Available online at www.sciencedirect.com
ScienceDirect
Science of Computer Programming 70 (2008) 127–148
Science of Computer Programming
www.elsevier.com/locate/scico

An assembler and disassembler framework for Java™ programmers

Bernd Mathiske*, Doug Simon, Dave Ungar

Sun Microsystems Laboratories, 16 Network Circle, Menlo Park, CA 94025, USA

Received 1 December 2006; received in revised form 1 February 2007; accepted 11 July 2007
Available online 24 October 2007

Abstract

The Java™ programming language is primarily used for platform-independent programming. Yet it also offers many productivity, maintainability and performance benefits for platform-specific functions, such as the generation of machine code.

We have created reliable assemblers for SPARC™ , AMD64, IA32 and PowerPC which support all user mode and privileged instructions and with 64 bit mode support for all but the latter. These assemblers are generated as Java source code by our extensible assembler framework, which itself is written in the Java language. The assembler generator also produces javadoc comments that precisely specify the legal values for each operand.

Our design is based on the Klein Assembler System written in Self. Assemblers are generated from a specification, as are table-driven disassemblers and unit tests. The specifications that drive the generators are expressed as Java language objects. Thus no extra parsers are needed and developers do not need to learn any new syntax to extend the framework for additional ISAs.

Every generated assembler is tested against a preexisting assembler by comparing the output of both. Each instruction's test cases are derived from the cross product of its potential operand values. The majority of tests are positive (i.e., result in a legal instruction encoding). The framework also generates negative tests, which are expected to cause an error detection by an assembler. As with the Klein Assembler System, we have found bugs in the external assemblers as well as in ISA reference manuals.

Our framework generates tens of millions of tests. For symbolic operands, our tests include all applicable predefined constants. For integral operands, the important boundary values, such as the respective minimum, maximum, 0, 1 and −1, are tested. Full testing can take hours to run but gives us a high degree of confidence regarding correctness.

Keywords: Cross assembler; Assembler generator; Disassembler; Automated testing; The Java language; Domain-specific framework; Systems programming

1. Introduction and motivation

Even though the Java programming language is designed for platform-independent programming, many of its attractions[1] are clearly more generally applicable and thus also carry over to platform-specific tasks. For instance, popular integrated development environments (IDEs) that are written in the Java language have been extended (see e.g. [6])

* Corresponding author.
E-mail addresses: Bernd.Mathiske@sun.com (B. Mathiske), Doug.Simon@sun.com (D. Simon), David.Ungar@sun.com (D. Ungar).

[1] To name just a few: automatic memory management, generic static typing, object orientation, exception handling, excellent IDE support, large collection of standard libraries.

0167-6423/$ - see front matter
doi:10.1016/j.scico.2007.07.007

to support development in languages such as C/C++, which get statically compiled to platform-specific machine code. Except for legacy program reuse, we see no reason why compilers in such an environment should not enjoy all the usual advantages attributed to developing software in the Java language (in contrast to C/C++). Furthermore, several Java virtual machines have been written in the Java language (e.g., [3,22,15]), including compilers from byte code to machine code.

With the contributions presented in this paper we intend to encourage and support further compiler construction research and development in Java. Our software relieves programmers of arguably the most platform-specific task of all, the correct generation of machine instructions adhering to existing general purpose instruction set architecture (ISA) specifications.

We focus on this low-level issue in clean separation from any higher level tasks such as instruction selection, instruction scheduling, addressing mode selection, register allocation, or any kind of optimization. This separation of concerns allows us to match our specifications directly and uniformly to existing documentation (reference manuals) and to exploit pre-existing textual assemblers for *systematic, comprehensive* testing. Thus our system virtually eliminates an entire class of particularly hard-to-find bugs and users gain a fundament of trust to build further compiler layers upon.

Considering different approaches for building assemblers, we encounter these categories:

Stand-alone assembler programs: These take textual input and produce a binary output file. Compared to the other two variants they are relatively slow and they have long startup times. Therefore stand-alone assemblers are primarily used for static code generation. On the plus side, they typically offer the richest feature sets beyond mere instruction encoding: object file format output, segmentation and linkage directives, data and code alignment, macros and much more.

Inline assemblers: Some compilers for higher programming languages (HLLs) such as C/C++ provide direct embedding of assembly language source code in HLL source code. Typically they also have syntactic provisions to address and manipulate HLL entities in assembly code.

Assembler libraries: These are aggregations of HLL routines that emit binary assembly instructions (e.g., [10,14]). Their features may not always be directly congruent with textual assembly language. For example, many methods in the x86 assembler library that is part of the HotSpot™ Java virtual machine [19], which is written in C++, take generalized location descriptors as parameters instead of explicitly broken down operands. Further along this path, the distinction between a mere assembler and the following category becomes quite diffuse.

Code generator libraries: These integrate assembling with higher level tasks, typically instruction selection and scheduling, arranging ABI compliance, etc., or even some code optimization techniques.

We observed that only the first of the above categories of assemblers is readily available to today's Java programmers: that is, stand-alone assemblers, which can be invoked by the `System.exec()` method. This approach may be sufficient for some limited static code generation purposes, but it suffers from the lack of language integration and the administrative overhead resulting from having separate programs. Regarding dynamic code generation, the startup costs of external assembler processes are virtually prohibitive.

We are not aware of any inline assembler in any Java compiler and to our knowledge there are only very few assembler libraries in the form of Java packages.[2]

According to our argument above concerning the separation of concerns, building a code generator framework would have to be an extension of, rather than an alternative to, our contributions.

In this paper we present a new assembler library (in the form of Java packages) that covers the most popular instruction sets (for any systems larger than handhelds): SPARC, PowerPC, AMD64 and IA32. In addition our library includes matching disassemblers, comprehensive assembler test programs, and an extensible framework with specification-driven generators for all parts that depend on ISA specifics. All of the above together comprise the Project Maxwell Assembler System (PMAS).

The design of the PMAS is to a large extent derived from the Klein Assembler System (KAS), which has been developed previously by some of us as part of the Klein Virtual Machine [20]. The KAS, which supports the two RISC ISAs SPARC and PowerPC, is written in Self [1], a prototype-based, object-oriented language with *dynamic* typing.

[2] We exclude Java byte code "assemblers", since they do not generate hardware instructions.

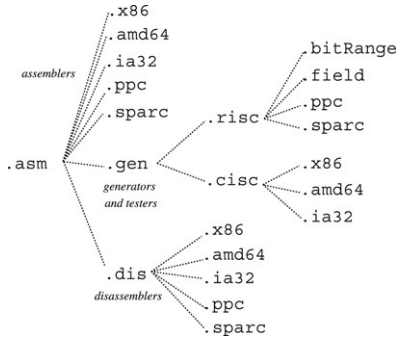


Fig. 1. Overview of the PMAS packages.

Providing an assembler with a programmatic interface already delivers a significant efficiency gain over a textual input based assembler as the cost of parsing is incurred during Java source compilation instead of during the execution of the assembler. In addition, we will discuss how making appropriate usage of Java's type system can also shift some of the cost of input validation to the Java source compiler.

2. Overview

Fig. 1 gives an overview of the PMAS package structure.[3] The `.gen` package and its subpackages contain ISA descriptions and miscellaneous generator and test programs. The `.dis` subtree implements disassemblers, reusing the `.gen` subtree. The other five direct subpackages of `.asm` contain assemblers, which have no references to the above two subtrees. Each package ending in `.x86` provides shared classes for either colocated `.amd64` and `.ia32` packages.

It is straightforward to reduce the framework to support any subset of ISAs, simply by gathering only those packages that do not have any names which pertain only to excluded ISAs.

