Specifying the Semantics of Machine Instructions*

Cristina Cifuentes Shane Sendall
Department of Computer Science and Electrical Engineering
University of Queensland
Brisbane Qld 4072, Australia
{cristina,shanes}@csee.uq.edu.au

Technical Report 422
December 1997

Abstract

Computer architecture manuals describe the instruction set of the machine and the semantics of those instructions by a combination of natural language and ISP (Instruction Set Processor) descriptions. The syntax of the instructions in assembly is well defined in the form of tables in the manual. However, the semantics is not so well specified and descriptions vary widely from one manual to another.

When developing a retargetable binary translator, as much as possible needs to be specified in order to automatically generate code from specifications, hence separating machine-independent issues from the manual coding stage. The specification of the semantics of machine instructions is one such task, with the aim of generating suitable code for an intermediate representation that is to be used during the analysis stage.

We describe the design process used to develop a semantic specification language, SSL, to integrate into a retargetable binary translation framework. The techniques described herein are suitable not just to binary translators but also to machine-code manipulation tools such as optimizing compilers, binary profilers, instrumentors, and binary debuggers.

Keywords: binary translation, retargetable, formal specification, CISC, RISC.

*This work is sponsored by the Australian Research Council under grant No. A49702762 and The University of Queensland.

1 Introduction

Binary translation is a form of re-engineering of machine code, which follows the taxonomy of Chikovsky and Cross [1]. Binary translation allows the running of code compiled for source platform M_s on destination platform M_d. Unlike an interpreter or emulator, a binary translator makes it possible to approach the speed of native code on machine M_d. Translated code may run more slowly than native code because low-level properties of machine M_s must often be modeled on machine M_d. For example, the Digital Freeport Express translator [3] simulates the byte order of SPARC, and the FX!32 translator [14, 5] simulates the calling sequence of the source x86 machine, even though neither of these is native to the target Alpha machine.

From a simplistic point of view, binary translation requires the translation of sequences of machine instructions from one machine to another. Although it is not very difficult to translate some sequences of machine instructions from one machine to another, other considerations make the task very difficult in practice. For example, binary code often mixes data and instructions in the same address space in a way that cannot be distinguished given the same representation for data and code in Von Neumann machines. This problem is exacerbated with indirect or indexed jumps, where the target value of the jump is known at runtime, but sometimes hard to determine statically, at translation time. Further, some of the older operating systems did not provide systems programmers with an ABI (application binary interface) to low-level system calls, hence allowing application writers to directly access the hardware. All of these problems and more are common to binary-code manipulation tools such as disassemblers, profilers, instrumentors and decompilers. Nevertheless, for binary translation purposes, this does not mean that the problem cannot be solved at all. In fact, given that the translated binary file will need to be executed, any information that could not be decoded statically will be available dynamically, hence allowing for runtime translation or interpretation of the binary code, at the expense of user time.

Retargetable Binary Translation

Current binary translators are driven by commercial interests and are fairly machine-dependent, normally limited to one or two platforms. In contrast, we are developing a retargetable binary translation framework which will aid in the specification and automation of the machine-dependent parts of the translation, therefore liberating programmers from time-consuming and error-prone jobs such as decoding and encoding machine instructions and their semantics, and allowing them to work on more interesting problems such as machine-independent analyses. Retargetability allows the construction of machine-independent tools by formal specifications of machine-dependent aspects of a problem and allowing the portability of the tools by reuse of specifications for a different machine.

The specification of the *syntax* of machine instructions and its associated assembly mnemonics has been made possible through the SLED (Specification Language for Encoding and Decoding) language [10] implemented by the New Jersey Machine-Code (NJMC) toolkit [9]. The toolkit allows users to write a specification for the syntax of machine instructions for a particular machine, and decode or encode to that syntax based on extra language statements available in the toolkit. In this way, a decoder of machine instructions can be fully specified and the user need only write the specification for a new machine instruction set (or reuse an existing one) and determine the appropriate stream of bytes to decode.

For binary translation analysis purposes, the *semantics* of the machine instructions decoded from a binary executable file is needed. Given that most machines perform similar operations such as loads, stores and branches, but have a slightly different semantic meaning attached to their machine instructions, the semantic meaning of such instructions should be able to be specified.

This paper describes the development process in the specification of semantics of machine instructions for a CISC and RISC machine. The paper is structured in the following way: Section 2 described previous work in the area of specifying semantics of machine instructions, Section 3 describes the formal specification of semantics via the Object-Z [4] language, Section 4 describes the informal refinement of the Object-Z specifications into a Semantic Specification Language called SSL; this language is described in Section 5. Section 6 provides examples on how the SPARC and 80286 architectures were modelled and Section 7 provides a discussion on the use of such specifications. The paper concludes with conclusions.

2 Previous Work

There has been little work done in the area of specifying the semantics of machine instructions as few tools deal with them in a retargetable way. Most tools embed the semantics of the particular instruction set they are dealing with into the source code; retargeting of such tools to other platforms is hard and time consuming due to the inherent dependency on the machine instruction set it was written for.

Computer architecture manuals describe the syntax and semantics of machine instructions by a combination of natural language and ISP (Instruction Set Processor notation) descriptions, for example see Intel's Pentium manual [6] and SPARC's V8 manual [12]. The syntax of the instructions in assembly and its associated binary or machine code is well defined in the manual in the form of tables. ISP is a high-level notation that resembles a structured programming language, with constructs for conditionals and iteration. The notation itself does not use a standard language, hence different manuals specify pseudo-code for the semantics in a different way. This makes the notation an ambiguous notation for specification purposes. Further, low-level machine dependencies such as memory alignment are also specified in ISP.

