Reverse-Engineering Instruction Encodings

Wilson C. Hsieh  Dawson R. Engler  Godmar Back
University of Utah  Stanford University  University of Utah
wilson@cs.utah.edu  engler@csl.stanford.edu  gback@cs.utah.edu

Abstract

Binary tools such as disassemblers, just-in-time compilers, and executable code rewriters need to have an explicit representation of how machine instructions are encoded. Unfortunately, writing encodings for an entire instruction set by hand is both tedious and error-prone. We describe DERIVE, a tool that extracts bit-level instruction encoding information from assemblers. The user provides DERIVE with assembly-level information about various instructions. DERIVE automatically reverse-engineers the encodings for those instructions from an assembler by feeding it permutations of instructions and analyzing the resulting machine code. DERIVE solves the entire MIPS, SPARC, Alpha, and PowerPC instruction sets, and almost all of the ARM and x86 instruction sets. Its output consists of C declarations that can be used by binary tools. To demonstrate the utility of DERIVE, we have built a code emitter generator that takes DERIVE’s output and produces C macros for code emission, which we have then used to rewrite a Java JIT backend.

1 Introduction

Binary tools such as assemblers, debuggers, disassemblers, dynamic code generation systems [3, 8, 10, 15, 18], just-in-time compilers [4, 7, 12], and executable code rewriters [14, 21, 24] need to contain a representation of how machine instructions are encoded. For example, a JIT needs to know that the x86 instruction addl %ecx, %ebx corresponds to the bits 0x011c (formed by a bitwise OR of the addl opcode 0x01c0 with 0x18 for the %ecx operand and 0x3 for the %ebx operand). Unfortunately, specifying instruction encodings with current tools requires looking up and detailing the offsets, sizes, and values of many instruction fields. Unsurprisingly, this process is both error-prone and tedious, especially for CISC machines such as the x86.

Currently, system builders must transcribe instruction encodings from an architecture reference manual. We have built a tool called DERIVE that eliminates the need to specify many of the bit-level details of instruction layout. The user supplies DERIVE with an assembly-level description of an instruction set: instruction names and operand types (registers, immediates, or labels). DERIVE outputs a description of how each instruction is encoded, which is given in the form of C structure declarations. The user does not specify binary-level details such as operand widths, operand offsets, opcode values, and register value encodings. As a result, the potential for misspecification by the user is less than that with other tools.

DERIVE is based on a simple observation: virtually all architectures for which a programmer needs binary encodings will already have programs (assemblers) that contain this information. Therefore, we should be able to extract the information from the assembler, which is what DERIVE does. At a high level, DERIVE does so by feeding permutations of each instruction to the system’s assembler, and doing equation solving on the assembler’s output to determine how the instruction is encoded (its opcode and its operand encodings). The DERIVE implementation solves the entire MIPS, SPARC, Alpha, and PowerPC instruction sets. It handles most of the ARM and x86 instruction sets: it does not yet handle some instructions, such as those with non-continuous fields.

DERIVE produces C data structure declarations that describe how an instruction set is encoded. It would not be difficult to produce declarations in other languages. These declarations can be used by tools that interpret or manipulate binaries. As an example, we have built a code emitter generator that takes the reverse-engineered declarations and generates C macros for fast code emission. To demonstrate the utility of these tools, we have rewritten Kaffe’s [25] JIT compiler to use these macros. Using our automatically generated code emitters reduced the size of the Kaffe backend description by 40%.

On a side note, DERIVE can be viewed as an assembler tester. Because of how it reverse-engineers instructions, it can quickly find differences between

---

A short version of this paper that only described the DERIVE solver appeared in the proceedings of the DYNAMO 2000 workshop [6].
what an architecture manual says, and what an assembler actually implements.

In Section 2 we discuss related work: Collberg’s compiler-reverse-engineering system and the New Jersey Machine-Code Toolkit. Section 3 describes our assumptions about instruction set encodings, and explains how DERIVE works. Section 4 discusses the code emitter generator that we have built on top of DERIVE. Section 5 summarizes our conclusions and describes our future work. Appendix A gives a complete input specification of the MIPS instruction set for DERIVE.

2 Related Work

The most important precedent to DERIVE is Collberg’s work on reverse-engineering compilers [1]. Collberg’s system does a “reverse interpretation” to infer what assembly instructions should be used to implement a high-level language. It runs pieces of mutated assembly code to see how the semantics of the instructions changes. His work is aimed at building automatically retargeting compilers, where the descriptions of the semantics of assembly instructions are used to automatically retarget the BEG back-end generator [5].

DERIVE is complementary to Collberg’s work. DERIVE computes information that can be used to bypass the assembler, which is important for building JITs and binary manipulation tools. DERIVE addresses a simpler problem domain than Collberg’s system, since it reverse-engineers a syntax transformation and not a semantic transformation. On the other hand, our problem domain is much more tractable: it takes on the order of minutes to hours for DERIVE to reverse-engineer an instruction set’s encoding, whereas Collberg’s system takes days to reverse-engineer an instruction set’s semantics.