The next section explains how to use the generated assemblers in Java programs. For each assembler there is a complementary disassembler, as described in Section 4. We describe the framework that creates these assemblers and disassemblers in Section 5. First we introduce the structure of our ISA representations (Section 5.1), then we sketch the instruction templates (Section 5.2) that are derived from them. We regard the latter as the centerpiece of our framework, as it is shared among its three main functional units, which provide the topics of the subsequent three sections: the structure of generated assembler source code (Section 5.3), then the main disassembler algorithms (Section 5.4) and fully automated assembler and disassembler testing (Section 5.5). Section 6 sketches how one would go about adding another ISA to the system. We briefly discuss related work in Section 7 and the paper concludes with notable observations and future work (Section 8).

3. How to use the assemblers

Each assembler consists of the top level package `com.sun.max.asm` and the subpackage matching its ISA as listed in Fig. 1. In addition, the package `com.sun.max.asm.x86` is shared between the AMD64 and the IA32 assembler. Hence, to use the AMD64 assembler the following packages are needed[4]: `com.sun.max.asm`, `com.sun.max.asm.amd64` and `com.sun.max.asm.x86`. None of the assemblers requires any of the packages under `.gen` and `.dis`.

To use an assembler, one starts by instantiating one of the leaf classes shown in Fig. 2. The top class `Assembler` provides common methods for all assemblers, concerning e.g. label binding and output to streams or byte arrays. The generated classes in the middle contain the ISA-specific assembly routines. For ease of use, these methods are purposefully closely oriented at existing assembly reference manuals, with method names that mimic mnemonics and parameters that directly correspond to individual symbolic and integral operands.

[3] We have split our source code into two separate IDE projects, one of which contains miscellaneous general purpose packages that are reused in other projects and that we regard as relatively basic extensions of the JDK. Here we only discuss packages unique to the PMAS.

[4] In addition, general purpose packages from MaxwellBase and the JRE are needed.

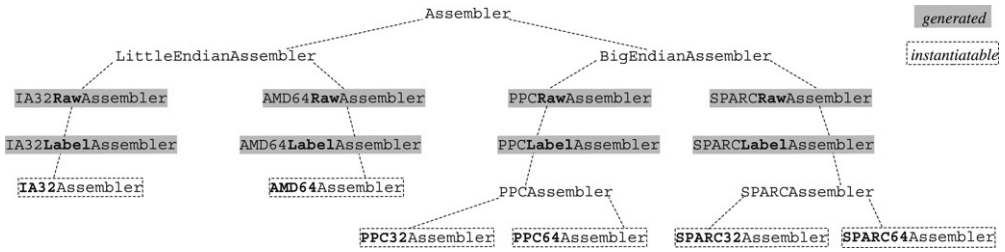


Fig. 2. Assembler class hierarchy.

```
memory address      offset  mnemonic  arguments                raw bytes in hex
0x0000000012345678  00      mov       rdx, [rsp + 12]          [48 8B 94 24 0C 00 00 00]
0x0000000012345680  08  L1: call      -305419345               [E8 AF AB CB ED]
0x0000000012345685  13      sub       rdx, rax                 [48 29 C2]
0x0000000012345688  16      cmpq      rdx, 0x0                 [48 81 FA 00 00 00 00]
0x000000001234568F  23      jnz       L1: -17                  [75 EF]
0x0000000012345691  25      mov       rcx[rdi * 8 + 20], rdx   [48 89 94 F9 14 00 00 00]
```

Fig. 3. Disassembled AMD64 instructions.

Here is an example for AMD64 that creates a small sequence of machine code instructions (shown in Fig. 3) in a Java byte array:

```
import static
    ...asm.amd64.AMD64GeneralRegister64.*;
    ...
public byte[] createInstructions() {
    long startAddress = 0x12345678L;
    AMD64Assembler asm =
        new AMD64Assembler(startAddress);

    Label loop = new Label();
    Label subroutine = new Label();
    asm.fixLabel(subroutine, 0x234L);

    asm.mov(RDX, 12, RSP.indirect());
    asm.bindLabel(loop);
    asm.call(subroutine);
    asm.sub(RDX, RAX);
    asm.cmpq(RDX, 0);
    asm.jnz(loop);

    asm.mov(20, RCX.base(), RDI.index(),
            SCALE_8, RDX);

    return asm.toByteArray();
}
```

Instead of using a byte array, assembler output can also be directed to a stream (e.g. to write to a file or into memory):

```
OutputStream stream = new ...Stream(...);
asm.output(stream);
```

The above example illustrates two different kinds of label usage. Label `loop` is bound to the instruction following the `bindLabel()` call. In contrast, label `subroutine` is bound to an absolute address. In both cases, the assembler creates PC-relative code, though, by computing the respective offset argument.[5] An explicit non-label argument can be expressed by using `int` (or sometimes `long`) values instead of labels, as in:

```
asm.call(200);
```

[5] In our current implementation, labels always generate PC-relative code, i.e. absolute addressing is only supported by the *raw* assemblers.

The variant of `call()` used here is defined in the *raw* assembler (`AMD64RawAssembler`) superclass of our assembler and it takes a "raw" int argument:

```
public void call(int rel32) { ... }
```

In contrast, the `call()` method used in the first example is defined in the *label* assembler (`AMD64LabelAssembler`), which sits between our assembler class and the raw assembler class:

```
public void call(Label label) {
  ... call(labelOffsetAsInt(label)); ...
}
```

This method builds on the raw `call()` method, as sketched in its body.

These methods, like many others, are syntactically differentiated by means of parameter overloading. This Java language feature is also leveraged to distinguish whether a register is used directly, indirectly, or in the role of a base or an index. For example, the expression `RSP.indirect()` above results in a different Java type than plain `RSP`, thus clarifying which addressing mode the given `mov` instruction must use. Similarly, `RCX.base()` specifies a register in the role of a base, etc.

If there is an argument with a relatively limited range of valid values, a matching *enum* class rather than a primitive Java type is defined as the parameter type. This is for instance the case regarding `SCALE_8` in the SIB addressing expression above. Its type is declared as follows:

```
public enum Scale ... {
  SCALE_1, SCALE_2, SCALE_4, SCALE_8;
  ...
}
```

Each RISC assembler features *synthetic* instructions according to the corresponding reference manual. For instance, one can write these statements to create some synthetic SPARC instructions [21]:

```
import static ...asm.sparc.GPR.*;

SPARC32Assembler asm = new SPARC32Assembler(...);
asm.nop();
asm.set(55, G3);
asm.inc(4, G7);
asm.retl();
...
```

Let's take a look at the generated source code of one of these methods:

```
/**
 * Pseudo-external assembler syntax:
 * {@code inc  }<i>simm13</i>, <i>rd</i>
 * <p>
 * Synthetic instruction equivalent to:
 * {@code add(rd.value(), simm13, rd)}
 * <p>
 * Constraint: {@code -4096 <= simm13 &&
     simm13 <= 4095}<br />
 *
 * @see #add(GPR, int, GPR)
 * @see "<a href="...</a> - Section G.3"
 */

public void inc(int simm13, GPR rd) {
  int instr = 0x80002000;
  checkConstraint(-4096 <= simm13 &&
                  simm13 <= 4095,
    "-4096 <= simm13 && simm13 <= 4095");
  instr |= ((rd.value() & 0x1f) << 14);
  instr |= (simm13 & 0x1fff);
  instr |= ((rd.value() & 0x1f) << 25);
```

```
  emitInt(instr);
}
```

As we see here, our assembler generator creates a javadoc comment disclosing the textual external assembler syntax of each instruction and pointing out the exact place ("section G.3") in the reference manual [21] to find a detailed instruction description.