EEL, an executable editing library [7] which aids users instrument executable programs, developed a simple semantic specification language based on the SLED language of the New Jersey Machine Code (NJMC) toolkit [9, 10]. EEL extends NJMC's specification constructs with a semantic construct, `sem`, which provides a simple attribute description of the semantic of an instruction (or a set of instructions). However, this language only models basic semantics of an instruction and does not provide a way of specifying the full semantics of the instruction, such as modifying condition codes, and machine-dependencies such as register windows on SPARC. The language has been used for writing profiling tools.

Our approach differs from previous work by allowing a user to specify the semantics of a set of machine instructions and using that specification for generating an intermediate language which is supported in a retargetable binary translation environment.

3 Specifying Semantics of Machine Instructions in Object-Z

The Object-Z specification language is an object-oriented extension to the Z formal specification language [13] and was developed specifically to facilitate specification in an object-oriented style. A Z specification typically models a system by specifying a number of state and operation schemas. A state schema groups together variables defining a part of the state of the system, and defines invariant relationships held between variables. An operation schema, based on the state schema, defines the way its functionality changes the system's state. Object-Z adds a class structure that encapsulates a single state schema with related operations. Each class can be examined and understood in isolation, and complex classes can be specified in terms of simpler classes through object-oriented structuring techniques. A result of using Object-Z is that system functionality and object-oriented design can be captured within one notation. A full description of the language can be found in Duke and Rose [4].

3.1 80286 Environment

The 80286 machine is a 16-bit segmented memory CISC architecture. It contains fifteen 16-bit registers that can be grouped into the following categories: general registers, segment registers, and status and control registers. Flag information is contained in a register which stores 11 flags. An 80286 instruction consists of two distinct pieces of information: an opcode specifying the operation, and zero, one, or two operands identifying the object(s) to be manipulated by the instruction. Instructions can reference operands by means of eight addressing modes: two for register operands, and six for memory operands. Immediate operands are supplied as part of the instruction itself. Objects in memory are accessed by means of a 32-bit address pointer.

The Object-Z environment declaration of the 80286 architecture is shown in a simplified form in Figure 1; it contains instances of all registers, flags and memory required by the 80286 architecture. Registers and flags are modelled independently, and memory is modelled as a sequence of *Memories* objects (an object which ensures the consistency of values stored in a byte memory location). 16-bit sizes are enforced on all word registers. The registers, flags, and memory are initialised to zero. *Repeat* and *Skip* have also been included in the environment, they act as control variables used for the repeat instructions.

3.2 SPARC Environment

SPARC is a 32-bit linear address space RISC architecture. All instructions are 32-bits wide and are aligned on 32-bit boundaries in memory. There are only three general instruction formats and they feature uniform placement of opcode and register address fields. Only load and store instructions access memory. It supports three addressing modes, two for memory addressing and one for register or immediate operands. Most instructions operate on two register operands, and place the result in a third register. At any one

$286\mathit{Env}$

$$\begin{array}{l}
AX, BX, CX, SP, CS, PC, ... : \mathit{Registers} \\
OF, ZF, CF, ... : \mathit{Flags} \\
\mathit{memory} : \mathrm{seq}\, \mathit{Memories} \\
\mathit{Repeat}, \mathit{Skip} : \mathbb{B} \\
\Delta \\
\mathit{FLAGS} : \mathbb{N} \\
\hline
AX.\mathit{size} = 16\mathit{BITS} \wedge BX.\mathit{size} = 16\mathit{BITS} \\
CX.\mathit{size} = 16\mathit{BITS} \wedge SP.\mathit{size} = 16\mathit{BITS} \\
CS.\mathit{size} = 16\mathit{BITS} \wedge PC.\mathit{size} = 16\mathit{BITS} \\
... \\
...
\end{array}$$

INIT

$$\begin{array}{l}
AX.\mathit{init} \wedge BX.\mathit{init} \wedge CX.\mathit{init} \\
SP.\mathit{init} \wedge CS.\mathit{init} \wedge PC.\mathit{init} \\
CF.\mathit{init} \wedge OF.\mathit{init} \wedge ZF.\mathit{init} \\
... \\
... \\
\forall\, m : \mathrm{ran}\, \mathit{memory} \bullet m.\mathit{init} \\
\mathit{Repeat} = \mathrm{false} \\
\mathit{Skip} = \mathrm{false}
\end{array}$$

..
..

Figure 1: The Object-Z specification of the 80286 architecture (simplified).

instance, a program sees 8 global integer registers plus a 24-register window into a larger register store.

The specification of the SPARC environment is shown in Figure 2. It contains instances of all registers, flags, and memory required by the SPARC architecture. Registers and flags are modelled independently, and memory is modelled as a sequence of *Memories* objects. Register windows are modelled simplistically with a current context window pointing to a window of the total group of registers. The current window is attended to by the *cwp* variable. The level of abstraction available to the environment through the Object-Z language allows the ease of potentially complex semantics implied by register windows.

