been translated).

If the successor block has not yet been translated, then the normal stub code is inserted. At some time later, after the successor has been translated and the predecessor block exits to the EM, the EM will access the map table, find the entry in the table, and retrieve the TPC for the successor block. At that point, the EM can set up a link to the successor block by overwriting the jump-and-link (branch-and-link in PowerPC parlance) in the predecessor with a direct jump to the successor. The steps in this process are illustrated in **Figure 2.30**. The code example from **Figure 2.28**, with chaining added, is given in **Figure 2.31**.

Chaining works in those situations where the SPC can be placed in a stub at the end of a translation block, as in **Figure 2.27**. These are situations where the destination of a branch or jump never changes. However, for register indirect jumps, especially procedure returns, the target may change from one execution of the jump to the next. Because a single SPC cannot be associated with the translation block terminated by the register indirect jump, chaining is not convenient in these cases. The easiest way to handle indirect jumps, therefore, is always to go through the EM and to let the EM look up the correct TPC via the map table. Faster methods for handling indirect jumps are described in the next two subsections.

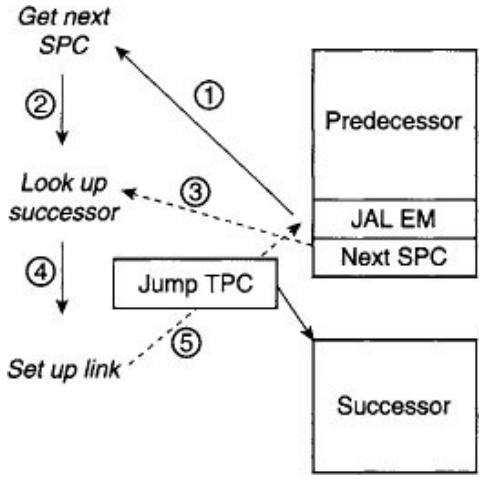


Figure 2.30 Creating a Link from a Translated Predecessor Block to a Successor Block.

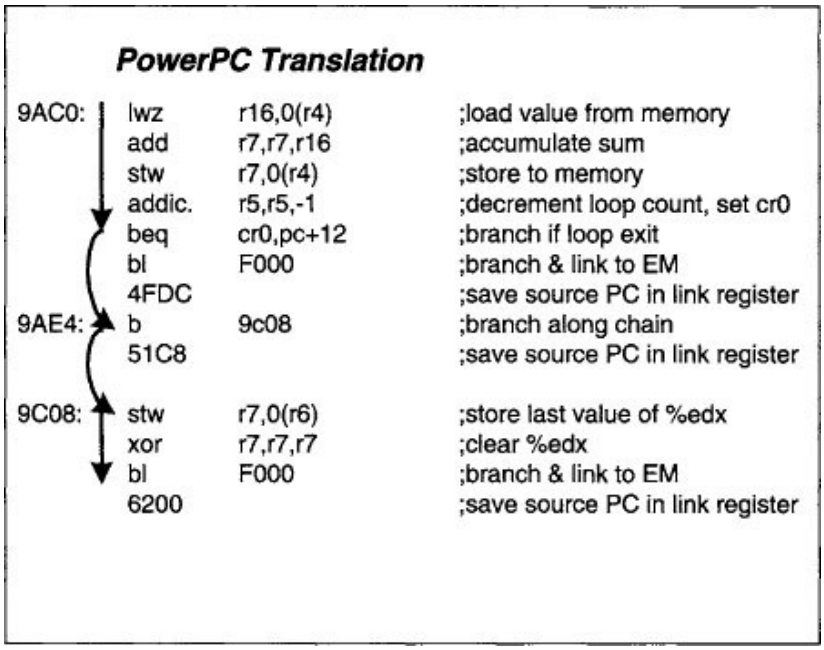


Figure 2.31 Translated PowerPC Code from an Earlier Example, with Chaining Implemented.

2.7.2 Software Indirect Jump Prediction

As pointed out earlier, implementing indirect jumps by map table lookup is expensive in execution time. It requires several instructions to hash the SPC to form an index into the map table, then at least one load and a compare to find a matching table entry, and finally a load and an indirect jump to the TPC address. In many cases, however, the jump target never or very seldom changes. In these cases, a form of software jump prediction can be employed to reduce the overhead of indirect jumps. This is essentially an implementation of *inline caching,* developed for fast Smalltalk emulation (**Deutsch and Schiffman 1984**). **Figure 2.32** illustrates the technique. In the example, Rx symbolizes a register holding the indirect jump target PC value. In a series of *if* statements, the most frequent SPC addresses, with their matching TPC addresses, are encoded into the translated binary code (given here in high-level form for clarity). Then the register value for an indirect jump can be compared with the SPC value; if there is a match, then a branch to the destination (possibly translated as a PC relative branch) takes place. Typically,

the comparisons are ordered, with the most frequent SPC destination address being given first. Of course, in the worst case, if all the prediction(s) are wrong, then the map table lookup must be performed anyway, so that additional performance is lost. Hence, this technique should be coupled with *profiling* that provides accurate information regarding indirect jump targets (**Section 4.2.4**).

```
if(Rx == addr_1) goto target_1;
else if (Rx == addr_2) goto target_2;
else if (Rx == addr_3) goto target_3;
else table_lookup(Rx);          do it the slow way
```

