The Machine-SUIF SUIFvm Library

Release version 2.02.07.15

Glenn Holloway and Michael D. Smith
{holloway,smith}@eecs.harvard.edu
Division of Engineering and Applied Sciences
Harvard University

July 15, 2002

Abstract

The SUIFvm architecture is a virtual machine used as an intermediate target in translation from SUIF 2 form to instructions for a real target machine. In the Machine-SUIF system, all code generators for specific physical targets take their input in the form of SUIFvm code. SUIFvm instruction lists also serve as an intermediate form for certain kinds of code optimization.

This library defines the SUIFvm instruction set and the class `CodeGen` that is the basis for Machine-SUIF code generation.

Contents

1 Introduction

The SUIFvm library both extends Machine SUIF with new interface functionality and defines a target architecture. This duality makes the SUIFvm library unique within Machine SUIF. This duality also makes the SUIFvm library a poor candidate for use as a template for your new interface or target libraries–you will almost certainly want to define a new library that either only adds new interface functionality or defines a new target architecture.

We begin with a description of the SUIFvm instruction set. Sections 3 through 6 then describe the target-definition aspects of the SUIFvm library. Section 7 presents the interface functionality added by the SUIFvm library. In particular, it describes the object that allows us to use SUIFvm as an intermediate target in translation from SUIF 2 form to instructions for a real target machine. All of the existing code generators in Machine SUIF take their input in the form of SUIFvm code. The remaining sections define the context and initialization code required by a combination interface/target library.

2 The SUIFvm instruction set

Here is a description of the SUIFvm instructions. Readers familiar with SUIF 1 and *The SUIF Library Reference Manual* that documents it will recognize most of these descriptions.

- `NOP`. Do nothing at all. All of the operands for these instructions should be null.
- `LDA`. Move the address (not the dereferenced value) specified by the source operand into the destination.
- `LDC`. Move a constant value (the source operand) into the destination.
- `LOD`. Load the value at the address contained in the source operand into the destination operand.
- `STR`. Store the value in the source operand at the address contained in the destination operand.
- `MEMCPY`. Memory to memory copy. Load the value from the address in the source operand and store it at the address in the destination operand.
- `MOV`. Copy the value of the source operand to the destination operand.
- `CVT`. Convert the source operand to the destination type and put it in the destination operand.
- `NEG`. Negation. Change the sign of the value in the source operand and put the result in the destination operand.
- `ADD`. Add the values in the first two source operands and put the result in the destination.
- `SUB`. Subtract the value in the second operand from the value in the first source and put the result in the destination.
- `MUL`, `DIV`. Multiply or divide the value in the first source operand by the value in the second operand and put the result in the destination.
- `REM`, `MOD`. Remainder and modulus. These two instructions are very similar. Both divide the value in the first source operand by the value in the second operand to find the remainder or modulus. The `rem` instruction is identical to the modulus operator in ANSI C. That is, if either source operand is negative, the sign of the result is undefined and depends on the semantics of integer division. The `mod` instruction is the same except that its result is always guaranteed to be positive.
- `NOT`. Bit-wise inversion. Compute the one's complement negation of the value in the source operand and put the result in the destination.