The work whose goals most closely match DERIVE is the New Jersey Machine-Code Toolkit (NJT) [20]. The NJT automatically generates routines to manipulate machine-code from user specifications written in a language called SLED. SLED specifications are exact descriptions of instruction layout, written at several levels of abstraction. At the lowest level of description, SLED “fields” are used to describe instruction bitfields. For example, the description of the SPARC instruction fields in SLED reads as follows [19]:

```
inst 0:31 op 30:31 disp30 0:29 rd 25:29
op2 22:24 imm22 0:21 a 29:29 cond 25:28
disp22 0:21 op3 19:24 rsi 14:18 i 13:13
asi 5:12 rs2 0:4 simm13 0:12 opf 5:13
```

DERIVE could be used to eliminate the field level of specification in NJT specifications. Users of DERIVE need only worry about an interface that is at the level of assembly language (the level at which NJT’s “constructors” are written). Removing such extra levels of detail eliminates some potential sources of specification error.

The authors of NJT use the assembler to do testing of specified encodings [9], by choosing random values from within the space of encodings. We turn the process around and use the assembler to derive encodings. Because a user of NJT specifies the various classes of encodings, the NJT specification tester does not need to search as much of the encoding space as DERIVE does.

NJT supports several features that DERIVE does not, such as support for relocation. For targets such as JITs, which are our primary interest, relocation is not an issue. As another example, NJT can handle synthetic assembly instructions cleanly. The current implementation of DERIVE does not yet handle complex synthetic instructions well; however, we could easily layer a tool that described synthetic instructions on top of DERIVE.

3 Implementation

DERIVE takes a high-level description of an instruction set and an assembler, and generates C header files that describe the instruction set. Figure 1 illustrates the tool chain for DERIVE. `idc` is described in Section 3.1; the solver is described in Section 3.2; and the code emitter generator is described in Section 4.

DERIVE extracts the encoding of each instruction by sampling a small number of possible input sets. We assume each instruction takes a fixed number of fields as operands. We assume that fields have three types: registers, immediates, and labels/jump targets. Registers are a (usually small) set of textual names; immediates are integer values that can range over a large set of possibilities (e.g., on the x86, some instruction take several 32-bit immediates); and labels are symbolic instruction addresses. These assumptions are sufficient to describe the instruction sets for which we have written specifications (which cover a wide range of modern ISAs).

We make the following assumptions about instruction encodings:

1. Assembly encodings are *stateless* transformations. That is, a particular instruction always has the same encoding. In addition, assembling two correct instructions together gives the same results as concatenating the results of as-
Figure 1: An example tool chain that uses DERIVE. We have built the tools that are in double boxes.

2. We assume that the validity and encoding of a value in a field does not depend on the value of any other field. We can thus compute a field’s encoding by enumerating every legal value for it while holding all other fields constant. This assumption allows us to solve for each field separately. Some instructions in certain instruction sets (such as the ARM) do not satisfy this assumption, in which case DERIVE must do a more expensive search.

3. The field for an $n$-bit immediate whose high bit is set is independent of the low bits of the immediate. That is, constants between $2^n-1$ and $2^n-1$ can all be encoded in the same instruction field. This assumption lets us solve for immediate fields more quickly; otherwise, our solver would have to enumerate all legal operand values.

4. Immediates are encoded with “simple” transformations. “Simple” means that an immediate’s value can have leading or trailing zeros removed, constants subtracted from it, or leading one bits added (for sign extension). If DERIVE detects a complex transformation that it cannot represent (such as the scale factor in x86 memory instructions, which is encoded as the logarithm of the scale factor), it immediately stops and tells the user, who can provide DERIVE with the transformation explicitly. Our implementation currently makes the assumption that negative immediates are encoded using twos-complement; this assumption could be changed fairly easily.

3.1 Specifying Instructions

The front end to DERIVE, idc, translates assembly-level instruction set descriptions into the intermediate form that DERIVE uses. idc is about 900 lines of lex and yacc code. Users describe instructions with a yacc-like description that contains a list of instructions to generate and a format description that describes the assembly syntax and the operand types. Note that the formats are grouped by assembly syntax, not by encoding class; we believe that the former is far more readable, as it expresses logical relationships rather than encoding relationships.

The description of SPARC ALU instructions reads as follows. Note that the field-level information necessary with NJT does not have to be specified by the user.

```plaintext
iregs = ( %g0, %g1, %g2, %g3, %g4, %g5, %g6, %g7, %o0, %o1, %o2, %o3, %o4, %o5, %o6, %o7, %i0, %i1, %i2, %i3, %i4, %i5, %i6, %i7 );
and, andcc, andn, andncc, or, orcc, orn, orncc, xor, xorcc, xnor, xnorcc, sl1, srl, sra, add, addcc, addx, addxcc, taddcc, taddccctv, sub, subcc, subx, subcc, tsbucc, tsbuccctv, mulcc,
```
umul, smul, umulcc, smulcc, udiv, sdiv, udivcc, sdivcc, save, restore
-> &op\& r_1:iregs, r_2:iregs, r_dest:iregs
| &op& r_1:iregs, imm, r_dest:iregs |

The production describes the assembler syntax and the types of the operands for multiple instructions. It states that there are register-register and register-immediate forms of all of the instructions listed. The placeholder &op& indicates where the instruction name appears. The operand specification says that the formatting substring ‘r_1:iregs’ is replaced by members of the list of registers iregs. Several conventions apply: register field names begin with r, a jump target has the unique name &label&, and an immediate field is any other field.