3.3 80286 and SPARC Instruction Specifications

The level of abstraction possible with SPARC's loads and stores proved to be quite high compared to the architecture manual's ISP definition. The store (ST) instruction stores a word into a memory location. Figure 3 shows its Object-Z specification: *ST* handles addressing modes by allowing a choice

$$\begin{array}{l}
\textit{SparcEnv} \\
\hline
r, asrw : \operatorname{seq} Registers \\
\%y, \%psr, \%tbr, \%fsr, \%wim, \%PC, \%nPC : Registers \\
N, Z, V, C : Flags \\
m : \operatorname{seq} Memories \\
cwp : \{0..NWINDOWS - 1\} \\
\Delta \\
\%g0, ..\%g7, \%o0, ..\%o7, \%l0, ..\%l7, \%i0, ..\%i7, \\
\%fp, \%sp : Registers \\
ins, outs, locals, globals : \mathbb{P}\, Registers \\
.. \\
\hline
\%g0 = \%r(0), ..\%i7 = \%r(31) \\
\{\%g0, \%g1, \%g2, \%g3, \%g4, \%g5, \%g6, \%g7\} = globals \\
\{\%o0, \%o1, \%o2, \%o3, \%o4, \%o5, \%o6, \%o7\} = outs \\
\{\%l0, \%l1, \%l2, \%l3, \%l4, \%l5, \%l6, \%l7\} = locals \\
\{\%i0, \%i1, \%i2, \%i3, \%i4, \%i5, \%i6, \%i7\} = ins \\
\forall x : \operatorname{dom} r \bullet r(x) = w(x + (16 * cwp.val) \\
\qquad\qquad \operatorname{mod}(NWINDOWS * 16)) \\
.. \\
\hline
\textit{INIT} \\
\hline
\forall i : \operatorname{dom} w \bullet w(i).init \\
.. \\
\hline
\end{array}$$

Figure 2: The SPARC environment consists of all the registers, flags, and memory instances. Also includes a definition of register windows.

dependent on the i field of the instruction. The calculated address is then used as the offset into memory for the storage of the third operand (rd). Definitions of loads and stores are reused where appropriate—in the tradition of object-orientation.

$$\begin{array}{l}
ST \mathrel{\widehat=} [instr? : Instruction;\ addr : Immediate \mid \\
\quad instr? \in \{Format4, Format5\}] \bullet \\
\qquad ([instr.i.val = 0] \bullet \\
\qquad\qquad addr.UpdateVal[(r(instr.rs1.val).val+ \\
\qquad\qquad\qquad r(instr.rs2.val).val)/newval?] \\
\qquad \Box \\
\qquad [instr.i.val = 1] \bullet \\
\qquad\qquad addr.UpdateVal[(r(instr.rs1.val).val+ \\
\qquad\qquad\qquad instr.simm13.val)/newval?]) \\
\qquad \fatsemi \\
\qquad m(addr.val).UpdateVal[r(instr.rd.val).val/newval?]
\end{array}$$

Figure 3: Object-Z specification of ST - stores a word into memory.

In contrast, in the 80286 architecture, only a basic notion of addressing mode has been applied. This means that fields and formats have been omitted from the specification, choosing instead to model the instructions from a black-box point of view, hence allowing only input and output variables. In the case where there are many addressing modes, the input and output variables are not strictly constrained to a certain type or size but rather the precondition describes what the variables cannot be. Therefore specific addressing modes are not supplied. An example of this black-box abstraction can be seen in Figure 4 in the specification of the add instruction, ADD, where the input and output variables $lsrc?$, $rsrc?$, and $dest!$ are not constrained to particular sizes or types but rather the precondition (lines 2 and 3) informs only of the invalid cases. The inclusion of every case in the specification would greatly increase the size of the specification, and also reduce its readability. Shown against their assembly derivatives (see Figure 5) of six different addressing modes, it is possible to see why the inclusion of specific cases for each option could produce a large specification document.

$$
\begin{array}{l}
ADD \mathrel{\widehat{=}} [dest!, lsrc?, rsrc?, temp : Operand] \bullet \\
\qquad\qquad [lsrc? \notin Immediate \wedge lsrc? \in Memories \\
\qquad\qquad\qquad \Rightarrow rsrc? \notin Memories \wedge \\
\qquad\qquad\qquad lsrc?.size = rsrc?.size \wedge \\
\qquad\qquad\qquad dest! = lsrc? \wedge \\
\qquad\qquad\qquad temp.size = TEMP] \bullet \\
\qquad\qquad\qquad\qquad temp.v.UpdateVal[(lsrc?.val + \\
\qquad\qquad\qquad\qquad\qquad rsrc?.val)/newval?] \ \bigwedge \\
\qquad\qquad\qquad\qquad dest!.v.UpdateVal[temp.val/newval?]
\end{array}
$$

Figure 4: 80286 ADD instruction – places the addition of its two operands back into the first operand.

`ADD CX, DX`	REGISTER, REGISTER
`ADD DI, ALPHA`	REGISTER, MEMORY
`ADD BETA, CL`	MEMORY, REGISTER
`ADD CL, 2`	REGISTER, IMMEDIATE
`ADD ALPHA, 2`	MEMORY, IMMEDIATE
`ADD AX, 20`	ACCUMULATOR, IMMEDIATE

Figure 5: 80286 ADD assembly instruction combinations – 6 addressing modes in total.