- `AND`, `IOR`, `XOR`. Compute the bit-wise `and`, inclusive `or`, or exclusive `or` of the values in the first two source operands and put the result in the destination.
- `ASR`, `LSR`. Shift the value in the first source operand right by the amount specified in the second operand. `asr` performs sign extension; `lsr` does not.
- `LSL`. Shift the value in the first source operand left by the amount specified in the second operand.
- `MIN`, `MAX`. Minimum and maximum. The result value is the minimum or maximum, respectively, of the two source operands.
- `ABS`. Absolute value. Compute the absolute value of the source operand.
- `ROT`. Rotate the value in the first source operand left or right by the amount specified in the second operand. If the shift amount is positive, the value is rotated to the left (toward the most-significant bit); if it is negative, the value is rotated to the right.
- `SEQ`, `SNE`, `SL`, `SLE`. Comparison instructions. If the first source operand is equal, not equal, less than, or less than or equal, respectively, to the second operand, assign the integer value one to the destination operand. Otherwise, set the destination operand to zero. The source operands must be either the same integer or floating point type, or two (possibly different) pointer types.
- `JMP`. Unconditional jump to the `target` label.
- `JMPI`. Jump indirect. An unconditional jump to the computed address given in the source operand. The `target` field is null.
- `BTRUE`. Branch if true. If the source operand contains a true (non-zero) value, control is transferred to the code at the `target` label. Otherwise, it continues with the next instruction in sequential order. The source operand must have an integer type.
- `BFALSE`. Branch if false. If the source operand contains a false (zero) value, control is transferred to the code at the `target` label. Otherwise, it continues with the next instruction in sequential order. The source operand must have an integer type.
- `BEQ`,`BNE`,`BGE`,`BGT`,`BLE`,`BLT`. Branch if conditional compare true. The two source operands are compared according to the indicated condition. If the result is true, control is transferred to the code at the `target` label. Otherwise, it continues with the next instruction in sequential order. The source operands must either have the same integer or floating-point type, or else two (possibly different) pointer types.
- `MBR`. Multi-way branch instruction. Transfers control to one of several target labels depending on the value of the source operand, an integer. The value of the source operand is compared with a set of constants to choose the target label. If it matches one of these case constants, the instruction branches to the corresponding label. Otherwise, it branches to the default target label, which is the `target` field of the instruction.

  The case constants and target labels are stored in an `instr_mbr_targets` annotation of type `MbrNote`. (See the *Machine-SUIF Machine Library* document.) The case constants are guaranteed to be an increasing, non-empty sequence of integers.
- `CALL`. Call instruction. The called procedure is either specified in a non-null `target` field or through a non-null first source operand whose value is the address of the callee. (But not both: either the target symbol or the first source must be null.) The remaining source operands express the arguments. The destination operand gives the result location for a value-returning call; otherwise, it is null.
- `RET`. Return from a procedure. Only the first source operand is used and it is optional. If specified, it is the return value and may contain an operand of any type except array or function types. If the procedure's function type has void return type, the operand must be null; otherwise the operand must not be null and must have the same type as the return type of the procedure.

- `ANY`. Generic instruction, with arbitrary sources and destinations. It is expected that one or more annotations will specify the exact operation and possibly its semantics.
- `MRK`. A pseudo-instruction marking a position in the program. Used to hold miscellaneous annotations such as line numbers.

3 Opcode functions and data structures

This section defines the OPI functions and data structures associated with the SUIFvm opcodes.

3.1 Target-specific data structures

We start with the extensible opcode enumeration.

5a ⟨*SUIFvm opcodes* 5a⟩≡ (6b)

```
namespace suifvm {
enum {                  // SUIFvm opcodes
    NOP = 2,
    CVT, LDA, LDC,
    ADD, SUB, NEG,
    MUL, DIV, REM, MOD,
    ABS, MIN, MAX,
    NOT, AND, IOR, XOR,
    ASR, LSL, LSR, ROT,
    MOV, LOD, STR, MEMCPY,
    SEQ, SNE, SL, SLE,
    BTRUE, BFALSE,
    BEQ, BNE, BGE, BGT, BLE, BLT,
    JMP, JMPI, MBR, CAL, RET,
    ANY, MRK
};
} // namespace suifvm

#define LAST_SUIFVM_OPCODE suifvm::MRK
```

The `suifvm_opcode_names` table is the extensible map from a SUIFvm opcode to its name. The `suifvm_cnames` table is the extensible map from a SUIFvm opcode to a C-language operator. You can use the methods of the `Vector` template class along with a redefinition of `LAST_SUIFVM_OPCODE` to extend the SUIFvm instruction space.