3.2 The Solvers

DERIVE is composed of three solvers, each specialized to derive a specific operand type: the register solver solves for register fields/operands, the immediate solver solves for immediate fields, and the jump solver solves for jump target fields. Each solver uses the assembler to compute instruction encodings. DERIVE emits code into an assembler file, runs the assembler, and finds the code in the resulting executable. DERIVE tests for any difference in endianness between the target and the solving architectures, and appropriately swaps bytes in the object code before doing any solving.

To demarcate the machine code in the executable, DERIVE explicitly emits fenceposts around the code using assembler data directives. Our fenceposts are a randomly chosen sequence of bytes. It is unlikely that DERIVE emits a sequence of instructions that match the fencepost, since it does not exhaustively search the set of instructions: we only search the entire space of values for register fields, and not for immediate and jump fields. Although we do not currently do so, a simple way to detect and avoid fencepost conflicts would be to solve each instruction twice with different fencepost values.

DERIVE represents instruction encodings as an opcode mask and an arbitrary number of operand fields. The opcode mask contains the bitmask of 1's that must be set in an instruction to specify a given opcode. Register fields are specified as a sequence of masks, one for each legal register value; immediates and label fields are represented as a size, an offset, and a simple transform.

The solvers all work in the same basic manner. Each one locates a field and determines its size by emitting one instruction for each legal operand value of that field (while holding all other fields' values fixed) and finding all bits that change in the binary encoding. Figure 2 illustrates this process for the last operand of the SPARC add instruction, and the resulting binary instructions that it analyzes.

Figure 2: The assembly that DERIVE generates to solve for the last register field of the SPARC add instruction, and the resulting binary instructions that it analyzes.
move such dependencies by checking the generated code: `DERIVE` gets around this particular bug by making sure that each constant value examined actually results in a different instruction.

### 3.2.1 Register Solver

The register solver is `DERIVE`'s most basic solver, and is called by the other solvers. It has two tasks: First, it computes an instruction's opcode mask with respect to the register fields. The opcode mask consists of all of the bits that are not used by the register fields. Second, it finds both the location and size of each "register" operand field, along with the bitmask that must be set to specify a given operand value. A register operand is any operand for which the client enumerates the possible textual values.

The register solver works in the following manner for a particular register field:

1. Iterate over all legal registers in the field. For each register, create a copy of the instruction format string. Replace the operand with the register value, and all other operands with some legal values. Emit the instructions and read them back into a buffer `inst`

2. While iterating over each register value, incrementally reduce the opcode mask by ANDing it with each binary instruction:

   ```
   op_mask = "0; 
   foreach r in registers
   op_mask &= inst[r];
   ```

   This process will leave the mask with 1's in every bit that has a 1 set for all instruction instances.

3. Examine the emitted instruction stream, and look for all bits that change between 0 and 1. Such bits belong to the current field, since all other operand values were fixed, and all values for the field were enumerated.

   To find these bits, bitwise AND each instruction with the complement of the opcode mask and logically summing the result:

   ```
   field = 0;
   foreach r in registers
   field |= (inst[r] & ~op_mask);
   ```

   At the end, the `field` mask has a 1 set for every bit in the field. The size of the field is bounded by the most and least significant bits set in this computed mask. The field's offset is given by its least significant bit.

4. To derive the 1's that must be set to encode each register for the current field, iterate again over all legal registers, and bitwise AND each instruction with the computed field mask:

   ```
   foreach r in registers
   fmask[r] = inst[r] & field;
   ```

We have implemented two main extensions to this simple scheme. First, on some architectures, a specific register value can change the actual instruction encoding. For example, on the x86, different instruction forms are used when the `%eax` register is used as an operand. The solver detects such discontinuities by checking that all instances of an instruction are of the same length. That is, when it emits the sequence of instructions (as shown in Figure 2), it checks that the number of instruction bytes emitted equals the number of instructions multiplied by the size of one instruction. If it does not, it solves for the register values independently, and emits multiple instruction specifications: each specification identifies which specific register values it corresponds to.

Second, the solver allows users to supply register operand lists that contain illegal values, which it automatically nulls. This syntax is useful for situations where instructions only accept subsets of possible register values. For example, the SPARC architecture supports floating-point instructions that take different combinations of registers, depending on whether the inputs and outputs are single-, double-, or quad-precision. Specifying exactly which operands are legal would increase the size of the SPARC specification by about 40%, and would also increase the probability of error. Instead, clients can state that every floating-point instruction takes any floating-point register as an operand:

   ```
   fregs = ( %f0, %f1, %f2, %f3, %f4, %f5, %f6,
            %f7, %f8, %f9, %f10, %f11, %f12, %f13, %f14,
            %f15, %f16, %f17, %f18, %f19, %f20, %f21,
            %f22, %f23, %f24, %f25, %f26, %f27, %f28,
            %f29, %f30, %f31 );
   ```

   ```
   fadds, fsubs, fmuls, fdivs, faddd, fsubd,
   fmuld, fdivd, faddq, fsubq, fmulq, fdivq,
   fmuld, fmulq
   --> &op& r_1:fregs, r_2:fregs, r_3:fregs;
   ```

`DERIVE` automatically eliminates "bad" registers by first randomly selecting operand values until it finds a sequence that the assembler accepts. It then finds all legal values for a field by trying all of its values while holding the others operands fixed to legal values.
3.2.2 Immediate Solver

The immediate solver computes the width and position of each immediate field, and also any transformation of the values in immediate fields. The immediate solver is called after the jump solver, if there are relative jump targets; it is used directly to solve for absolute jump targets (since these are just transformed immediate values). We explain it before the jump solver, because its behavior is closer to that of the register solver.