Figure 2.32 Software Indirect Jump Prediction via Inline Caching. *Source PC values are given as immediate values addr_i; corresponding target PC values are given as target_i.*

When using this method, the EM maintains a side table that keeps track of the successor translated blocks that can be reached via such software predictions. In the event that the successor translated block is removed from the code cache, the software prediction code in the predecessor block must also be modified or removed (and is replaced with a map table lookup).

2.7.3 Shadow Stack

When a translated code block contains a procedure call via an indirect jump to a target binary routine, the SPC value must be saved by emulation code as part of the source architected state, either in a register or in a memory stack (depending on the source ISA). Then when the called procedure is finished, it can restore this SPC value, access the map table, and jump to the translated block of code at the return address. As pointed out earlier, this map table lookup adds overhead to the emulation process. The overhead can be avoided if the target return PC value can be made available directly, i.e., as a link address.

To perform this optimization, the return (link) value of the target code is pushed onto a *shadow stack* maintained by the emulation manager (**Chernoff et al. 1998**). Note that this return value is not

necessarily the memory address immediately following the procedure jump; if the jump is at a translation block boundary, for example, then the target return value may be linked to the beginning of the next translated block. In any case, when it is time to return, this translation code address is popped from the shadow stack and a map table lookup can be avoided. There is one problem, however. The source code could have changed the contents of its stack between the call and the eventual return. Consequently, before the shadow stack can be used to provide a target return address, the value on the top of the stack must be checked against the corresponding source return address. Hence, the shadow stack frame is expanded to include not only the return address of the target translated code but the return address of the source binary. The return address of the source is compared with the source field in the shadow stack return address. If there is a match, then the target return address from the shadow stack can be used. If there is a mismatch, then the source return address is used to access the map table in the conventional manner.

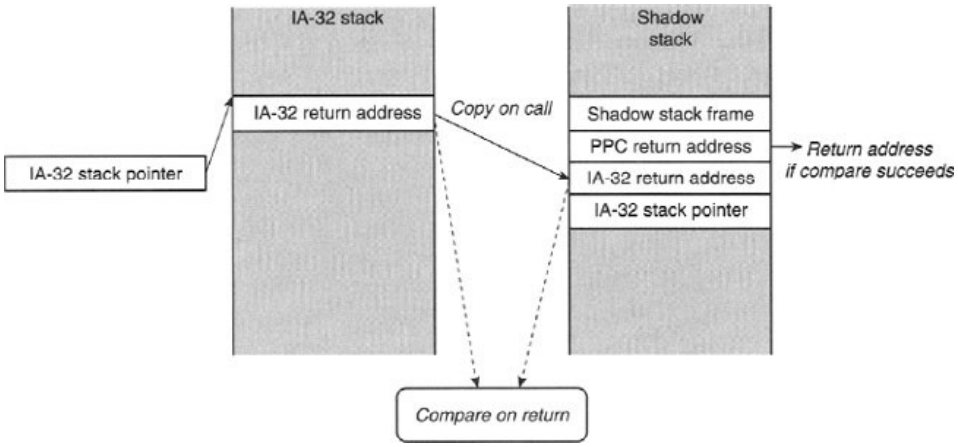


Figure 2.33 Shadow Stack Implementation to Expedite Returns to Translated Code.

The shadow stack mechanism where the source ISA is IA-32 and the target ISA is PowerPC is illustrated in **Figure 2.33**. The IA-32 uses an architected stack. When the code performs a procedure call, the PowerPC return address is pushed on the shadow stack; along with the IA-32 return address. Then when it is time to return, the IA-32

the IA-32 return address. Then when it is time to return, the IA-32 return address is loaded from the emulated IA-32 stack. This address is compared with the IA-32 return address saved on the shadow stack. If these match, then the shadow stack PowerPC return value is consistent with the source code return value, and the PowerPC return value can be used to jump to the return translation block. Otherwise, the IA-32 return value retrieved from the IA-32 stack is used to hash into the PC map table for the return address.

Note that the IA-32 stack pointer is also saved on the shadow stack. If the guest program should happen to cut back the IA-32 stack by discarding a number of stack frames, this can be detected by comparing the pointer on the shadow stack with the emulated IA-32 stack pointer. If there is a mismatch, the emulated IA-32 stack pointer can then be used for cutting back the shadow stack so that it is consistent with the emulated source stack.