5b ⟨*SUIFvm opcode vectors* 5b⟩≡ (6b) 5c ▷

```
extern Vector<char*> suifvm_opcode_names;
extern Vector<char*> suifvm_cnames;
```

We initialize these vectors using the following functions, invoked in this library's initialization routine.

5c ⟨*SUIFvm opcode vectors* 5b⟩+≡ (6b) ◁ 5b

```
void init_suifvm_opcode_names();
void init_suifvm_cnames();
```

3.2 OPI opcode functions

We provide the typical set of opcode functions.

6a ⟨*SUIFvm opcode OPI functions* 6a⟩≡ (6b)
```
bool target_implements_suifvm(int opc);
char *opcode_name_suifvm(int opc);

int opcode_line_suifvm();
int opcode_ubr_suifvm();
int opcode_move_suifvm(TypeId);
int opcode_load_suifvm(TypeId);
int opcode_store_suifvm(TypeId);

int opcode_cbr_inverse_suifvm(int opc);
```

3.3 Header file for `opcodes.h`

6b ⟨*suifvm/opcodes.h* 6b⟩≡
```
/* file "suifvm/opcodes.h" */

⟨Machine-SUIF copyright 17⟩

#ifndef SUIFVM_OPCODES_H
#define SUIFVM_OPCODES_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "suifvm/opcodes.h"
#endif

#include <machine/machine.h>

⟨SUIFvm opcodes 5a⟩

⟨SUIFvm opcode vectors 5b⟩

⟨SUIFvm opcode OPI functions 6a⟩

#endif /* SUIFVM_OPCODES_H */
```

4 `Instr` functions

The functions provided in this header file implement the instruction predicates of the OPI for the SUIFvm target.

6c ⟨*suifvm/instr.h* 6c⟩≡
```
/* file "suifvm/instr.h" */

⟨Machine-SUIF copyright 17⟩

#ifndef SUIFVM_INSTR_H
#define SUIFVM_INSTR_H

#include <machine/copyright.h>
```

```
#ifdef USE_PRAGMA_INTERFACE
#pragma interface "suifvm/instr.h"
#endif

#include <machine/machine.h>

bool is_ldc_suifvm(Instr*);
bool is_move_suifvm(Instr*);
bool is_cmove_suifvm(Instr*);
bool is_line_suifvm(Instr*);

bool is_ubr_suifvm(Instr*);
bool is_cbr_suifvm(Instr*);
bool is_call_suifvm(Instr*);
bool is_return_suifvm(Instr*);
bool is_binary_exp_suifvm(Instr*);
bool is_unary_exp_suifvm(Instr*);
bool is_commutative_suifvm(Instr*);
bool reads_memory_suifvm(Instr*);
bool writes_memory_suifvm(Instr*);
bool is_builtin_suifvm(Instr*);

#endif /* SUIFVM_INSTR_H */
```

5 Printing SUIFvm code

Recall that class `Printer` provides the specifics for printing machine code in the correct assembly-language form for a particular architecture. For the SUIFvm architecture, there is no well-defined ASCII assembly form, but printing code in this style is often helpful for debugging. When creating a `Printer` subclass for a real target, you should not use `PrinterSuifVm` as a model. Use the `Printer` object in the Alpha library, for example, as a guide.