The main difference between this solver and the register solver is that it is impractical to enumerate all legal values for an immediate operand: an \( n \)-bit immediate field would require \( 2^n \) permutations. Register fields tend to be small (on the order of 5 bits for modern architectures), whereas immediate fields can be substantially larger. As a result, we solve for each bit size of the immediate field, rather than each possible value.

The solver first finds an immediate field's size by iterating upwards from 1 bit, 2 bits, etc., until the assembler refuses to assemble the instruction. It then iterates down from the maximum number of bits (call it \( n \)) and solves for each immediate size.

The immediate solver works in the following manner for each bit size \( m \):

1. Choose a random \( m - 1 \) bit value and create two \( m \)-bit constants, \( v_0 \) and \( v_1 \), by setting \( v_0 \) to the value and \( v_1 \) to its complement. Then set the \( m \)th bit in both \( v_0 \) and \( v_1 \):

\[
\begin{align*}
x &= (1 << (m - 1)); \\
\# \text{randomize low m-1 bits} \\
v_0 &= \text{random()} \% x; \\
\# v_1 \text{ complements those bits} \\
v_1 &= \neg v_0 \% x; \\
\# \text{set mth bit in v0 and v1} \\
v_0 &= v_0 \| x; \\
v_1 &= v_1 \| x;
\end{align*}
\]

By emitting instructions with these two constants, we force the low \( m - 1 \) bits of the immediate fields to have complementary bit values, while all other bits remain constant. Next, create two instructions, the first with the immediate operand replaced with \( v_0 \), the second with \( v_1 \). Use the the register solver to derive the register field encodings for these two similar-looking instructions.

\[
\begin{align*}
\# \text{copy instruction} \\
\text{inst0} &= \text{inst1} = \text{inst}; \\
\# \text{replace immediate operand with v0} \\
\text{rewrite}(&\text{inst0.fmt, op.field name, v0}); \\
\# \text{replace immediate operand with v1} \\
\text{rewrite}(&\text{inst1.fmt, op.field name, v1}); \\
\# \text{solve each copy} \\
\text{register_solve}(s0, inst0); \\
\text{register_solve}(s1, inst1);
\end{align*}
\]

The two specifications \( s_0 \) and \( s_1 \) should have the same register masks, and the same size in bytes. The only difference between the two should be the opcode masks computed by the register solver, which will differ by exactly the bits that differ in \( v_0 \) and \( v_1 \).

2. Find the immediate field by first XORing the two opcode masks. Since only the lower \( m - 1 \) bits of the field differ, this action sets 1's exactly in the location of these bits and 0's everywhere else; we add the \( m \)th bit in explicitly (note that we assume that this bit is contiguous with the rest of the field). The least significant bit gives the field's offset. Refine the opcode mask by removing all field bits from it.

\[
\begin{align*}
\# \text{set the low m-1 bits in field} \\
\text{field_bits} &= s0.\text{op_mask} \& s1.\text{op_mask}; \\
\# \text{find least significant bit} \\
\text{field_offset} &= \text{lsb(field_bits)}; \\
\# \text{add back mth field bit} \\
\text{field_bits} &= 1 << (m + \text{field_offset} - 1); \\
\# \text{fix opcode mask} \\
s0.\text{op_mask} &= s0.\text{op_mask} \& \neg \text{field_bits};
\end{align*}
\]

3. Check the value of the field against the value of the encoded field to see if any simple transformations are used, such as a shift to the right by a small constant.

4. Check to see if a previous encoding matches this one. Since we work our way down from larger-valued immediates to smaller-valued immediates, we may have already discovered the encoding for the field. For example, a SPARC 13-bit signed immediate field encodes all signed values between 1 bit and 13 bits. In contrast, on the x86 4-byte memory displacements are encoded differently than 2-byte memory displacements.

If we have already found an encoding, the current encoding is ignored; otherwise, it is added to the list of derived encodings. We must evaluate the encoding for each immediate size, because some instruction encodings on certain architectures (x86) vary with the size of the immediate provided.

Our actual solver is more general than this sketch, and handles instructions with an arbitrary number of immediate operands. For instructions with more
than one immediate operand, any variable-length immediate fields are not independent of each other—the length of one will affect the position of any others. Therefore, DERIVE cannot solve for immediate fields separately.

The current implementation has two limitations. First, it assumes that every immediate value that fits in a given field is legal: that is, there are no “holes” in the value space for an immediate field. This restriction is not a real problem. Second, it does not handle immediatess that are encoded as multiple non-contiguous bit ranges. It should not be difficult to extend DERIVE to handle such immediatess.