SPARC uses delay slots in control transfer instructions as a form of pipelining. Therefore, any instruction that transfers control has the option of executing a delayed instruction (the next physical instruction in the stream). The delayed instruction is executed while the branch is taking place. Delayed transfers of control were modelled with the concept of a *Next Program Counter* (*nPC*). The *nPC* is an environment variable like the program counter (*PC*), which stores the next program position rather than the current one (stored by the *PC*). An example of a control transfer instruction, branch if not equal (*BNE*), is shown in Figure 6. The *BNE* instruction takes the branch if the Z (zero) flag is not set and executes the delay instruction unconditionally. Since the program counter (*PC*) is updated before every instruction, the *PC* is pointing at the delay instruction, therefore the next program counter (*nPC*) is set to the branch location (the offset *disp*22). When the Z (zero) flag is set, the delay instruction is optional and dependent on the instruction's annul field (a). To annul the execution of the delay instruction, *PC* is set to the instruction after the delay instruction (the offset *INSTR_SIZE*).

$$\begin{array}{l}
\mathit{BNE} \mathrel{\widehat=} [\mathit{instr?} : \mathit{Instruction} \mid \mathit{instr?} \in \{\mathit{Format3}\}] \bullet \\
\qquad\qquad [Z.\mathit{val} = 0] \bullet \mathit{nPC}.\mathit{UpdateVal}[(\mathit{PC}.\mathit{val} + \\
\qquad\qquad\qquad\qquad \mathit{join}(\mathit{instr?}.\mathit{disp22}.\mathit{val}, 0, 0))/\mathit{newval?}] \\
\qquad\qquad [] \\
\qquad\qquad [Z.\mathit{val} = 1] \bullet \\
\qquad\qquad\qquad [\mathit{instr?}.a.\mathit{val} = 1] \bullet \mathit{PC}.\mathit{UpdateVal}[\\
\qquad\qquad\qquad\qquad (\mathit{PC}.\mathit{val} + \mathit{INSTR\_SIZE})/\mathit{newval?}]
\end{array}$$

Figure 6: Object-Z specification of *BNE* – branches on Z flag equal to 0.

4 From Object-Z to SSL – A Semantic Specification Language

The transformation of the Object-Z specifications into a simple and parsable semantic specification language (SSL) was done through an iterative informal refinement process. Some difficulties were experienced in the refinement process when trying to bridge the high-level specification to a concrete form. These problems were mainly experienced due to the black-box approach used to specify addressing modes—these were not explicitly modelled in Object-Z, instead, an operand name was used. This meant that during the refinement to SSL, some addressing modes needed to be checked by refering to the architecture manual.

The SSL language was developed with integration into the SLED language [10, 9] in mind, via a library or template. Hence, an application writer is able to do syntax decoding of machine instructions using SLED and NJMC, and perform semantic analysis using SSL and SRD (a Semantic

Representation Decoder tool that implements SSL).

The process was iterative as several requirements were sought out of SSL. These requirements were constantly checked in the new versions of the language:

- provide a simple and compact notation,
- model the semantics of machine instructions separately or per groups of instructions,
- model basic transfers of information via registers and memory locations,
- model complex and basic instructions without introduction of recursion or function calls,
- strictly model sizes of operands, registers and memory accesses,
- provide a universal model for flags and their interactions via named registers and macro "functions", and
- model broad environment structure and semantics to handle many architectures and their idiosyncrasies; in other words expressability.

Out of these requirements, the architecture environment is not yet fully supported, and is not discussed in this paper.

5 Specification Concepts of SSL

We describe the syntax and semantics of SSL informally, through natural language and examples. A complete description of the syntax in EBNF form and a more detailed semantic specification can be found in [11, 2].

SSL allows for the description of the semantics of a *list of instructions* by means of *statements* or register transfers. Most statements are *assignment* statements, but there is also support for *conditional* and *flag* statements. The register transfers for a group of instructions can be grouped via a *table*. Individual assignment register transfers allow for a variety of *expressions* (arithmetic, bitwise, logical and ternary). The base elements of an expression are *values*, and the base elements of an instruction are *variables*. We explain each of these in detail in the next sections.

5.1 Variables and Values

A *variable* can be a register, memory, or parameter to an instruction operand. A *value* is the contents of a variable (denoted with the prefix prime symbol (')) or a numerical constant. A value can be signed extended by means of the `!` symbol.

For example, `r[5]` is register 5, and `'m[100000]!` is the sign-extended value of the memory location 100000.

5.2 Constants

Constants are names assigned to numerical values that do not change. Constants are commonly used to describe fixed values of a machine, for example, `WORD := 32`.

5.3 Expressions

Three groups of expressions are supported: unary, binary and ternary, each with an expressions as a member. Expressions are thought of as trees, with the leaves being the values of the expression and the inner nodes being the operators of the expressions.

Unary expressions include the negation (`NOT`) of an expression and the sign extension (`!`) of an expression.

Binary expressions include arithmetic, bitwise and logical expressions, as well as bitwise expressions (`@`). The first three types of binary expressions are commonly found in most programming languages. The latter expression, bitwise-extraction, is needed to extract bits of a field, and hence the top and bottom bits need to be specified. This expression is derived from the SLED language. Examples of each of these types of expressions follow:

```
'r[1] + 'r[5]      // arithmetic
'r[1] | 'm[1000]   // bitwise
'r[5] or 'r[1]     // conditional
'r[5] @ [0:19]     // bitwise-extraction
```

The ternary expression `?:` consist of a logical expression, a true-branch expression, and a false-branch expression. The semantics is as per the C language: if the logical expression evaluates to true, the true-branch expression is evaluated, otherwise the false-branch expression. For example, returning a boolean based on the contents of register 1 can be specified as:
`'r[1] = 0 ?  0 :  1.`

Expressions can be casted to another size (in number of bits required). Casting can upgrade the size of the value of an expression or downgrade it. Casting is denoted by postfixing the size in brackets. For example, `'r[rs1]`{64} casts the value of register `rs1` to 64 bits.