7 ⟨*class* `PrinterSuifVm` 7⟩≡ (8a)

```
class PrinterSuifVm : public Printer {
  protected:
    virtual void print_instr_alm(Instr*);
    virtual void print_instr_cti(Instr*);
    virtual void print_instr_dot(Instr*);
    virtual void print_instr_label(Instr*);
    virtual void print_instr_user_defd(Instr*) { }

    virtual void print_opcode(Instr*);
    virtual void print_sym_disp(Opnd addr_sym, Opnd disp);
    virtual void print_address_exp(Opnd addr_exp);

    virtual char* size_directive(TypeId);
    virtual void process_value_block(ValueBlock*);

    // local variables used in print_var_def and its helpers
    char *cur_directive;
    int cur_opnd_cnt;

  public:
    PrinterSuifVm();

    virtual void start_comment() { fprintf(out, "\t# "); }
```

```
    virtual void print_instr(Instr*);
    virtual void print_opnd(Opnd);

    virtual void print_extern_decl(VarSym*);
    virtual void print_file_decl(int fnum, IdString fnam);
    virtual void print_var_def(VarSym*);

    virtual void print_global_decl(FileBlock*);
    virtual void print_proc_begin(ProcDef*);
    virtual void print_proc_decl(ProcSym*) { }
    virtual void print_proc_entry(ProcDef*, int file_no_for_1st_line);
    virtual void print_proc_end(ProcDef*);
};
```

8a ⟨*suifvm/printer.h* 8a⟩≡

```
/* file "suifvm/printer.h" */

⟨Machine-SUIF copyright 17⟩

#ifndef SUIFVM_PRINTER_H
#define SUIFVM_PRINTER_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "suifvm/printer.h"
#endif

#include <machine/machine.h>

⟨class PrinterSuifVm 7⟩

#endif /* SUIFVM_PRINTER_H */
```

6 C-language printing

Class `CPrinterSuifVm`, which refines `CPrinter`, is the analogue of `PrinterSuifVm` for generating C output files instead of assembly language. The methods that it overrides are those for printing SUIFvm instructions and global directives.[1]

8b ⟨*class* `CPrinterSuifVm` 8b⟩≡ (9c)

```
class CPrinterSuifVm : public CPrinter {
  public:
    CPrinterSuifVm();
    virtual ~CPrinterSuifVm() { delete [] print_instr_table; }

    virtual void print_instr(Instr *mi);
    virtual void print_global_decl(FileBlock *fb);

⟨CPrinterSuifVm protected parts 9a⟩
};
```

[1] At present, the only global directive is a `#include` which introduces a few useful abbreviations.

The printing method for a particular instruction is selected from a dispatch table that is indexed by opcode. The following methods are used as elements in that table.

9a ⟨CPrinterSuifVm *protected parts* 9a⟩≡ (8b) 9b ▷

```
  protected:
    virtual void print_instr_alm(Instr *);
    virtual void print_instr_cti(Instr *);
    virtual void print_instr_dot(Instr *);
    virtual void print_instr_mbr(Instr *);
    virtual void print_instr_cal(Instr *);
```

These other methods defined at this level:

9b ⟨CPrinterSuifVm *protected parts* 9a⟩+≡ (8b) ◁ 9a

```

    virtual void print_instr_user_defd(Instr *) { }
    virtual void print_opcode(Instr *mi);
    virtual void print_addr_binop(Instr *mi);
```

`print_user_defd` allows further derivation from this class: it deals with opcodes that are out of range for the target.

`print_opcode` simply prints the opcode of an instruction in C syntax, and `print_add_binop` prints an address as a binary expression.

6.1 Header file for module `cprinter.h`

The interface file has the following layout:

9c ⟨*suifvm/c_printer.h* 9c⟩≡

```
/* file "suifvm/c_printer.h" */

⟨Machine-SUIF copyright 17⟩

#ifndef SUIFVM_CPRINTER_H
#define SUIFVM_CPRINTER_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "suifvm/cprinter.h"
#endif

#include <machine/machine.h>

⟨class CPrinterSuifVm 8b⟩

#endif /* SUIFVM_CPRINTER_H */
```

7 Support for code generation

This section describes a class called `CodeGen` that supports translation of SUIFvm instructions to target-specific code. A target-specific instance of `CodeGen` is used by our `gen` pass to control its dispatch to code-generation methods for each kind of SUIFvm instruction.

An OPI user gets access to a target-specific `CodeGen` object by using the following OPI function.