### 3.2.3 Jump Solver

The jump solver derives the encoding for relative jump target fields (labels). Jumps can be classified in two ways: (1) relative vs. absolute jumps and (2) jumps that take immediates vs. those that only take labels. The jump solver finds encodings for relative jumps that take labels as operands. Jumps that accept immediate operands are handled by invoking the immediate solver. Absolute jumps that only take labels do not seem to occur in practice.

The difference between the jump solver and the immediate solver is that the jump solver must generates values (labels) differently. To set all bits in an $n$-bit immediate field, the immediate solver can emit the immediate directly; the jump solver may have to place a label about $n$ instructions away. As a result, it would be impractical to solve for large offsets directly. We assume that backwards jumps are encoded using two-complement, so that we can solve for the sizes of offsets.

DERIVE computes whether a jump is absolute or relative by emitting two consecutive jumps to the same target and comparing the emitted code values. Absolute jumps will have identical bits, since both instructions encode the same target address. Relative jumps will differ, since they are different distances from the target and thus will have different offset values.

Given an instruction that is a relative jump, the solver must find the jump target field’s width and offset in the instruction, starting point (how many bytes a jump of 0 bytes actually jumps from the jump instruction), and minimum jump size. The target address of the jump is $\text{target} = \text{start} + \text{encoded field} \times \text{jump size}$.

1. Find the minimum jump size by emitting jumps of 1, 2, ..., bytes until the jump field changes.

2. Find the starting point for the jump (the placement of the label that results in an offset of 0). The starting point is found by searching for a label placement that results in an instruction $i$, where ORing against $i$ is the identity function. In other words, emit jumps to different labels around the jump, and OR each of the jumps against the others to find one whose offset field must be all 0's.

3. Find the label’s offset in the instruction by emitting a forward jump just past the starting point for the jump. These two instructions will differ by a single bit, which is the lowest bit in the target field.

4. Find the label’s size in the instruction by emitting a negative-offset label. Assuming that relative jumps are encoded using two-complement (or even ones-complement), this label sets the high bit of the label. We assume that the label is contiguous in the instruction, and that it consists of the bits between the label’s high and low bits.

5. Finally, determine how the jump distance is encoded by referencing labels at known offsets and comparing the field’s value to these locations. We check for the following transformations: subtraction by a constant, truncation of trailing zeros, or having leading bits truncated. For example, truncation of trailing zeros from a byte offset happens in SPARC jump instructions, since instructions are word-aligned.

Like the other solvers, the jump solver assumes that label fields are contiguous. It would be challenging to deal with non-contiguous label fields, because of the need to generate labels at large distances from jumps.

### 3.3 User Extensions

Some instruction sets (such as the ARM) do not satisfy all of the assumptions we have described. In addition, incorrect assemblers can provide bad information to DERIVE. We give the user mechanisms to address these problems: the user can inform DERIVE of complex immediate encodings, tell it when register field values may depend on each other, and provide explicit field widths when necessary. Table 1 lists the cases where we have needed to use these mechanisms.

The following description fragment shows how a user can specify complex encodings. The user provides C code that translates between the value that is mapped into the immediate field and the input
that the assembler expects. In this example, the x86 instruction encodes the logarithm of the scale factor that is given to the assembler. (This example could also be handled by making the scale factor a register type and enumerating the possible scale values.)

%
unsigned pow2(unsigned x) { return 1 << x; }
%

ops_2_mem --> &op& r_1:regs,
disp(base:regs, index:index_regs, scale:pow2);

The following example shows how a user provides information about non-independent fields. For the PowerPC instructions mentioned, the base register ra must not be the same as the target register rd.

%
bool update_disp(char *args[]) {
char *rd = args[0], *ra = args[1];
if (!strcmp(ra, "r0") || !strcmp(ra, rd))
    return FALSE;
else return TRUE;
}
%

lhzu, lhzu, lhu, lwz --> &op& update_disp R_d:regs, disp(R_a:regs);

Finally, DERIVE provides hooks that enable the user to overcome some assembler bugs that we have encountered. For example, GNU as allows the SPARC V9 ticc instruction to take an immediate that is too large. DERIVE reports that fields overlap, and the user can explicitly tell DERIVE the field width of 7 bits as follows:

Table 1: Exceptions that we have found to our encoding model. DERIVE’s hooks let clients easily extend its model to handle these cases. For fields that depend on each other, users annotate dependent registers in their specification, and supply a function that takes a list of symbolic register names and returns TRUE if they are a legal combination. Users add missing transformations by providing a function that takes an immediate and returns the transformed value, and annotating immediates that use this encoding in the specification.