Finally, there are two types of expressions that deal with tables. These expressions will be explained once the concept of *table* is introduced in Section 5.5.1.

5.4 Statements

Statements describe transfers of information to/from registers. All transfers have to be specified; there are no side-effects on transfers other than those described by a statement. Most transfers will be assignments, however, there is also need for conditional ($\Rightarrow$) statements and support for condition codes as we do not want to fully specify these transfers, but merely know if a change in a condition code could happen or not.

An *assignment statement* consists of the size of the assignment (in bits), the variable of the target of the assignment, and an expression describing the value of the assignment. For example,

```
*32* r[rd] := 'imm22 << 10
```

assigns 32 bits of the contents of `imm22` left-shifted 10 bits to register `rd`.

A *conditional statement* consists of a membership logical expression, followed by a list of statements. If the logical expression is true, the list of statements is valid. Membership is denoted by the operator `|=`. A membership logical expression tests if a value is a member of a set of numbers (or ranges of numbers). For example, `'r[rd] |= {2,3}` tests if the value of register `rd` is either 2 or 3.

The *empty statement* is denoted by `-`. This statement is useful when describing the semantics of the `NOP` instruction.

Support for Condition Codes

Condition codes are treated as named registers of size 1 bit. Although only 0 or 1 can be assigned to a condition code, assignment statements to condition codes can be quite complex if fully described. For example, the SPARC V8 manual describes the overflow of an add instruction which sets the condition codes as:

```
V <- (r[rs1]<31> and operand2<31> and
      (not result<31>)) or ((not r[rs1]<31>)
      and not operand2<31> and result<31>)
```

Although this expression could be specified in SSL, we do not want to know how the condition code was set other than it *may* be changed—this removes overhead during translation time as an overflow condition code will have a similar meaning in all architectures.

Since we are interested in knowing if the value of a condition code may have been changed, we provide the following two macros:

- updateflags: specifies the named condition codes that may be changed by the instruction.

For example, the 80286 multiply instruction modifies all 6 condition codes; this is specified as:
`defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)`.

- undefineflags: this macro specifies that the value of all current named condition codes is undefined or unknown. Few machine instructions produce this effect; for example the 80286 divide instruction:
`undefineflags()`.

5.5 Tables

Tables are used for grouping names of instructions alone or pairs of instructions and operators or expressions. These are useful when describing the semantics of a group of instructions that behave in a similar way; the group can be declared in a table and given a name (the name of the table), which is then used in the specification, as described in the next section (Section 5.6).

Tables of instructions are handy when grouping instructions that come from the same family, such as the store instructions. On SPARC, there are store double-word, word, half-word, and byte instructions, both in alternate or non-alternate storage, for a total of 8 instructions. The semantics of the store family of instructions is same, the only difference is in the size of the operand. Hence these instructions could be grouped in the `STORE` table:

```
[STORE] := {STD, STDA, ST, STA, STH, STHA, STB, STBA}
```

Instructions that perform an arithmetic or bitwise operation can be grouped in a table which pairs the instruction name with its operator. In the following example, we have grouped the add, subtract, add and set condition codes, and subtract and set condition codes, in the 2-dimensional `[ARITH,OP3]` table. The name `ARITH` is used to identify the instruction names, and the `OP3` name is used to identify the operator symbol. The table is viewed as a 2-dimensional array for usage purposes:

```
[ARITH, OP3] := { (ADD_, "+"), (SUB_, "-"),
                  (ADDCC_, "+"), (SUBCC_, "-") }
```

Finally, the third type of table pairs instructions and expressions. This is useful when grouping conditional instructions for example, as the condition of the instruction is dependent on a condition code or a set of condition codes; i.e. they are based on an expression of condition codes. In the following example, the table `[JUMPS,COND]` is created. The `JUMPS` name indexes instruction names, and the `COND` name indexes expressions associated with the instructions:

```
[JUMPS,COND] :=
      { (BA_, 1), (BN_,0), (BNE_, ~'%Z), (BE_, '%Z),
        (BG_, ~('%Z |('%N ^ '%V))),
        (BLE_, '%Z |('%N ^ '%V)), (BGE_, ~('%N ^ '%V)),
        (BL_, '%N ^ '%V), (BGU_, ~('%C | '%Z)),
        (BLEU_, '%C | '%Z), (BCC_, ~'%C),
        (BCS_, '%C), (BPOS_, ~'%N), (BNEG_, '%N),
        (BVC_, ~'%V), (BVS_, '%V) }
```

5.5.1 Tables and Expressions

In Section 5.3 we mentioned that there were two types of expressions that dealt with tables: the table expression and the table operand. The former allows users to index the second element of tables that pair instructions and expressions, by indexing on the expression (i.e. the COND field in the previous example). The latter allows users to index the second element of tables that pair instructions and operators, by indexing on their operator (i.e. the OP3 field in the second to last example above).