9d ⟨*function* `target_code_gen` 9d⟩≡ (12b)

```
CodeGen* target_code_gen();
```

This is a target-specific function whose implementation is obtained from the like-named method of the class `SuifVmContext`, described in the next section.

7.1 Class `CodeGen`

We present three views of the `CodeGen` class. Those readers interested in the OPI methods for code generation to an existing target should read only the first subsection. The second subsection contains information relevant to those readers interested in defining a `CodeGen` object for a new target. The last subsection describes details of our implementation.

7.1.1 OPI details

The following class defines a `CodeGen` object:

10 ⟨*class* `CodeGen` 10⟩≡ (12b)

```
class CodeGen {
  public:
    virtual ~CodeGen() { }
    virtual void init(OptUnit *unit);   // must be called 1st!
    virtual void massage_unit(OptUnit *unit) { }
    virtual void finish(OptUnit *unit); // must be called last!

    void translate_instr(OptUnit *unit, Instr*, InstrList *receiver);
    void translate_list(OptUnit *unit, InstrList *in_place);

    void emit(Instr*);

⟨CodeGen private parts 12a⟩

⟨CodeGen protected parts 11⟩
};
```

Code generation is always done with respect to some `OptUnit`. When your optimization pass starts working on a new `OptUnit`, you must call `init` to initialize the code generator for this `OptUnit`. When your optimization pass is done with an `OptUnit` where you did some code generation, you must call `finish`. Under the current implementation, you must call `finish` if you called `init`, even if you did not call any other method of `CodeGen`.

The `massage_unit` method exists as a hook to make the code generation of an entire optimization unit easier. This hook allows us to massage the semantics of a SUIFvm optimization unit into something closer to the semantics of an optimization unit on the target. We used to use this hook in the Digital Alpha code generator, for example, to change the prototype of calls to functions that return structures. This call is not guaranteed to be idempotent, so please be careful how you use it.

Between calls to `init` and `finish`, you can translate SUIFvm instructions into target instructions using the `translate_instr` or `translate_list` methods, which translate a single SUIFvm instruction or a list of SUIFvm instructions into one or more target instructions.

7.1.2 Developing a target instance

You develop a `CodeGen` object for a specific target (or class of targets) by creating a derived class of `CodeGen`. The `CodeGenAlpha` class found in the `alpha` library is an example of such a derived class.

The derived class needs to define several methods that translate SUIFvm instructions to instructions of the target machine. The rest of this section describes this process in detail. During this discussion, we will make reference to the protected parts of the `CodeGen` class. The majority of the protected data members in the base class exist here simply because they are common to all derived classes of `CodeGen`.