<table>
<thead>
<tr>
<th>Violated Assumption</th>
<th>Architecture</th>
<th>Instruction Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fields values</td>
<td>ARM</td>
<td>Register pre-/post-indexed addressing modes</td>
</tr>
<tr>
<td>are independent</td>
<td></td>
<td>Base and index registers must differ</td>
</tr>
<tr>
<td></td>
<td>PowerPC</td>
<td>Load multiple instructions</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Address register must not be a target</td>
</tr>
<tr>
<td>Simple transforms</td>
<td>x86</td>
<td>Scale factor in memory addressing</td>
</tr>
<tr>
<td></td>
<td>SPARC</td>
<td>Index is encoded as a log</td>
</tr>
<tr>
<td></td>
<td>ARM</td>
<td>Certain addressing modes expect offset x4</td>
</tr>
</tbody>
</table>

Table 2: The time it takes derive to run through each architecture, and how long our architecture descriptions are. x86-kaffe is the subset of x86 we needed to retarget Kaffes’s JIT to use derive-generated emitters.

tgu, tleu, tcc, tcs, tsp, tneg, tvc, tvs
--&op& r_c:cc, imm:7;

3.4 Using DERIVE

Table 2 summarizes the times it takes for DERIVE to run on several instruction sets, and also shows the length of DERIVE’s specifications. As the number of architectures in the table shows, DERIVE’s assumptions survive well under use. The assumptions in our model make DERIVE reasonably fast. ARM is slow because some of its instruction addressing modes violate the independence assumption. As a result, solving those instructions takes an inordinate amount of time, because DERIVE must check all combinations of register values. x86 is slow because of the large number of instructions and addressing...
modes, as well as the special encodings for certain registers. The subset of the x86 ISA necessary to retarget the Kaffe JVM’s [25] JIT was small enough to run quickly.

While using DERIVE to reverse-engineer several instruction sets, we have come across several errors or inconsistencies in gas and various architectural manuals. The following list describes these errors and inconsistencies, and demonstrates that DERIVE’s reverse-engineering methodology can also be viewed as a useful testing methodology.

- GNU as does not assemble the Alpha wh64 instruction.
- GNU as does not handle the Alpha rounding/trapping modes of floating-point sqrt instructions.
- GNU as on MIPS silently truncates the top bits of absolute addresses larger than 28 bits.
- GNU as does not quite handle setting the user mode bit in ARM addressing mode 4 correctly.
- GNU as accepts immediates that are too large for SPARC ticc instructions.
- GNU as often accepts n-bit positive values for n-bit signed immediate operands.
- “See MIPS Run” [22], Table 8.6, is incorrect for mtcl and dtcl.
- In the ARM manual [13], addressing mode 3 of register pre-/post-indexed instructions do not have the same listed restrictions as those same instructions in addressing mode 2, although gas enforces those restrictions.
- The Alpha manual [2] description of the cvtst (IEEE conversion) instruction seems incorrect, because it lists the /s suffix (a VAX rounding mode) as an option.

4 Using DERIVE’s Output

An important motivation behind building DERIVE was our desire to avoid hand-specifying the x86 instruction set. This distaste was an important reason why we have not retargeted two of our JIT systems [7, 8] to the x86, despite repeated requests.

One use of DERIVE-generated specifications is to generate code emitters from them. We have written an emitter generator that processes instruction specifications and generates C procedures or macros that can emit instructions into a code buffer. Figure 3 shows the structure declaration for DERIVE’s output. Figure 4 shows a sample specification that DERIVE generates for the MIPS break instruction, which causes a breakpoint. Each instruction specification is transformed into a function or macro whose arguments are an index into the code buffer and the registers, immediates, or labels that are operands for that instruction. The choice of macros allows the compiler to propagate constants if parameters, such as register values, are known at compile-time.

The following example shows the macro generated for the x86 addl instruction, and how it is used:

```c
#define E_addl_rr_1(_code, rf, rt) do { \
    register unsigned short _0 = (0x0001\ 
    | (((rf)) << 11))\ 
    | (((rt)) << 8));\ 
    *(unsigned short*)((char*)_code) = _0;\ 
    _code = (void *)((char*)_code + 2);\ 
    while(0)
} while(0)
/* emit "addl %ecx, %ebx" in code_buffer */
E_addl_rr_1(code_buffer, RREGcx, RREGbx);
```

With simple heuristics, we are able to generate code automatically that in most cases is as efficient and readable as the code a human would write. There are two challenges in generating efficient emitter code. First, we try to keep the number of arithmetic and bit shift operations on the instruction’s operands small. Second, we try to minimize the number of stores to the code buffer. For architectures with constant instruction lengths, such as all RISC architectures, our generated emitters incur one memory write per instruction generated.

Operands to an instruction are not always known when the instruction is generated. Dynamic code generators, for example, do not know the value of a forward-referenced label. Such a label must be patched later by a simple linker, once the actual value of the operand becomes known. DERIVE allows instruction operands to be marked so that they are not used in the emitter macro. Instead, the emitter generator generates an additional macro that can be called to fill in the operand. DERIVE clients can build their own linkers on top of this mechanism.