The expressions that relate to tables can be used as part of assignment statements which facilitate the description of the semantics of an instruction. For example, our earlier addition expression 'r[1] + 'r[5] would be more generally specified if taken in context of the arguments passed to an addition instruction: a register rs1, a register or an immediate reg_or_imm, and the destination register rd, and specified as an assignment statement:

```
*32* r[rd] := 'r[rs1] + 'reg_or_imm
```

When being part of the [ARITH,OP3] table above, the whole set of instructions can be specified as follows:

```
*32* r[rd] := 'r[rs1] OP3[idx] 'reg_or_imm
```

where idx is an indexed variable into the table (based on the instruction parsed), and it would be 1 if the ADD instruction were parsed.

In a similar way, we can index in the [JUMPS,COND] table to determine the condition of a branch instruction. In this case, if the condition is true, the named register %nPC is set to a displacement from the current PC, otherwise it is set to the next physical instruction. Note that this is a simplification of the complete semantics for illustration purposes only:

```
*32* %nPC := ((COND[idx] = 1) ?  '%PC + (4 * disp22) :  '%PC + 4)
```

5.6 SSL Instructions

An SSL instruction is the way we describe the semantics for one particular machine/assembly instruction. An SSL instruction takes the name of the assembly instruction or a table name as its left-hand-side (LHS) and a list of SSL statements (as per Section 5.4) on its right-hand-side (RHS). The RHS and LHS are separated by indentation for readability purposes.

The assembly ORcc instruction takes three arguments on SPARC; a register and another register or an immediate value, and the destination register. The input arguments are bitwise-or'd and the result is placed on the destination register. The 4 condition codes are also updated. This instruction can be specified as follows:

```
ORCC rs1,reg_or_imm,rd       *32* r[rd] := 'r[rs1] or 'reg_or_imm
                             defineflags (%N, %Z, %V, %C)
```

The following example illustrates how to specify the semantics for a group of arithmetic and bitwise instructions that have been grouped in the INSTR_TABLE table. The LHS specifies the name of the instruction (i.e. one of the ones in the table) and the number and names of the parameters. The RHS specifies the semantic operation to be performed based on the index of the instruction in the table.

```
[INSTR_TABLE, OP1] := { (ADD_,"+"), (AND_,"&"),
                        (OR_,"|"), (SUB_,"-"), (XOR_,"^") }
INSTR_TABLE[idx] param1    *8* r[1] := 'r[1] OP1[idx] 'param1
```

6 Modelling of Semantics of CISC and RISC Machine Instructions using SSL

SSL has been used to model the semantics of machine instructions for a CISC (Intel's 80286) and a RISC (SPARC) machine. The 80286 was chosen instead of the Pentium due to its CISC characteristics and the fact that it is a subset of Pentium, with only 250 instructions as opposed to 500 in the Pentium. In this section we show extracts of the complete specifications for these machines; the complete specs are available in [11, 2] or from the web site: http://www.it.uq.edu.au/csm/bintrans.html.

6.1 The Underlying Fetch-Execute Cycle

Underlying any semantic specification is the fetch-execute cycle that the processor follows when executing machine instructions. The standard cycle is the following:

1. Fetch the instruction from memory at the location pointed to by the PC register,
2. Increment PC by the size of the instruction fetched,
3. Decode the fetched instruction, and

4. Execute the instruction.

The cycle is repeated until the running process terminates.

Architectures such as SPARC have slightly modified the fetch-execute cycle by introducing another register to keep track of targets of delayed instructions. In their case, the PC points to the next instruction to be executed, and the nPC points to the next PC value; i.e. the instruction after the next one. Once an instruction is fetched, the PC takes the value of the nPC, and the nPC is incremented by the size of the instruction fetched.

It is implicit in our specifications that the relevant registers (PC or PC and nPC) are updated after the instruction is fetched (i.e. decoded by the NJMC toolkit). The "execution" or semantic representation of transfers of control instructions will update the PC (and nPC) registers as required, hence providing the equivalent information to that of the fetch-execute cycle. However, as will be noted in Section 7, this is not always a good specification decision as it will make analysis (for binary translation) harder. One should be able to use the traditional fetch-execute cycle and remove any other architectural issues.

6.2 Examples of 80286 and SPARC Specifications

The arithmetic and logical instruction table is shown in Figure 7 as an example of an 80286 group of instructions specified in SSL. `[ARITHLOG,OP1]` is a table containing pairs of arithmetic and logical instruction, and their operators. Due to the large number of addressing modes in x86, each instruction in this group takes 9 different forms depending on the arguments to the instruction, and these forms are differenciated by decorating the name of the table in each case. Each instruction consists of an assignment, which is either of 8 or 16 bits in size, and assigns the result of the expression onto the destination register (which in several instances is an implicit register as described in the architecture manual). The second statement for each instruction is a `defineflags` macro statement which describes transfers of information to condition codes.

The use of tables for instructions, operators, and expressions greatly decrease the size of the specification, at a small expense on readability of the specification. The use of flag macros enhances the specification by increasing its readability, and abstracting from the nitty-gritty details of flag transfers.