11 ⟨`CodeGen` *protected parts* 11⟩≡ (10)

```
protected:
  virtual void translate_null(Instr*) = 0;
  virtual void translate_nop(Instr*) = 0;
  virtual void translate_cvt(Instr*) = 0;
  virtual void translate_lda(Instr*) = 0;
  virtual void translate_ldc(Instr*) = 0;
  virtual void translate_add(Instr*) = 0;
  virtual void translate_sub(Instr*) = 0;
  virtual void translate_neg(Instr*) = 0;
  virtual void translate_mul(Instr*) = 0;
  virtual void translate_div(Instr*) = 0;
  virtual void translate_rem(Instr*) = 0;
  virtual void translate_mod(Instr*) = 0;
  virtual void translate_abs(Instr*) = 0;
  virtual void translate_min(Instr*) = 0;
  virtual void translate_max(Instr*) = 0;
  virtual void translate_not(Instr*) = 0;
  virtual void translate_and(Instr*) = 0;
  virtual void translate_ior(Instr*) = 0;
  virtual void translate_xor(Instr*) = 0;
  virtual void translate_asr(Instr*) = 0;
  virtual void translate_lsl(Instr*) = 0;
  virtual void translate_lsr(Instr*) = 0;
  virtual void translate_rot(Instr*) = 0;
  virtual void translate_mov(Instr*) = 0;
  virtual void translate_lod(Instr*) = 0;
  virtual void translate_str(Instr*) = 0;
  virtual void translate_memcpy(Instr*) = 0;
  virtual void translate_seq(Instr*) = 0;
  virtual void translate_sne(Instr*) = 0;
  virtual void translate_sl(Instr*) = 0;
  virtual void translate_sle(Instr*) = 0;
  virtual void translate_btrue(Instr*) = 0;
  virtual void translate_bfalse(Instr*) = 0;
  virtual void translate_beq(Instr*) = 0;
  virtual void translate_bne(Instr*) = 0;
  virtual void translate_bge(Instr*) = 0;
  virtual void translate_bgt(Instr*) = 0;
  virtual void translate_ble(Instr*) = 0;
  virtual void translate_blt(Instr*) = 0;
  virtual void translate_jmp(Instr*) = 0;
  virtual void translate_jmpi(Instr*) = 0;
  virtual void translate_mbr(Instr*) = 0;
  virtual void translate_cal(Instr*) = 0;
  virtual void translate_ret(Instr*) = 0;
  virtual void translate_any(Instr*) = 0;
  virtual void translate_mrk(Instr*) = 0;

  InstrList *receiver;                // client-provided output list
  Note line_note;                     // prevailing source-line annotation

  // convenient variables
  OptUnit *cur_unit;                  // defined by init()
  const char *cur_unit_name;          // name of current procedure
  CProcedureType *cur_unit_type;      // type of current procedure
```

```
// variables that direct CodeGen
bool report_int_overflow, report_int_divideby0;

// working variables from cur_unit's stack_frame_info annotation
bool is_leaf;
bool is_varargs;
int max_arg_area;

virtual void transfer_params(OptUnit*, bool is_varargs) = 0;

CodeGen() { }
```

We start our discussion with the methods that are required to appear in any derived class. The derived class must define a constructor. You may also find it useful to redefine the virtual `init`, `massage_unit`, and `finish` methods. In the case of `init` and `finish`, you should always have your implementation of these methods include a call to the base class `init` or `finish` method. The base class methods contain operations that you still want to perform, e.g. the base-class `init` method will set many of the variables declared in ⟨`CodeGen` *protected parts* 11⟩. For example, the base class `init` method initializes the data members dependent upon the current `OptUnit`.

Each `translate_`*opcode* method is a pure virtual method. Each target library derives a subclass of `CodeGen` that implements these methods. Each implementation maps a SUIFvm instruction to one or more instructions in the target architecture. A translation function takes a single parameter: a pointer to an `Instr`. The instruction is assumed to be the SUIFvm instruction that you want to translate. The translated instructions are placed in an `InstrList` called `receiver`.

This class contains one other pure virtual method, `transfer_params` that the target library must define. It is invoked at the entry of a procedure to create a sequence of instructions that connects the call linkage conventions with the parameter symbols.

7.1.3 Implementation details

The following defines the private portion of the `CodeGen` class:

12a ⟨`CodeGen` *private parts* 12a⟩≡ (10)

```
  private:
    void translate_mi(Instr*);
```

The `translate_mi` routine simply does the work common among all of the public `translate_*` methods.

7.2 Header file for module `CodeGen.h`

The interface file has the following layout:

12b ⟨*suifvm/code_gen.h* 12b⟩≡

```
/* file "suifvm/code_gen.h" */

⟨Machine-SUIF copyright 17⟩

#ifndef SUIFVM_CODE_GEN_H
#define SUIFVM_CODE_GEN_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "suifvm/code_gen.h"
```

```
#endif

#include <machine/machine.h>

⟨class CodeGen 10⟩

⟨function target_code_gen 9d⟩

#endif /* SUIFVM_CODE_GEN_H */
```

8 The SUIFvm target-characterization context

This section begins with the declaration of the new interface context `SuifVmContext`. Any target library that supports code generation from SUIFvm needs to include a specialization of this context class in its context class declaration.