In addition, given that the generator has a complete model of the classes being generated, we use the `@see` tag to create links in the javadoc to related elements. For example, in the example above a link is generated to show the original `add` instruction from which the synthetic `inc` instruction is derived.

As is often the case for RISC (but not x86) assembler methods, the `int` argument above is limited to a value range of less than 32 bits, here only 13. In such cases, we resort to dynamic checking resulting in runtime exceptions if an out-of-range argument is passed. In all other situations, our assemblers are statically type-safe.

4. How to use the disassemblers

Since performance is less critical for disassemblers than for assemblers, our disassemblers are not generated as Java source code. Instead they are manually written programs, which on every program restart rely on the PMAS framework to generate instruction *template* tables (see 5.2). Thus using a disassembler always requires loading the respective packages under `.gen` as well.

4.1. Textual disassembling

This Java statement sequence disassembles AMD64 instructions from an input stream, given a start address for PC-relative decoding, and delivers textual output to the console:

```
AMD64Disassembler disasm =
  new AMD64Disassembler(startAddress);
BufferedInputStream stream = ...;
  new BufferedInputStream(...);
disasm.scanAndPrint(stream, System.out);
```

Applied to the instructions in the AMD64 code example in Section 3, this produces the output in Fig. 3.

Our disassembled syntax for AMD64 (and IA32) is a blend of so-called Intel and AT&T syntax [7], with some modifications. The address rendering mimics C/Java syntax, which we find much more intuitive. Simple indexing is indicated by '[' and ']', similar to array access in the Java Programming Language:

```
eax[ebx]          // base[index]
```

Registers etc. are named as in Intel syntax, in lower case without AT&T's "%" prefix. Indirect access looks like indexing without a base (or with implicit base 0):

```
[ecx]             // [indirect]
```

Displacements are added/subtracted from the index/indirect operand:

```
ebp[eax - 12]     // base[index + displ.]
[esi + 100]       // [indirect + displ.]
```

Scale is displayed as multiplication of the index:

```
rax[rdx * 8]      // base[index * scale]
ecx[ebx * 4 + 2] // base[index * scale + d.]
```

Scale literals are either 2, 4 or 8. A scale of 1 is left implicit, i.e. not printed.

Whereas displacement literals and offset literals are rendered as signed decimal integer numbers, we prefer unsigned hexadecimal integer numbers for direct memory references (pointer literals) and immediate operands.

Unlike displacements, offsets do not have a space between the sign and the number, e.g.:

```
jmp +12
call -2048
```

This disambiguation also conveniently allows us to express RIP (Relative to Instruction Pointer) addressing as a combination of an offset operand and indirect addressing, e.g.:

```
add [+20], eax
mov ebx, [-200]
```

Operand order follows Intel syntax, placing destination operands to the left with source operands to their right.

Some mnemonics may have operand size suffixes as in AT&T (gas) syntax:

Suffix	Intel Size	Java Size	# bits
b	byte	byte	8
w	word	short	16
l	long word	int	32
q	quad word	long	64

Thus there is no need for operand size indicators (e.g. `DWORD PTR`) for pointers as in Intel syntax.

The disassembler displays synthetic labels for all target addresses within the disassembled address range that hit the start address of an instruction. Operands that coincide with such a label are displayed with the respective Label prepended as demonstrated by label `L1` in Fig. 3. Instructions that reference a label are printed giving both the label and its underlying raw value, e.g.:

```
jmp L1: +100
adc [L2: +128], ESI
```

The disassemblers for SPARC and PowerPC create textual output that is virtually identical to the syntax found in the reference manuals, except for also providing label synthesis. In any case, the source code that needs to be changed to adjust any disassembler's output to one's personal liking is fairly minimal.

4.2. Programmatic disassembling

The disassemblers can also be used to analyze and manipulate disassembled instructions programmatically independent of creating textual output.

Individual disassembled instructions are modelled by the class `DisassembledInstruction` and its subclasses, which specialize it for each individual instruction set.

The following statements disassemble one SPARC machine code instruction at a given known address:

```
SPARC32Disassembler disasm =
  new SPARC32Disassembler(startAddress);
BufferedInputStream stream = ...;
  new BufferedInputStream(...);

Sequence<SPARC32DisassembledInstruction>
  instructions =
    disassembler.scanOneInstruction(stream);
```

Since there isn't always a unique mapping of the machine code to a corresponding assembler instruction, the disassembler returns a collection containing all possible alternatives. Ultimately, it is left to the user to pick a preferred variant among these, though the ordering within the result can be predispositioned by the following methods.

```
public enum AbstractionPreference {
    RAW, SYNTHETIC;
}

public void
  setAbstractionPref(AbstractionPreference a);
```

This method sets a bias for the disassembler whether to report synthetic instructions before non-synthetic ones or vice versa.

```
public enum SpecificityPreference {
    LOW, HIGH;
}

public void
  setSpecificityPref(SpecificityPreference s);
```

This method sets the preference whether to report more specific or less specific instructions first. A disassembled instruction is regarded as more *specific* than another if it contains more bits that are predetermined when excluding explicit operands. In other words, its constant *opcode-like* content is larger, whereas the other instruction contains more parameterization bits.

Furthermore, there is a setting that suppresses reporting of instructions that do not match the anticipated number of operand parameters:

```
public void setExpectedNumberOfArgs(int n);
```

The above settings have not only been conceived for user convenience. They are essential to automatically direct the disassembler's bias when used in the assembler tester. Not resolving the ambiguity that is especially prevalent in RISC assembler notation would otherwise leave many instructions only randomly matching up with the disassembler's results.

Another situation that requires upfront settings as opposed to per instruction selection is applying the disassembler to a longer stretch of machine code at once.

```
Sequence<SPARC32DisassembledInstruction>
  instructions =
    disassembler.scan(bufferedInputStream);
```

This call returns disassembled instructions in sequence of their occurrence in the input stream. For each collection element, the choice between multiple possible instructions is resolved using the current settings according to the above methods. By default, the preference is for synthetic instructions of high specificity and the number of parameters is left unconstrained.

Each reported disassembled instruction carries the following information:

1. a byte array containing the bytes that comprise the machine instruction,
2. the instruction's offset from the disassembler's start address,
3. the template that describes the instruction,
4. a list of actual arguments that have been extracted from the machine code, locating parameters as indicated by the template.

Given the latter three items, the assembler tester can reassemble the same instruction and verify whether the resulting bytes equal those provided by the disassembled instruction.

5. The generator framework

Having described the use of our assemblers and disassemblers, we will now address how they are implemented and tested. The following subsections describe how our framework starts from the internal description of an ISA and derives an abstract representation of assembler methods called *templates*. These constitute the centerpiece of the assembler generators, the disassemblers and for fully automated testing.

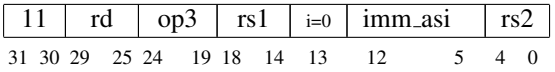
5.1. Constructing an ISA representation

We represent each ISA by a collection of instruction *descriptions* in the form of Java object arrays. Each of these specifies the exact composition of a group of closely related instructions.

The first object in every description must be a string which specifies the *external name*, i.e. the instruction mnemonic used in external assembler syntax. This string will also be used as the *internal name*, i.e., the base name for generated assembler methods, unless a second string is also given, which then defines a different internal name. For example, as the `return` instruction in the SPARC ISA clashes with a Java keyword, we gave it the internal name `return_`.