Our emitter generator can emit extra debugging information. First, our emitter generator is able to generate emitters that check the validity of their arguments. This feature is useful to catch bugs as early as possible during dynamic code generation. In addition, the generator can also produce emitters that print a textual description of each instruction generated during dynamic code generation.
struct inst_spec {
    char *inst;       /* instruction name */
    char *fmt;        /* assembly format */
    unsigned short n_ops;   /* operand count */
    unsigned nbytes;    /* instruction size */
    char *namesuffix;   /* suffix used by emitter generator */
    unsigned char op_mask[MAX_BYTES]; /* opcode mask */
}

struct operand {
    char *sym_op;     /* symbolic name of operand */
    enum op_type { REG, IMME, LBL_R, LBL_A } type; /* operand type */
    t_type encoding;  /* is immediate transformed? */
    unsigned lo;      /* lowest legal value of field */
    unsigned nbits;   /* number of bits in field */
    unsigned mask[MAX_REGS]; /* field mask */
    int offset;       /* offset in instruction */
    sign_type signed_field; /* signed or unsigned field? */
    int relative_offset; /* where relative jumps start */
    from: offset from end of jump */
    int wants_ref;    /* generate a separate emitter */
    to set this field */
}

Figure 3: The C encoding description that DERIVE outputs. It includes the instruction, formatting string used to generate the instruction, the instruction mask, and a list of operand specifications. t_type is an enum that represents some simple transformations on immediates. The output of DERIVE could be modified for use with other languages.

{  "break",        /* instruction name */
    ",op& imm",    /* assembly format */
    1,             /* operand count */
    4,             /* instruction size */
    ",",           /* suffix used by emitter generator */
    0xd, 0x0, 0x0, 0x0, }, /* opcode mask */
    {  { "imm",     /* name of operand */
        IMME,      /* type of operand */
        IDENT,    /* operand transformation */
        0,        /* lowest legal value */
        10,       /* number of bits */
        { 0 },    /* field mask */
        16,       /* offset in instruction */
        I_UNSIGNED,     /* unsigned */
        0,        /* ignored for non-jumps */
        0        /* do not generate an extra emitter */
    },

Figure 4: DERIVE-generated specification for the MIPS break instruction.
The emitters are also useful as a means of testing DERIVE. For each instruction, we generate the emitter and a test program that invokes it with a given subset of parameters. The instruction encodings generated by the emitter function are then compared to the output generated by the target assembler. This procedure allows us to detect DERIVE without actually running code on the target platform. DERIVE’s emitter generator can cross-compile between architectures whose endianness matches.

To demonstrate that our tools work, we have re-targeted Kaffe’s [25] x86 JIT backend to use automatically generated emitters from just a subset of the x86 ISA. We reduced the number of lines in the backend description from 2,084 to 1,267. We also discovered that the original coders missed shorter instruction encodings in one case. Retargeting the JIT took approximately one day, which indicates that the emitter functions generated by DERIVE are usable in real applications.

DERIVE-generated emitters can be used in several other ways to support dynamic code generation. For example, they can be used to support back end construction for general-purpose dynamic code generation systems such as vcode [8] and ccg [17]. These systems provide assembly-like interfaces to C for dynamic code generation, and need encoding information to actually generate code. DERIVE could also be used to generate code for application-specific systems such as DPF [7]. Templates can be specified in terms of symbolic instruction sequences, fed to DERIVE to get the corresponding binary encoders, and then incorporated into the system.

5 Conclusions

The DERIVE system reverse-engineers instruction encodings from the system assembler. Users need only give assembly-level information about the instructions for which they want encodings, and not low-level information about bitfield layout. DERIVE successfully reverse-engineers instruction encodings on the SPARC, MIPS, Alpha, ARM, PowerPC, and x86. In the last case, it handles variable-sized instructions, large instructions (16 bytes), multiple instruction encodings determined by operand size, and other CISC features. As a proof of its utility, we have built a code emitter generator on top of DERIVE.

We plan to extend DERIVE’s techniques to reverse-engineer object code file formats, including debugging and linkage information. Such information will enable us to build a set of reverse-engineered tools, including versions of ATOM [21], dynamic linking libraries [11], object-level sandboxes [23], executable optimizers, and linkers. Builders of such tools are plagued by the need to repeatedly reimplement functionality contained in existing software. For some systems, it is too expensive to call existing programs: dynamic code generation systems cannot afford the time to call an assembler. In other cases, the software has an inappropriate form and must be rewritten from scratch. For example, one common “trick” that commercial companies use to discourage third-party vendors is to have proprietary symbol table layouts, which change on every software release [16].

The cost of manually reverse-engineering these formats has forced some implementors to avoid object-level modifications, in spite of the strong advantages for such an approach.

The source code for the current version of DERIVE is freely available at the following URL: http://www.cs.utah.edu/~wilson/derive.tar.gz.

Acknowledgments

We thank John Carter, David Chen, Eddie Kohler, Max Poletto, Patrick Tullmann, various anonymous referees, and our shepherd Vern Paxson for their comments on earlier drafts of this paper. Wilson Hsieh was supported in part by an NSF CAREER award, CCR-9876117. Dawson Engler was supported in part by DARPA contract MDA904-98-C-A933 and by a Terman Fellowship. Godmar Back was supported by DARPA contract F30602-99-0503.