An example of tables and membership expressions is shown in Figure 8. The table `[LOG,OP1]` contains logical assembly instructions and their operators. The first 6 instruction assign a value to a register. The last 6

```
[ARITLOG,OP1] := { (ADD_, "+"), (AND_, "&"), (OR_, "|"),
                   (SUB_, "-"), (XOR_, "^") }

ARITLOG[idx]^"iAL"  i8          *8* %AL := '%AL OP1[idx] i8
                                defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)
ARITLOG[idx]^"iAX"  i16         *16* %AX := '%AX OP1[idx] i16
                                defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)
ARITLOG[idx]^"mrb" eaddr, reg8  *8* eaddr := 'eaddr OP1[idx] 'reg8
                                defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)
ARITLOG[idx]^"mrw" eaddr, reg   *16* eaddr := 'eaddr OP1[idx] 'reg
                                defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)
ARITLOG[idx]^"rmb" reg8, eaddr  *8* reg8 := 'reg8 OP1[idx] 'eaddr
                                defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)
ARITLOG[idx]^"rmw" reg, eaddr   *16* reg := 'reg OP1[idx] 'eaddr
                                defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)
ARITLOG[idx]^"wb" eaddr, i8     *8* eaddr := 'eaddr OP1[idx] i8
                                defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)
ARITLOG[idx]^"b"  eaddr, i8     *8* eaddr := 'eaddr OP1[idx] i8
                                defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)
ARITLOG[idx]^"w"  eaddr, i16    *16* eaddr := 'eaddr OP1[idx] i16
                                defineflags(%SF,%ZF,%AF,%PF,%CF,%OF)
```

Figure 7: SSL definition of the arithmetic and logical instruction from the 80286 architecture

instruction do that and also affect the condition codes. When specifying the semantics of the instructions in this table, the extra functionality attached to the last 6 instructions can be specified by means of a membership expression; if the index in the table is between 6 and 11 (tables are indexed from 0), the instruction affects the condition codes as per specified.

```
# logical table
[LOG,OP1] := { (AND,"&"), (ANDN,"&~"), (OR,"|"),
              (ORN,"|~"), (XOR,"^"), (XNOR,"^~"),
              (ANDCC,"&"), (ANDNCC,"&~"), (ORCC,"|"),
              (ORNCC,"|~"), (XORCC,"^"), (XNORCC,"^~"),

LOG[idx]  rs1, reg_or_imm, rd      *32* r[rd] := 'r[rs1] OP1[idx] 'reg_or_imm
                                   (idx |= {6..11}) =>
                                             defineflags(%N, %Z, %V, %C)
```

Figure 8: SSL Specification for Rotates in the 80286

6.3 Modelling Higher Order Instructions

The semantic description of most assembly instructions is straight forward as most instructions are self-contained; that is, they refer to only arguments that come within the instruction itself. However, there are a few instructions which relate to other instructions, typically the *next* instruction in the instruction stream. These instructions are referred to as higher order instructions and deserve further explanation as to their semantic specification.

Delayed Instructions

On SPARC, delayed instructions take the form of a pair of instructions; the first one is a control transfer instruction and the second one (commonly referred to as the delay slot instruction) can be any type of instruction (although normally it is not another control transfer instruction). These instructions are used whenever transfering control to another memory location in the program, as the next instruction in the stream can actually be executed prior to the transfer of control by the first instruction. This is possible due to the architecture's pipeline.

Branch instructions have a further constraint—the delay slot instruction can be annulled or not depending on the value of the 'a' field in the instruction. A machine-dependent specification of the branches on SPARC follows:

```
# Jump table
[JUMPS,COND] :=
     { (BA_, 1), (BN_,0), (BNE_, ~'%Z), (BE_, '%Z),
       (BG_, ~('%Z |('%N ^ '%V))), (BLE_, '%Z |('%N^'%V)),
       (BGE_, ~('%N ^ '%V)), (BL_, '%N ^ '%V),
       (BGU_, ~('%C | '%Z)), (BLEU_, '%C | '%Z),
       (BCC_, ~'%C), (BCS_, '%C), (BPOS_, ~'%N),
       (BNEG_, '%N), (BVC_, ~'%V), (BVS_, '%V) }

JUMPS[idx]  disp22, a     *32* %nPC := ((COND[idx] = 1) ? '%PC + (4 * disp22) :
                                       ((a = 1) ? '%PC + 4 : '%nPC))
                          *32* %PC := ((COND[idx] = 0 and ('a = 1)) ?
                                       '%PC + 4 : '%PC)
```

Repeat String Instructions

On x86, the repeat instructions allow the next instruction in the stream to be repeated the number of times specified in the `cx` register based on an implicit condition in the repeat instruction itself. The next instruction in the stream must be a string instruction (i.e. one of cmps, lods, movs, scas, or stos, in either byte or word mode). The REP, REPNE and REPNZ instructions execute the next instruction in the stream while the value of the cx register is not zero; each iteration decreases the value of the register by one. This instruction could have been modelled with a loop construct in the language, however, we felt that SSL should not include loops as this will most likely be misused by writers of semantic specifications. Hence it was modelled with the equivalent of two global flags: Skip and Rpt (for skip and repeat).