The other context class defined in this section is a refinement of the `MachineContext` class for use of `suifvm` as a target machine.

8.1 Class `SuifVmContext`

The `SuifVmContext` class has only a single method. It allows us to get access to the target code generator object.

13a ⟨*class* `SuifVmContext` 13a⟩≡ (15d)

```
class SuifVmContext {
  public:
    SuifVmContext();
    virtual ~SuifVmContext();

    virtual CodeGen* target_code_gen() const = 0;

  protected:
    ⟨SuifVmContext protected matter 13b⟩
};
```

13b ⟨`SuifVmContext` *protected matter* 13b⟩≡ (13a)

```
    mutable CodeGen *cached_code_gen;
```

8.2 Class `MachineContextSuifVm`

As a target, SUIFvm defines implementations for only the OPI functions defined in the Machine-SUIF `machine` library.

13c ⟨*class* `MachineContextSuifVm` 13c⟩≡ (15d)
```
class MachineContextSuifVm
      : public virtual Context, public virtual MachineContext {
  public:
    MachineContextSuifVm() { }
    virtual ~MachineContextSuifVm() { }

    ⟨MachineContextSuifVm generic-pointer method 14a⟩

    ⟨MachineContextSuifVm printer methods 14b⟩

    ⟨MachineContextSuifVm code-finalizer method 14d⟩

    ⟨MachineContextSuifVm instruction-predicate methods 14e⟩

    ⟨MachineContextSuifVm opcode-generator methods 15a⟩

    ⟨MachineContextSuifVm opcode query methods 15b⟩

    ⟨MachineContextSuifVm register query method 15c⟩
};
```

The target's generic-pointer type, which corresponds to `type_addr` in the `machine` library, is fetched by:

14a ⟨`MachineContextSuifVm` *generic-pointer method* 14a⟩≡ (13c)
```
TypeId type_addr() const;
```

The target's `Printer` pointer, which corresponds to global variable `target_printer` in the `machine` library, is fetched by:

14b ⟨`MachineContextSuifVm` *printer methods* 14b⟩≡ (13c) 14c ▷
```
Printer* target_printer() const;
```

Similarly for the `CPrinter` pointer, which corresponds to global variable `target_c_printer` in the `machine` library:

14c ⟨`MachineContextSuifVm` *printer methods* 14b⟩+≡ (13c) ◁ 14b
```
CPrinter* target_c_printer() const;
```

The target's `CodeFin` generator, which corresponds to global function `target_code_fin`, should never be called, because code finalization is not meaningful for the SUIFvm target.

14d ⟨`MachineContextSuifVm` *code-finalizer method* 14d⟩≡ (13c)
```
CodeFin* target_code_fin() const { claim(false); return NULL; }
```

The target-specific instruction predicates can be used through like-named methods of the `Context` class.

14e ⟨`MachineContextSuifVm` *instruction-predicate methods* 14e⟩≡ (13c)
```
bool is_ldc(Instr*) const;
bool is_move(Instr*) const;
bool is_cmove(Instr*) const;
bool is_line(Instr*) const;
bool is_ubr(Instr*) const;
bool is_cbr(Instr*) const;
bool is_call(Instr*) const;
bool is_return(Instr*) const;
bool is_binary_exp(Instr*) const;
bool is_unary_exp(Instr*) const;
bool is_commutative(Instr*) const;
bool is_two_opnd(Instr*) const;
```

```
bool reads_memory(Instr*) const;
bool writes_memory(Instr*) const;
bool is_builtin(Instr*) const;
```

The target-specific opcode generators can be used via like-named methods of the `Context` class.

15a ⟨`MachineContextSuifVm` *opcode-generator methods* 15a⟩≡ (13c)

```
int opcode_line() const;
int opcode_ubr() const;
int opcode_move(TypeId) const;
int opcode_load(TypeId) const;
int opcode_store(TypeId) const;
int opcode_cbr_inverse(int cbr_opcode) const;
```

The functions asking target-specific questions about opcodes can be used via like-named `Context` methods.