2.8 Instruction Set Issues

Thus far, we have provided several examples of instruction set emulation. There are many more details to be taken care of in translating and interpreting a complete instruction set. Some of these details will be discussed in this section; some are specific to certain instructions sets, and others apply more generally.

2.8.1 Register Architectures

Virtually every ISA uses registers of some kind. Registers are at the very top of the storage hierarchy and are critical to performance. Hence, the way registers are handled is a key issue during emulation. The general-purpose registers of the target ISA are used for a number of functions, including (1) holding general-purpose registers of the source ISA, (2) holding special-purpose registers of the source ISA, (3) pointing to the source register context block and to the memory image, and (4) holding intermediate values used by the

the memory image, and (4) holding intermediate values used by the emulator.

If the number of target registers is significantly larger than the number of source registers, then all the foregoing uses for target registers may be satisfied simultaneously. For example, in emulating an IA-32 on a 32-register RISC such as the PowerPC, all the IA-32 general-purpose registers and some of the special registers, such as the program counter, can be mapped to target registers, and there are registers left over to point to the register context block, to the source memory image, and scratch registers for use by emulation code.

In other cases, there may not be enough target registers to perform all the foregoing functions, especially when the source and target ISAs have approximately the same number of general-purpose registers. In these cases, the usage of the target general-purpose registers must be carefully managed. Two registers are needed to point to the register context block and to the source memory image; these pointers are used very frequently during the emulation process. Similarly, a target register should be assigned to the program counter. Such an assignment of registers is used in **Figure 2.18**, for example. If interpretation is being used, a target register should also be designated to hold the SPC. If binary translation or predecoding is used, an additional register may hold the TPC. If the source ISA has a stack pointer, condition codes, or other frequently used special registers, target registers should be assigned to these as well. In total, this could consume from three to ten target registers. Thus, between three and ten general-purpose target registers may be needed for holding the state of source ISA resources, beyond those needed for mapping source general-purpose registers.

After assigning target registers to the most common or performance-critical source resources, the emulation manager must allocate the target's remaining general-purpose registers. If inter-

pretation is used, the remaining registers may be used as scratch registers by the interpreter. If translation is used, then source registers can be mapped to target registers on a translation block basis. All registers read in the translated code are copied from the context block into target registers when the translation block is entered. Any modified registers are copied back to the context block when the translation block is exited. If the translation block is especially large, then source registers may be spilled to the context block. In some of the binary translation examples given earlier, e.g., **Figure 2.19**, these intermediate spills to the context block are not shown, though they would be used in practice.

2.8.2 Condition Codes

Condition codes are special architected bits that characterize instruction results (zero, negative, etc.) and are tested by conditional branch instructions. However, across ISAs there is little uniformity in the way condition codes are used or whether they are used at all. The Intel IA-32 ISA sets condition codes implicitly as a side effect of executing most instructions; consequently, the condition codes are updated nearly as often as general-purpose registers. The SPARC ISA contains explicitly set codes, which update condition codes only when it is anticipated that they will actually be used. The PowerPC ISA has a number of condition code registers that are also explicitly set. Adding to the complication, some ISAs do not use any condition codes; the MIPS ISA is an example.

Emulation complexity varies significantly, depending on whether the source ISA, the target ISA, both, or neither uses condition codes. The easiest situation is if neither ISA uses condition codes. Almost as easy is the case where the source ISA does not use condition codes but the target ISA does. Here, some additional target instructions may be required to generate condition code values, but there is no need to maintain any source condition code state. The emulation of MIPS on a PowerPC is illustrated in **Figure 2.34**. Here, a conditional

branch in the MIPS ISA is translated to two instructions in the PowerPC ISA—one that sets a condition register (crl in the figure) and a second that performs the conditional branch by testing crl.

```
beq    r1,r2,offset
```

(a)

```
cmpw   cr1,r1,r2
beq    cr1,offset
```

(b)

Figure 2.34 Binary Translation of a Conditional Branch from the MIPS ISA to the PowerPC ISA. (*a) MIPS branch equal; (b) PowerPC translation to a compare instruction followed by a branch.*

If a target ISA has no condition codes, then the source condition codes must be emulated, and this emulation can be time consuming. The most difficult problems usually occur when the source ISA has implicitly set condition codes and the target ISA has no condition codes. It is because of these difficulties that we have been ignoring the emulation of IA-32 condition codes in examples up to now.

Recall that implicitly set condition codes are modified as a side effect of many instructions. In the Intel IA-32, the condition codes are a set of “flags” held in the EFLAGS register. The IA-32 integer add instruction always sets six of these condition codes (flags) whenever it is executed. The condition codes set by the add are:

OF: indicates whether integer overflow occurred

SF: indicates the sign of the result

ZF: indicates a zero result

AF: indicates a carry or borrow out of bit 3 of the result (used for BCD arithmetic)

CF: indicates a carry or borrow out of the most significant bit of the result