5.1.1. RISC instruction descriptions

A RISC instruction has 32 bits and it is typically specified as sequences of bit fields. See for example the specification of the `casa` instruction from the SPARC reference manual [21]:



Our system follows this structure by treating RISC instructions as a sequence of bit fields that either have constant values or an associated operand type. We have the following description types at our disposal:

RiscField: describes a bit range and how it relates to an operand. This implicitly specifies an assembly method parameter. A field may also append a suffix to the external and internal names.

RiscConstant: combines a field with a predefined value, which will occupy the field's bit range in each assembled instruction. This constitutes a part of the instruction's opcode.

InstructionConstraint: a predicate that constrains the legal combination of argument values an assembler method may accept. See Section 5.1.3 for more detail.

String: a piece of external assembler syntax (e.g., a parenthesis or a comma), that will be inserted among operands, at a place that corresponds to its relative position in the description.

Here is our specification of the `casa` instruction:

```
define("casa", op(0x3), op3(0x3c), "[", _rs1,
  "]␣", i(0), _imm_asi, ",", _rs2, _rd);
```

This specifies the name, the external syntax and the fields of the `casa` instruction. There are three constant fields (`RiscConstant`) `op`,[6] `op3` and `i`, as well as 4 variable fields (`RiscField`) `_rs1`, `_imm_asi`, `_rs2` and `_rd`. An example using the external syntax would be:

```
casa [G3]12, I5
```

The same field may be constant in some instructions, but variable in others. When writing field definitions for an ISA, one defines the same field in two ways, once as a Java value and once as a Java method.

```
public static final
 SymbolicOperandField<GPR> _rd =
  createSymbolicOperandField(GPR.SYMBOLIZER,
                             29, 25);

public static RiscConstant rd(GPR gpr) {
    return _rd.constant(gpr);
}

private static final ConstantField _op3 =
  createConstantField(24, 19);

public static RiscConstant op3(int value) {
    return _op3.constant(value);
}
...
public class ConstantField extends RiscField {
  RiscConstant constant(int value) {
      return new RiscConstant(this, value);
  }
}
```

This means that one can reference the return register field as a parameter operand (`_rd`) or as a constant field (e.g., `rd(G3)`). Field `op3` however, is always supposed to be constant, so we made only its method `public`.

[6] This field is not named in the reference manual diagram.

The following specifies a PowerPC instruction featuring an opcode field and 6 parameter fields:

```
define("rlwinm", opcd(21), _ra, _rs,
       _sh, _mb, _me, _rc);
```

In all instruction descriptions we apply the *static import* feature of the Java language, to avoid qualifying static field and method names. This greatly improved both the ease of writing descriptions and their readability. For instance, the above would otherwise have to be written as:

```
define("rlwinm", PPCFIelds.opcd(21),
    PPCFields._ra, PPCFields._rs,
    PPCFields._sh, PPCFields._mb,
    PPCFields._me, PPCFields._rc);
```

The PowerPC ISA is particularly replete with *synthetic* instructions, the specifications of which build on other instructions [12]. To match the structure of existing documentation closely, there is a `synthesize` method that derives a synthetic instruction from a previously defined (raw or synthetic) instruction. This method interprets its instruction description arguments to replace parameters of the referenced instruction with constants or alternative parameters. For example, we can define the *rotlwi* instruction by referring to the above *rlwinm* instruction:

```
synthesize("rotlwi", "rlwinm", sh(_n),
           mb(0), me(31), _n);
```

Here, we specify a new parameter field `_n` and cause the generated assembly method to assign its value to field `_sh`. The fields `_mb` and `_me` become constant with the given predefined values.

5.1.2. x86 instruction descriptions

The number of possible instructions in x86 ISAs is about an order of magnitude larger than in the given RISC ISAs. If one tried to follow the same approach to create instruction descriptions, one would spend an enormous amount of time just writing the description listings. More importantly, our primitives to specify RISC instructions are insufficient to express instruction prefixes, suffixes, intricate mod r/m relationships, etc. Instead of a rich bit-field structure, x86 instructions tend to have a byte-wise composition determined by numerous not quite orthogonal features.

As opcode tables provide the densest, most complete, well-publicized instruction set descriptions available for x86, we decided to build our descriptions and generators around those. For an x86 ISA, the symbolic constant values of the following description object types are verbatim from opcode tables found in x86 reference manuals (e.g., [13]):

AddressingMethodCode: We allow `M` to be used in lieu of the operand code `Mv` to faithfully mirror published opcode tables in our instruction descriptions.
OperandTypeCode: e.g. `b, d, v, z`. Specifies a mnemonic suffix for the external syntax.
OperandCode: the concatenation of an addressing mode code with an operand type code, e.g. Eb, Gv, Iz, specifies explicit operands, resulting in assembler method parameters.
RegisterOperandCode: e.g. `eAX, rDX`.
GeneralRegister: e.g. `BL, AX, ECX, R10`.
SegmentRegister: e.g. `ES, DS, GS`.
StackRegister: e.g. `ST, ST_1, ST_2`.

The latter three result in implicit operands, i.e. the generated assembler methods do not represent them by parameters. Instead we append an underscore and the respective operand to the method name. For example, the external assembly instruction `add EAX, 10` becomes `add_EAX(10)` when using the generated assembler. We also generate the variant with an explicit parameter that can be used as `add(EAX, 10)`, but that is a *different* instruction, which is one byte longer in the resulting binary form. External textual assemblers typically do not provide any way to express such choices.

In addition, these object types are used to describe x86 instructions:

HexByte: an enum providing hexadecimal unsigned byte values, used to specify an opcode. Every x86 instruction has either one or two of these. In the case of two, the first opcode must be `0F`.
ModRMGroup: specifies a table in which alternative additional sets of instruction description objects are located, indexed by the respective 3-bit opcode field in the mod r/m byte of each generated instruction.

ModCase: a 2-bit value to which the *mod* field of the *mod r/m* byte is then constrained.
FloatingPointOperandCode: a floating point operand not further described here.
Integer: an implicit byte operand to be appended to the instruction, typically `1`.
InstructionConstraint: same as for RISC above, but much more rarely used, since almost all integral x86 operand value ranges coincide with Java primitive types.

Given these features, we can almost trivially transcribe the "One Byte Opcode Map" for IA32:

```
define(_00, "ADD", Eb, Gb);
define(_01, "ADD", Ev, Gv);
...
define(_15, "ADC", eAX, Iv);
define(_16, "PUSH", SS);
...
define(_80, GROUP_1, b,
  Eb.excludeExternalTestArgs(AL), Ib);
...
define(_CA, "RETF",
       Iw).beNotExternallyTestable();
       // gas does not support segments
...
define(_6B, "IMUL", Gv, Ev,
       Ib.externalRange(0, 0x7f));
...
```

Many description objects and the respective result value of `define` have modification methods that convey special information to the generator and the tester. In the example above we see the exclusion of a register from testing, the exclusion of an entire instruction from testing and the restriction of an integer test argument to a certain value range. These features suppress already known testing errors that are merely due to restrictions, limited capabilities, or bugs in a given external assembler.

Analogous methods to the above are available for RISC instruction descriptions. For x86, however, there are additional methods that modify generator behavior to match details of the ISA specification which are not explicit in the opcode table. This occurs for example in the "Two Byte Opcode Table" for AMD64:

```
define(_0F, _80, "JO",
  Jz).setDefaultOperandSize(BITS_64);
...
define(_0F, _C7,
  GROUP_9a).requireAddressSize(BITS_32);
```

5.1.3. Instruction constraints

Instruction constraints are predicates that constrain the values an assembler method may accept. A predicate may apply to only one parameter (e.g. specifying the legal range of values of the parameter) or to a combination of parameters (e.g. two parameters may not have the same value). Constraints are specified by implementing the following interface:

```
interface InstructionConstraint {
  String asJavaExpression();

  boolean check(Template t,
    Sequence<Argument> args);

  boolean referencesParameter(Parameter p);

  Method predicateMethod();
}
```

The `asJavaExpression` method is used to generate the Java source code boolean expression that evaluates the constraint.