15b ⟨`MachineContextSuifVm` *opcode query methods* 15b⟩≡ (13c)

```
bool target_implements(int opcode) const;
char* opcode_name(int opcode) const;
```

SUIFvm has no registers, so we implement selected `reg_...` methods of class `MachineContext` to give null results.

15c ⟨`MachineContextSuifVm` *register query method* 15c⟩≡ (13c)

```
int reg_count() const
    { return 0; }
const NatSet* reg_allocables(bool maximals = false) const
    { static NatSetSparse empty; return &empty; }
const NatSet* reg_caller_saves(bool maximals = false) const
    { static NatSetSparse empty; return &empty; }
const NatSet* reg_callee_saves(bool maximals = false) const
    { static NatSetSparse empty; return &empty; }
```

8.3 Header file for `contexts.h`

15d ⟨*suifvm/contexts.h* 15d⟩≡

```
/* file "suifvm/contexts.h" */

⟨Machine-SUIF copyright 17⟩

#ifndef SUIFVM_CONTEXT_H
#define SUIFVM_CONTEXT_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "suifvm/contexts.h"
#endif

#include <machine/machine.h>
#include <suifvm/suifvm.h>

⟨class SuifVmContext 13a⟩

⟨class MachineContextSuifVm 13c⟩

#endif /* SUIFVM_CONTEXT_H */
```

9 SUIFvm library initialization

The header file `init.h` defines the initialization routine for the `suifvm` library and the string constant used to indicate use of this library as the current machine target.

16a ⟨*suifvm/init.h* 16a⟩≡

```
/* file "suifvm/init.h" */

⟨Machine-SUIF copyright 17⟩

#ifndef SUIFVM_INIT_H
#define SUIFVM_INIT_H

#include <machine/copyright.h>

#ifdef USE_PRAGMA_INTERFACE
#pragma interface "suifvm/init.h"
#endif

#include <machine/machine.h>

extern "C" void init_suifvm(SuifEnv* suif_env);

extern IdString k_suifvm;

#endif /* SUIFVM_INIT_H */
```

10 Header file for the SUIFvm library

Any OPI passes that use the OPI interface extensions defined by the `suifvm` library should include the following header file. Since this library is also a target library, any library that extends SUIFvm as a target machine should also include this header file.

16b ⟨*suifvm/suifvm.h* 16b⟩≡

```
/* file "suifvm/suifvm.h" */

⟨Machine-SUIF copyright 17⟩

#ifndef SUIFVM_SUIFVM_H
#define SUIFVM_SUIFVM_H

#include <machine/copyright.h>

#include <suifvm/init.h>
#include <suifvm/opcodes.h>
#include <suifvm/instr.h>
#include <suifvm/code_gen.h>
#include <suifvm/printer.h>
#include <suifvm/c_printer.h>
#include <suifvm/contexts.h>

#endif /* SUIFVM_SUIFVM_H */
```

The header files in the `suifvm` library depend only on the `machine` library.

11 Copyright

All of the code is protected by the following copyright notice.

17 ⟨*Machine-SUIF copyright* 17⟩≡ (6 8a 9c 12b 15 16)

```
/*
    Copyright (c) 2000 The President and Fellows of Harvard College

    All rights reserved.

    This software is provided under the terms described in
    the "machine/copyright.h" include file.
*/
```

12 Acknowledgments

This work was supported in part by an DARPA/NSF infrastructure grant (NDA904-97-C-0225), a NSF Young Investigator award (CCR-9457779), and a NSF Research Infrastructure award (CDA-9401024). We also gratefully acknowledge the generous support of this research by Advanced Micro Devices, Compaq, Digital Equipment, Hewlett-Packard, International Business Machines, Intel, and Microsoft.