References


A MIPS description

{%
char *directive = "\n.set noreorder
.set nomacro" 
"\n.set noret" ;
%
}

regs = ($0, $1, $2, $3, $4, $5, $6, $7, $8, $9, 
$10, $11, $12, $13, $14, $15, $16, $17, 
$18, $19, $20, $21, $22, $23, $24, $25, 
$26, $27, $28, $29, $30, $31 );

fcond = ($fc0, $fc1, $fc2, $fccc, $fccc0, 
$fccc5, $fccc7 );

nop, nvec, tblr, tblw, tblr, tblp, eret --> &op&

movf, movt --> &op& r豸d:regs, r_s:regs, r_c:fcond;

jr, jal, mhi, mthi, mfl0, mto --> &op& r_s:regs;

jalr, tge, tgeu, tilt, tiltu, teq, tne, mfc0, dfc0, 
cfo, mtc0, dmc0, ctco, cfc0, mfc1, dfc2, 
cfo2, etc2, mfc2, dfc4, mfc5, 
mcfc6, &fccc7 ;

// MIPS assemblers use "div" as a macro
// this syntax is how the real hardware
// instructions of the same names can be generated

div, divu, ddiv, ddivu --> &op& $0", r_w:regs, r_s:regs;

break, syscall --> &op& imm;

j, jal --> &op& &label&;

bltz, bgez, bgtz, bgezl, bgtzl, bgezl, bgtzl, 
bgezl, bnez, bnez, bgtzl, bnez, bgtzl --> &op& r_s:regs, &refk &label&;

tgei, tgeiu, ttil, ttilu, teqi, tnei --> &op& r_s:regs, imm;

beq, bne, beqi, bnei --> &op& r_s:regs, r_t:regs, &label&; 

addu, addiu, slti, sltiu, andi, ori, xori, lui --> &op& r_d:regs, &refk imm;

daddi, daddiu --> &op& r_d:regs, r_w:regs, &refk imm;

// we can define this form once and use it
// in different places (denoted by a single arrow)

branch --> &op& &label&;

branchc --> &op& r_s:fcmd, &label&;

// these are coprocessor branches
bc0f, bc0t, bc0fl, bc0tl, bc2f, bc2t, bc2fl, bc2tl: branch;

// these instructions come in two flavors
bc1f, bc1t, bc1fl, bc1tl: branch, branchc;

add.s, add.d, sub.s, sub.d, mul.s, mul.d, div.s, 
div.d --> &op& r_d:regs, r_v:regs, r_s:regs;

mfc1, mfc2, mfc3, ctcl, 
--> &op& r_t:regs, r_v:regs;

mqts, mqtr, dabs, abs.d, movs, mov.d, 
neg.s, neg.d, round.l.s, round.l.d, trunc.l.s, 
trunc.l.d, ceil.s, ceil.d, floor.l.s, 
floor.l.d, round.w.s, round.w.d, trunc.w.s, 
trunc.w.d, ceil.w.s, ceil.w.d, floor.w.s, 
floor.w.d, recip.s, recip.d, rsqrt.s, rsqrt.d, 
ctv.s.d, cvt.w.s, cvt.s.w, cvt.w.d, 
ctv.d.l, cvt.w.d, ctv.l.s, ctv.l.d 
--> &op& r_d:regs, r_v:regs;

movs, movt, movf, movd --> &op& r_d:regs, r_v:regs, r_s:fcmd;

movz.s, movz.d, movn.s, movn.d 
--> &op& r_d:regs, r_v:regs, r_s:regs;

cfs, cfq, cun, cun.d, cseq.s, cseq.d, 
cseq.w, cseqq, cseqq.d, cseals, cseal.d, 
cseal.s, csealw, csealq, csealqd, 
cfms, cfmd, cseq.s, cseq.d, cngls, cngld, 
cnglw, cngqd, cngql, cngql.d, cngqd.l, 
cngls.l, cngls.d, cnglw.d, cngld.l, 
cngld.d, cngls.w, cnglw.w, cngld.w, 
ctv.s.d, ctv.w.s, ctv.s.w, ctv.w.d, 
ctv.d.l, ctv.w.d, ctv.l.s, ctv.l.d 
--> &op& r_d:cond, r_v:regs, r_s:regs;

// the assembler expects parentheses
lwxc1, ldxcl, swc1, mfc1 
--> &op& r_d:regs, r_v:regs ( r_s:regs );

// prefetch instructions with hints
pref --> &op& r_h:hints, imm ( r_v:regs );
prefx --> &op& r_h:hints, r_v:regs ( r_s:regs );
hints = ( 0, 1, 4, 5, 6, 7 );

madd.s, madd.d, msub.s, msub.d, mmadd.s, mmadd.d, 
mmsub.s, mmsub.d 
--> &op& r_d:regs, r_r:regs, r_s:regs, r_t:regs;

ldl, ldr, lb, lh, lwl, lwu, lhu, lwr, lwu, sb, 
wv, ml, sv, mlr, mv, 11, lwv, 12, ldr, ldc2, 
ldc, sc, mv2, scd, mdc2, sd 
--> &op& r_d:regs, &refk imm ( r_v:regs );

l.s, l.d, s.s, wc1, r.d, rdc 
--> &op& r_d:regs, imm ( r_v:regs );

cache --> &op& r_d:cache_ops, imm ( r_v:regs );
cache_ops = ( 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 
11, 13, 15, 16, 17, 18, 19, 20, 21, 
23, 24, 26, 27, 30, 31 );
}