The `check` method is used by the tester to differentiate between valid and invalid test cases.

The `referencesParameter` method indicates whether a constraint involves a given parameter, which is relevant when deriving a synthetic instruction. In particular, if a parameter of an original instruction is removed or replaced in a derived synthetic instruction, any constraints that apply to the parameter must be removed from the synthetic instruction.

Predefined constraints express a relational test between two terms. Building on these, arbitrarily complex constraints can be constructed in the form of Java methods. The `predicateMethod` method enables cross linking in the javadoc for complex constraints.

An example of a predefined constraint is one that tests whether two parameters are not equal to each other. It can be created by the following factory method.

```
InstructionConstraint notEqual(final Parameter p1,
  final Parameter p2)
{
  return new InstructionConstraint() {
    public boolean check(Template t,
      Sequence<Argument> args) {
        return t.bindingFor(p1, args).asLong()
          != t.bindingFor(p2, args).asLong();
    }

    public String
      asJavaExpression() {
        return p1.valueString() + "!=" +
          p2.valueString();
    }

    public boolean
      referencesParameter(Parameter p) {
        return p == p1 || p == p2;
    }

    public Method predicateMethod() {
      return null;
    }
  };
}
```

The following factory method creates a complex constraint, which evaluates a given method with the given parameters. It also generates source code that performs the same evaluation.

```
InstructionConstraint create(final Method m,
  final Parameter... parms) {
{
  boolean isStatic = isStatic(m.getModifiers());
  return new InstructionConstraint() {
    public Method predicateMethod() {
      return m;
    }

    public boolean check(Template t,
      Sequence<Argument> args) {
      int pi;
      Object rcvr;
      Object[] objs;
      if (isStatic) {
        pi = 0;
        rcvr = null;
        objs = new Object[parms.length];
      } else {
        pi = 1;
        objs = new Object[parms.length - 1];
```

```
        rcvr = t.bindingFor(parms[0], args);
      }
      int i = 0;
      for (; i != objs.length; ++i, ++pi) {
        Parameter p = parms[pi];
        Argument a = t.bindingFor(p, args);
        if (a instanceof ImmediateArgument) {
          objs[i] = ((ImmediateArgument)a).
            boxedJavaValue()
        } else {
          objs[i] = a;
        }
      }
      // Exception handling omitted
      return (Boolean) m.invoke(rcvr, objs);
    }

    public String asJavaExpression() {
      String s = "";
      int i;
      if (isStatic) {
        s += m.getDeclaringClass().getName();
        i = 0;
      } else {
        s += parms[0].variableName();
        i = 1;
      }
      s += '.' + m.getName() + '(';
      while (i < parms.length) {
        Parameter p = parms[i];
        s += p.variableName();
        if (i != parms.length - 1) {
          s += ',';
        }
      ++i;
      }
      return s + ')';
    }

    public boolean
      referencesParameter(Parameter p) {
        return Arrays.contains(parms, p);
    }
  };
}
```

We apply this machinery for instance to the definition of the `lswi` (load string word immediate) instruction on PowerPC. This instruction loads a variable number of bytes from memory into a range of registers. Its register operand specifying the effective address of the load must not be within the target register range. This condition is tested by the following predicate method in class `ZeroOrRegister`:

```
boolean isOutsideRange(GPR target, int n) {
  int rt = target.value();
  int ra = value();
  int numRegs = (n + 3) / 4;
  int lastReg = (rt + numRegs - 1) % 32;
  boolean wrapsAround = lastReg < rt;
  if (wrapsAround) {
    return lastReg < ra && ra < rt;
  }
  return ra < rt || lastReg < ra;
}
```

We can now simply apply the previously described factory method to formulate the above predicate as a constraint in the definition of the `lswi` instruction:

```
Class c = ZeroOrRegister.class;
Method m = c.getDeclaredMethod(
  "isOutsideRange", GPR.class, Integer.TYPE);
InstructionConstraint ic = create(m, _ra0,
  _rt, _nb);
define("lswi", opcd(31), _rt, _ra0_notR0,
       _nb, xo_21_30(597), _res_31, ic);
```

5.1.4. Expressions

Sometimes an ISA includes an instruction definition whose assembler syntax includes operands that are not directly mapped to bits in the encoded instruction. Consider the following PowerPC instruction definition:

Synthetic	**Original**
clrlslwi ra,rs,b,n ($n \leq b < 32$)	rlwinm ra,rs,n,b-n,31-n

In this example, the `b` and `n` operands for `clrlslwi` are related to the fields `sh`, `mb` and `me` in the original `rlwinm` instruction shown below by the equations `sh == n`, `mb == b - n` and `me == 31 - n`.

rlwinm RA, RS, SH, MB, ME

21	RS	RA	SH	MB	ME	0
0	6	11	16	21	26	31

Expressions such as the above equations can be specified by implementing this interface:

```
interface Expression {
  long evaluate(Template t,
    Sequence<Argument> args);

  String valueString();
}
```

As with instruction constraints, factory methods are provided for creating expression objects. Here is the method used to create a subtraction expression:

```
Expression sub(Object first, Object second) {
  return new Expression() {
    public long evaluate(Template t,
                  Sequence<Argument> args) {
      return evaluateTerm(first, t, args) -
             evaluateTerm(second, t, args);
    }

    public String valueString() {
      return termValueString(first) + '-' +
             termValueString(second);
    }
  };
}
```

The `evaluateTerm` method evaluates a given term to a `long` value and the `valueString` method produces Java source code for the equivalent evaluation. Each term of an expression must be a constant, a parameter operand or another expression.

Using this expression support, the instruction definition for `clrlslwi` is as follows[7]:

[7] Here we also employ constraints (see Section 5.1.3), namely `lessEqual` and `lessThan`.

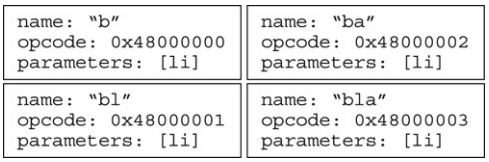


Fig. 4. PPC templates.

```
synthesize("clrlslwi", "rlwinm", sh(_n),
           mb(sub(_b, _n)), me(sub(31, _n)),
           _b, _n, lessEqual(_n, _b),
           lessThan(_b, 32));
```

5.2. Instruction templates

The assembler generator, the disassembler and the assembler tester of each ISA share a common internal representation derived from instruction descriptions called instruction *templates* and a common mechanism to create these, the *template generator*.

All templates contain the following information about an instruction:

Name the name of the assembler method that will be generated
Serial a unique numeric identifier for the template (useful for debugging)
Description the instruction description from which the template was derived
Label Parameter Index the index of the parameter specifying a control flow target address. This is only used for control flow instructions.

Further structural details of an instruction are defined in the ISA specific template subclasses. This reflects differences in the underlying ISA's such as fixed-size versus variable size instructions.