The Rpt flag is set to 1 while the value of cx is greater than 0, and it is reset to 0 when the register becomes 0. This flag is used by the string instructions to update the value of the PC register prior to termination of the instruction, namely, by "going back" one instruction (to the repeat instruction) if the value of the flag was on. The Skip flag is then needed to "skip over" the string instruction in the last iteration. The SSL specification follows:

```
# REPT table for repeat instrs with condition 'cx > 0'
REPT := { REP_, REPNE_, REPNZ_ }

# REP uses Skip and Rpt registers to enable iteration
REPT[idx]     ('%CX = 0) =>
                        (*1* %Skip := 1
                         *1* %Rpt  := 0)
              ('%CX > 0) =>
                        (*16* %CX := '%CX - 1
                         *1* %Rpt := 1)

# STRS table for string instructions
STRS = { CMPS, LODS, MOVS, SCAS, STOS }

STRS[idx]b     ...  // other stuff here
               *16* %PC := '%PC + ('%Rpt ? -1 : 0)
```

7 Discussion

The specification of machine instructions using Object-Z was useful to learn about the instruction set supported by each machine. It provided an unambiguous way of specifying what each instruction was doing, and allowed the simplification of some of the machine dependencies available in ISP and natural language descriptions in the manual (such as memory alignment for example). It also forced us to really understand the architecture manual descriptions as you cannot specify something that you do not understand.

Given the ambiguity of natural language, extra knowledge was acquired through local architecture experts and reviewing other books. The specifications are long as they were done on an instruction at a time basis, hence they are 27 and 15 pages long for SPARC and 80286 respectively.

The refinement of the Object-Z specifications to the SSL language allowed for the compacting of the specifications by grouping common instructions into tables. The language was extensively revised for readability and expressability, and only simple constructs were allowed in the language. The SSL files for SPARC and 80286 are 210 and 382 lines long respectively (commented files).

We have implemented a semantic representation decoder tool (SRD) to parse SSL files and store them in a template format suitable for instantiation by individual machine instructions in a binary-code decoder tool.

The specifications as they stand are machine-dependent, in that they model architectural issues such as register windows and the nPC register on SPARC. For retargetable binary translation analysis, this representation will be transformed into a machine independent representation in order to perform analyses in a machine independent way. However, some of the initial analyses require the machine-dependent representation (for example, to determine the arguments to a call and the calling convention used), so both representations will be needed.

Higher order instructions require a semantic representation that is independent of the machine. For example, both the specifications of delay instructions and repeat instructions require "forced" changes of the program counter register, which is not normally user-controlled in machine programs. If binary translating from SPARC to x86, the nPC register is not available in the destination machine, and although it can be modelled by a dedicated register, the changes to the contents of the PC register with that of the nPC register would involve nasty code to get around the fact that the PC register cannot be assigned to directly other than via control transfer instructions. Allowing for this type of code would create a big overhead on execution time of the translated program.

8 Conclusions

We have specified the semantics of machine instructions for a RISC and a CISC processor using a formal description language, Object-Z, and derived from it a specification language called SSL for the description of semantics of machine instructions for a variety of machines.

SSL provides a simple language for specifying the semantics of machine instructions in a compact way. SRD, a semantic representation decoder tool implements SSL and provides an interface to users of this language to the NJMC toolkit via a template file. SRD provides an interface to this template file in order to get an instance of an instruction's description. This tool is suitable for decoding the semantics of instructions once the syntax has been decoded, and it fully integrates with the NJMC toolkit (a syntax decoder).

SSL has been integrated in a retargetable binary translation environment which aims at translating binary code from one machine to another via specification languages which specify the machine-dependent aspects of the translation process. In this way, machine-independent analyses will need to be written once and used in any number of machines. The current interface for SSL is written in the C++ language.

Acknowledgements

We would like to thank the members of the binary translation group, Mike van Emmerik, David Ung and Doug Simon, for useful discussions on the SSL language. We also thank Norman Ramsey for email discussions on the specification of machine instructions; he is specifying the semantics of machine instructions for a retargetable optimizing compiler using a pure functional language called λ-RTL [8].

References

[1] E.J. Chikofsky and J.H. Cross. Reverse engineering and design recovery: A taxonomy. *IEEE Software*, 7:13–17, January 1990.

[2] C. Cifuentes, S. Sendall, M. van Emmerik, D. Ung, D. Simon, and N. Ramsey. The UQ retargetable binary translator. Internal report, Binary translation group, Deparment of Computer Science, The University of Queensland, 1997.

[3] Digital. Freeport express. http://www.novalink.com/ freeport-express, 1995.

[4] R. Duke and G. Rose. Formal object-oriented specification and design using Object-Z. Book to be published, 1997.

[5] R.J. Hookway and M.A. Herdeg. Digital FX!32: Combining emulation and binary translation. *Digital Technical Journal*, 9(1):3–12, 1997.

[6] Intel. *Pentium Processor Family Developer's Manual – Volume 3: Architecture and Programming Manual.* Intel Corporation, 1995.

[7] J.R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *SIGPLAN Conference on Programming Languages, Design and Implementation*, pages 291–300, June 1995.

[8] N. Ramsey and J.W. Davidson. Specifying instructions' semantics using CSDL (preliminary report). Technical Report CS-97-31, University of Virginia, Department of Computer Science, Charlotesville, VA, November 1997.

[9] N. Ramsey and M. Fernández. The New Jersey machine-code toolkit. In *Proceedings of the 1995 USENIX Technical Conference*, pages 289–302, January 1995.

[10] N. Ramsey and M. Fernández. Specifying representations of machine instructions. *ACM Transactions of Programming Languages and Systems*, 19(3):492–524, 1997.

[11] S. Sendall. Semantics of machine instructions. Honours thesis, University of Queensland, Department of Computer Science, 1997.

[12] Sparc. *The SPARC Architecture Manual – Version 8.* Sparc International, Menlo Park, California, 1992.

[13] J.M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall, 2 edition, 1992.

[14] T. Thompson. An Alpha in PC clothing. *Byte*, pages 195–196, February 1996.