In addition to describing the structure of an information which is required to generate an assembler method, instruction templates also include attributes that are specific to the automated testing framework. They are used to mark instructions that are not accepted by the external assembler as well as addressing limitations in the internal disassemblers. As an example of the former, we define a special AMD64 instruction ("MOVZXD") which is a zero-extended move from a 32 bit register to a 64 bit register that simply doesn't exist in the assembly accepted by the GNU assembler. In terms of the latter, on PowerPC, there are synthetic instructions whose operands are not correlated one-to-one with some bits in the encoded instruction. Recovering such operands during disassembly requires a simultaneous equation solver, something that is not currently implemented.

The ISA specific details for templates are described in the following sections.

5.2.1. RISC templates

For RISC, an instruction template is created by binding constants to all the non-parameter operands in an instruction description and by building the cross product of all possible bindings for option fields.

For example, in the following description of the PowerPC unconditional branch instructions, `_lk` and `_aa` are option fields (`OptionField`) that each define a single bit in the encoded instruction. An option field object includes a set of option (`Option`) objects, one for each possible value of the field. In addition to specifying a value of an option field, an option object also specifies an instruction name suffix. In this example, the options included with the `_lk` field are [0, ""] and [1, "l"]. The options for the `_aa` field are [0, ""] and [1, "a"].

```
define("b", opcd(18), _li, _lk, _aa);
```

The templates created by enumerating over the cross product of these options are shown in Fig. 4.

5.2.2. CISC templates

The template generator for x86 is far more complex. It explores the cross product of all possible values of the following instruction properties, considering them in this order: address size attribute, operand size attribute, mod case, mod r/m group, rm case, SIB index case and SIB base case. The search for valid combinations of the above is directed by indications derived from the respective instruction description objects.

For shorter instructions, a result may be found after the first few stages. For example, an instruction that does not have a mod r/m byte, as determined by examining its opcode, may have templates with different address and operand size attributes, but enumerating different mod cases, etc., is unnecessary.

There are numerous relatively difficult to describe circumstances that limit the combinatorial scope for valid instructions. In such cases, the template generator internally throws the exception `TemplateNotNeededException` in the respective description object visitor to backtrack among the above stages. For example, instructions with addressing method code `M` occurring in their description do not require consideration of any other rm cases than the normal case when exploring mod case 3. In other words, if two general registers are used directly as operands (mod case 3), then there will be no complex addressing forms involving a SIB byte and no special (rm) cases such as memory access by an immediate address.

The number of templates that can be generated for any given instruction description ranges anywhere from 1 (for most RISC instructions) to 216 (for the *xor* AMD64 instruction).

5.3. Generating assembler source code

Each assembler generator writes two Java source code classes containing hundreds or thousands of assembly methods[8]: a *raw* assembler class and a *label* assembler class. As indicated in Fig. 2, these generated classes are accompanied by manually written classes that implement all necessary support subroutines as e.g. output buffering, label definition and binding, and instruction length adjustments and that define symbolic operand types, such as registers, special constants, etc.

For x86, we managed to use Java enums to represent all symbolic operands. For most symbolic RISC operands, though, we had to resort to a manually created pattern that mimics enums in order to capture relatively complex interrelationships such as subsetting. For example, only every second SPARC floating point register syntactically can be “double” and only every fourth can be “quadruple”.

By limiting the scope of all symbolic operand constructors to their respective class we restrict symbolic operands to predefined constants and thus we *syntactically* and therefore *statically* prevent the passing of illegal arguments to assembler methods.

To represent integral values we use Java primitive types (i.e., `int`, `short`, `char`, etc.) of the appropriate value size. If the range of legal values for an integral parameter does not exactly correspond to the range of legal values for the Java type then we add a constraint accordingly to the instruction description.

A generator needs to be run only once per assembler release. It also programmatically invokes a Java compiler[9] to reduce the generated source code to class files.

A generated raw assembler class contains one assembly method for every instruction template derived from the given ISA description. The corresponding label assembler class inherits all these methods and provides additional derivative methods with *label* parameters in lieu of primitive type (raw) parameters, wherever this is useful, based on the semantics of the respective instruction.

Each label assembler method translates its label arguments to raw operands and calls the corresponding underlying raw assembler method. In the case of x86, label assembler methods also support span-dependent instruction selection. The inherited top level assembler class (see Fig. 2) provides reusable algorithms for label resolution and for span-dependent instruction management.

The generated code that assembles a RISC instruction shifts and masks incoming parameters into position and combines the resulting bits using the logical *or* operation.

The assembly of x86, on the other hand, is mostly organized as a sequence of bytes, with conditional statements guarding the emission of certain prefixes. Some more complex bytes such as mod r/m bytes or REX prefixes also require a certain amount of bit combining.

5.4. Implementing disassemblers

The disassemblers also reuse the template generator, but they are entirely manually written. They have simple, almost but not quite brute force algorithms with usable but not great performance. At startup, a disassembler first

[8] The generated AMD64 assembler (without optional 16 bit addressing) contains 8450 methods in ≈85k lines of code, half of which are comments. The totals for the SPARC assembler are 832 methods and ≈13k lines of code.

[9] Programmatic Java compiler invocation is provided for both Sun’s javac (used in NetBeans) and for IBM’s Jikes compiler (used in Eclipse).

creates all templates for the given ISA. When applied to an instruction stream it then tries to find templates that match its binary input. The details for this task vary between the RISC and x86 disassemblers. They are described in the following two subsections. In either case, the disassembler extracts operand values and then produces a textual output including program counter addresses, offsets, synthesized labels and raw bytes.

5.4.1. RISC disassemblers

A SPARC or PowerPC disassembler only needs to read a 32 bit word to obtain a full bit image of any given instruction. To explain how it then finds a matching template, we use these notions:

opcode mask: the combined (not necessarily contiguous) bit range of all constant fields in an instruction,
opcode: a binding of bit values to a given opcode mask,
opcode mask group: a collection of templates that share the same opcode mask,
specificity: the number of bits in an opcode mask,
specificity group: a collection of opcode mask groups with the same specificity (but different opcode bit positions).

The disassembler keeps all instruction templates sorted by specificity group. To find the template with the most specific opcode mask it will iterate over all specificity groups in order of decreasing specificity. Optionally, it can do the opposite.

During the iteration, the disassembler tries the following with each opcode mask group in the given specificity group. A logical *and* of the opcode mask group's opcode mask with the corresponding bits in the instruction yields a hypothetical opcode. Next, every template in the opcode mask group that has this opcode is regarded as a candidate. For each such candidate, the disassembler tries to disassemble the instruction's encoded arguments.

If this succeeds, we reassemble an instruction from the candidate template with these arguments. This simple trick assures that we only report decodings that match all instruction constraints. Finally, if the resulting bits are the same as the ones we have originally read, we have a result.

5.4.2. x86 disassemblers

An AMD64 and IA32 disassembler must determine the instruction length on the fly, sometimes backtracking a little bit. An instruction is first scanned byte by byte, gathering potential prefixes, the first opcode, and, if present, the second opcode. The disassembler can then determine a group of templates that matches the given opcode combination, ignoring any prefixes at the moment.

The disassembler iterates over all templates in the same *opcode group*, extracts operand values and reassembles as described above in the case of RISC. In short, some effort is made to exclude certain predictably unnecessary decoding attempts, but overall, the x86 disassembler algorithm uses an even more brute force approach than the RISC disassemblers.

5.5. Fully automated self-testing

The same template generators used to create assemblers and disassemblers are reused once again for fully automated testing of these artifacts. The respective test generator creates an exhaustive set of test cases by iterating over a cross product of legal values for each parameter of an assembler method. For symbolic parameters, the legal values amount to the set of all predefined constants of the given symbol type. For integral parameters, if the number of legal values is greater than some plausible threshold (currently 32), only a selection of values representing all important boundary cases is used. Finally, the set of legal values is checked against any constraints present in the template. Support for evaluating a test case against a constraint is described in Section 5.1.3.

The above represent positive test cases, i.e., they are expected to result in valid instruction encodings. In addition, the testing framework generates negative test cases, i.e., test cases that should cause an assembler to display error behavior (e.g., return an error code or throw an exception). There will be far fewer negative test cases than positive test cases as our use of Java's static typing leaves very few opportunities for specifying illegal arguments. By far most negative test cases in the ISAs implemented so far are due to RISC integral fields whose ranges of legal values are not exactly matched by a Java primitive type (e.g., int, short, char, etc.).

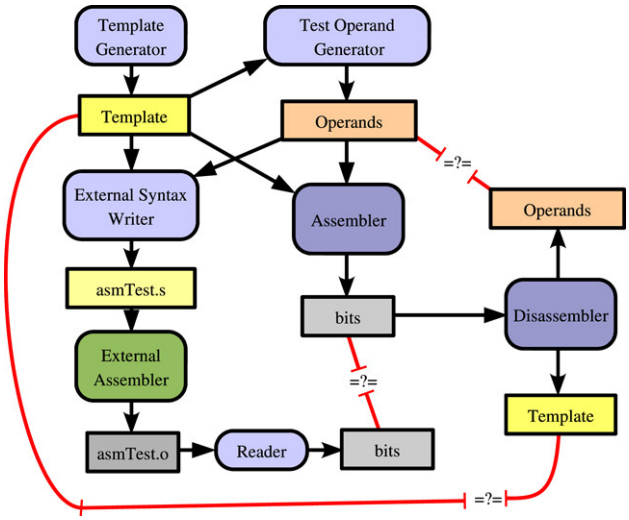


Fig. 5. Testing.

For complete testing of an assembler and its corresponding disassembler, the template generator creates all templates for an ISA and presents them one by one to the following testing procedure.[10]

First, an assembler instance is created and the assembler's Java method that corresponds to the given template is identified by means of Java reflection. Then the test generator creates all test case operand sets for the given template.

Fig. 5 illustrates the further steps taken for each operand set. The assembler method is invoked with the operand set as arguments and the identical operand set is also passed together with the template to the external syntax writer, which creates a corresponding textual assembler code line and writes it into an assembler source code file. The latter is thereupon translated by an external assembler program (e.g., gas [7]), producing an object file. By noticing certain prearranged markers in the object file, a reader utility is able to extract those bits from the object file that correspond to the output of the external syntax writer into another byte array.

Now the two byte arrays are compared. Next, one of the byte arrays is passed to the disassembler, which reuses the same set of templates from above and determines which template exactly matches this binary representation. Furthermore, the disassembler extracts operand values.

Eventually, the template and operand values determined by the disassembler are compared to the original template and operand values.

Probing all variants of even a single mnemonic may take minutes. Once a test failure has been detected, we can arrange for a very short restart/debug cycle by limiting testing to a subset of instruction templates. The instruction templates in question are easily identified by serial numbers, which are listed in the test output. When restarting the test program, we then constrain the range of templates to be retested by means of a command line option.

6. Introducing an additional ISA

This section sketches the basics for adding another ISA to the PMAS at the example of the 32 bit ARM instruction set [5], which represents a relatively straightforward instance of the RISC family.

First, we need to draft classes that declare all symbolic operands. Among those, the general purpose registers can be written as follows:

```
package com.sun.max.asm.arm;

public enum ARMRegister {
  R0, R1, R2, R3, R4, R5, R6, R7,
  R8, R9, R10, R11, R12, R13, R14,
```

[10] The description is slightly simplified: the actual program does not write a new assembler source file per instruction, but accumulates those for the same template.

```
    PC;

    public static final
      Enumerator<ARMRegister> ENUMERATOR =
        new Enumerator<ARMRegister>(
          ARMRegister.class);
}
```

Next, we create a utility class specifying bit fields that determine operand positioning and typing. Some of these fields are paired with a method that implements a constant, expecting an argument at assembler construction time.

```
package com.sun.max.asm.gen.risc.arm;

public final class ArmFields {

  public static final OptionField _cond = ...

  public static final _bits_27_26 =
    ImmediateOperandField.create(27, 26);

  public static RiscConstant
    bits_27_26(int value) {
      return _bits_27_26.constant(value);
  }

  public static final _bit_25 =
    ImmediateOperandField.create(25, 25);

  public static RiscConstant
    bit_25(int value) {
      return _bit_25.constant(value);
  }

  public static final _opcode =
    ImmediateOperandField.create(24, 21);

  public static RiscConstant
    opcode(int value) {
      return _opcode.constant(value);
  }

  public final OptionField _s =
    OptionField.create(20, 20).
      withOption("", 0).withOption("S", 1);

  public static final
    SymbolicOperandField<ARMRegister> _Rn =
      SymbolicOperandField.create(
        ARMRegister.ENUMERATOR, 19, 16);

  public static final
    SymbolicOperandField<ARMRegister> _Rd =
      SymbolicOperandField.create(
        ARMRegister.ENUMERATOR, 15, 12);

  public static final _shifter_operand =
    ImmediateOperandField.create(11, 0);
  ...
}
```

These fields can now be used in instruction descriptions, for which we write another utility class. The example below gives a complete description of the `ADC` instruction.

```
package com.sun.max.asm.gen.risc.arm;

public final class ARMInstructions
    extends RiscInstructionDescriptionCreator {
  ARMInstructions(
   ARMTemplateCreator templateCreator) {
    super(templateCreator);

    define("ADC", _cond, bits_27_26(0),
           bits_25(1), opcode(5), _s,
           _Rn, _Rd, _shifter_operand);
    ... // put many more descriptions here
  }
}
```

To put these instruction descriptions to use, we specialize a number of classes in the RISC framework classes, mostly by writing boiler plate class extensions:

1. the template creator class (see `ARMTemplateCreator` above),
2. the template class,
3. the assembler class,
4. the assembler generator class,
5. the disassembler class,
6. the disassembled instruction class,
7. the tester class,
8. the "assembly" class, which acts as a central namespace that the assembler, disassembler and tester share.

7. Related work

We have based our design on the Klein Assembler System (KAS), deriving for example the following features from it:

- specification-driven generation of assemblers, disassemblers and testers,
- instruction descriptions in the form of object arrays,
- instruction templates as the central internal representation for multiple purposes,
- most of the generator, disassembler and tester algorithms for RISC instructions,
- employment of existing external assemblers for testing.

Furthermore, we were able to copy and then simply transcode the instruction descriptions for SPARC and PowerPC. The x86 part of the PMAS has no precedent in the KAS. Whereas the general approach carried over and there is considerable reuse between the RISC and the x86 part of the framework, we had to devise different instruction descriptions, template structures, template generators and disassemblers for AMD64 and IA32.

The NJMC toolkit [17] has many similarities with our architecture. It is a specification driven framework for generating assemblers and disassemblers. Like ours, it includes mechanisms for checking the correctness of a specification internally as well as against an external assembler [8]. However, the decoders it generates are more efficient and extensible.[11] Also, it produces C source code, uses a special language for the specifications (SLED [18]) and is implemented in ML. This use of three different languages makes using, extending and modifying the toolkit harder in the context of a Java based compilation system. In contrast, the PMAS uses Java as the single language for all its purposes.

Other specification language based assembler generators are also described in [23,4] and similar publications about cross assemblers.

There are several extremely fast code generator and assembler frameworks written for and implemented in C/C++ which are specially apt for dynamic code generation.

[11] This is something we plan to remedy as described in Section 8.

GNU lightning [10] provides an abstract RISC-like instruction set interface. This makes assembly code written to this interface highly portable while trading off complete control over the native instructions that are emitted.

CCG [16] is a combination of preprocessor and runtime assembler that allows code generation to be embedded in arbitrary C programs and requires no compiler-specific extensions (such as inline asm statements or the various assembler-related extensions implemented by gcc). It gives the programmer complete control over what instructions are emitted.

One can find many more code generator frameworks (e.g., [9]) for C/C++.

8. Observations and future work

Conventionally, assemblers that run on one hardware architecture and generate code for another are categorized as *cross assemblers* and those that don't are not. Interestingly, this categorization is no longer static, i.e. determined at assembler build time, when it comes to assemblers written in a platform-independent language such as Java. Whether they cross-assemble is merely a dynamic artifact of invoking them on different platforms. On the other hand, one could argue that they run on a virtual instruction set, Java byte codes, and are therefore *always* cross-assembling.

Developing in the Java 5 language [11], we found that the features (generics, enums, static imports, varargs, annotations, etc.) introduced in this release of the language contributed substantially to our design, especially to support static type-safety of assembler applications.

The use of static typing by the Java compiler to prevent expressing illegal operand values greatly reduces the number of negative tests (e.g. ≈6000 for AMD64). Most are derived from RISC instruction operands that cannot be modelled precisely by a Java primitive type (e.g. int, short, char, etc.).

We first created a minimal framework that would only cover very few instructions but contained all parts necessary to run our instruction testing suite as described in Section 5.5. Thus we were able to catch numerous bugs, resulting from faulty instruction descriptions, missing features in our framework and various mistakes, early on. Then we expanded the number of instruction descriptions and added functionality as needed.

The effort required to develop the assembler generators shrank with each successive ISA. The CISC ISAs (IA32 and AMD64) took about 3 engineer months to complete. A large portion of this time can be attributed to the development and extension of the general framework as these were the first ISAs we implemented. The SPARC and PowerPC ISA ports each took about 1 month. Once again, about half of this time can be attributed to adding missing features to the framework.

We discovered a handful of errors in most ISA reference manuals and we even found a few bugs in every external assembler. This could be determined by three-way comparisons between our assemblers, the external assemblers and reference manuals.

Even though there is no absolute certainty regarding the validity of our instruction encodings and decodings, we stipulate that the number of bugs that our system would contribute to a complex compiler system should be minimal.

For now we have been focusing on correctness, functionality and completeness, and we have not yet had enough time to analyze and tune the performance of any part of the PMAS.

Not having emphasized performance in the design of the disassemblers, generators and testers, we find it sufficient that the disassemblers produce pages of listings quicker than humans can read even a single line and that each generator runs maximally for tens of seconds.

With regard to assembler performance, we paid attention to avoiding impediments that would be difficult to remedy later. Our first impressions suggest that even without tuning our assemblers are fast enough for use in static compilers and in optimizing dynamic compilers with intermediate representations. Performance is clearly not yet adequate when emitting code in a single pass JIT, though. To remove the most obvious performance bottleneck, we plan to replace currently deeply stacked output routines with more efficient operations, e.g., from `java.nio`.

As future work, we envision generating source code for disassemblers and providing a more elegant programmatic disassembler interface to identify and manipulate disassembled instructions abstractly, offering visitor and callback patterns rather than confronting the user with templates.

The full source code of the PMAS is available under a BSD license at [2]. We recommend direct CVS download into an IDE. Project files for both NetBeans and Eclipse are included. Complementary prepackaged assembler jar files are planned, but not yet available at the time of this writing.

Acknowledgements

Adam Spitz created a draft version for SPARC and PowerPC of the presented system by porting the Klein Assembler System from Self to the Java language.

We would like to thank Mario Wolczko and Greg Wright for their helpful comments, and their insightful perusal of our first draft. We'd also like to thank Cristina Cifuentes for sharing her insights on the NJMC toolkit with us.

References

[1] O. Agesen, L. Bak, C. Chambers, B.-W. Chang, U. Hölzle, J. Maloney, R.B. Smith, D. Ungar, M. Wolczko, The SELF 4.0 Programmer's Reference Manual, Sun Microsystems, 1995.
[2] Bernd Mathiske, Doug Simon, Project Maxwell Assembler System [online]. Available from: http://maxwellassembler.dev.java.net/, 2006.
[3] M. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. Serrano, V. Sreedhar, H. Srinivasan, The Jalapeno dynamic optimizing compiler for Java, in: ACM Java Grande Conference, June 1999.
[4] P.P.K. Chiu, S.T.K. Fu, A generative approach to universal cross assembler design, SIGPLAN Notices 25 (1) (1990) 43–51.
[5] David Seal (Ed.), ARM Architecture Reference Manual, second ed., Addison-Wesley, 2001.
[6] Eclipse.org. C/C++ Development Tools [online]. Available from: http://eclipse.org/cdt, 2003.
[7] D. Elsner, J. Fenlason, et al., Using the gnu AS assembler [online]. Available from: http://www.gnu.org/software/binutils/manual/gas-2.9.1/as.html, 1999.
[8] M.F. Fernandez, N. Ramsey, Automatic checking of instruction specifications, in: ICSE, 1997, pp. 326–336.
[9] C.W. Fraser, D.R. Hanson, T.A. Proebsting, Engineering a simple, efficient code-generator generator, ACM Letters on Programming Languages and Systems 1 (3) (1992) 213–226.
[10] Free Software Foundation, Inc., GNU Lightning [online]. Available from: http://www.gnu.org/software/lightning, 1998.
[11] J. Gosling, B. Joy, G. Steele, G. Bracha, The Java Language Specification, third edn., in: The Java Series, Addison-Wesley, Boston, Massachusetts, 2005.
[12] IBM Corporation, The PowerPC Architecture: A Specification for a New Family of RISC Processors, second ed., Morgan Kaufmann Publishers, 1994.
[13] Intel. IA-32 Intel Architecture Software Developer's Manual, Volume 2A: Instruction Set Reference, A-M. Available from: ftp://download.intel.com/design/Pentium4/manuals/25366619.pdf, 2006.
[14] P. Johnson, M. Urman, The Yasm Core Library: Libyasm [online]. Available from: http://www.tortall.net/projects/yasm/wiki/Libyasm, 2003.
[15] ovm.org. OVM [online]. Available from: http://www.cs.purdue.edu/homes/jv/soft/ovm/, 2005.
[16] I. Piumarta, The virtual processor: Fast, architecture-neutral dynamic code generation, in: Virtual Machine Research and Technology Symposium, USENIX, 2004, pp. 97–110.
[17] N. Ramsey, M.F. Fernandez, The New Jersey machine-code toolkit, in: USENIX Winter, 1995. pp. 289–302.
[18] N. Ramsey, M.F. Fernandez, Specifying representations of machine instructions, ACM Transactions on Programming Languages and Systems 19 (3) (1997) 492–524.
[19] Sun Microsystems. The Java HotSpot Virtual Machine, Technical White Paper, 2001.
[20] D. Ungar, A. Spitz, A. Ausch, Constructing a metacircular virtual machine in an exploratory programming environment, in: Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2005, ACM, 2005, pp. 11–20.
[21] D.L. Weaver, T. Germond, The SPARC Architecture Manual — Version 9, Prentice-Hall PTR, 1994.
[22] J. Whaley, Joeq: A virtual machine and compiler infrastructure, Science of Computer Programming 57 (3) (2005) 339–356.
[23] J.D. Wick, Automatic Generation of Assemblers (Outstanding Dissertations in the Computer Sciences), Garland Publishing, New York, 1975.