The Simple-SUIF Compiler Guide

A simple interface to SUIF for compiler courses

Version 1.0

Stanford Compiler Group

This manual is part of the SUIF compiler documentation set.

1 Introduction

This document describes Simple-SUIF, a compiler system for real languages and machines that is simple enough for use in a compiler course. Simple-SUIF provides a simplified interface to the full SUIF compiler system which has been developed by the compiler research group at Stanford. Although it is not a production-quality compiler, SUIF runs on most UNIX systems, is capable of handling large programs written in ANSI C and FORTRAN 77, and can generate code for MIPS processors, which are used in DECstations and SGI workstations.

2 Using Simple-SUIF

Before running a Simple-SUIF program, the source file must first be run through the front-end of the SUIF compiler to produce a binary SUIF file[1]. This file can then be read by a Simple-SUIF program. Such a program is produced by linking with the Simple-SUIF library, which takes care of all of the details of working with SUIF files. Only the main program and a function to process each procedure must be provided. The main program is a simple C++ program to read the command-line arguments, initialize the SUIF library, and call the Simple-SUIF driver. Because this program rarely needs to be changed, most applications can just copy the main program from another Simple-SUIF program (e.g. printsimple).

The library's driver function reads each procedure and converts it to a list of Simple-SUIF instructions. It then passes the list of instructions to the `do_procedure` function provided by the user. The code in the `do_procedure` function can then analyze the instructions and implement various optimizations. When it is finished, it returns the list of instructions, which the driver then converts back to SUIF and writes to the output file.

[1] Simple-SUIF requires that its input files use a certain style of SUIF. Specifically, each procedure body must be a simple list of instructions. No abstract syntax trees (ASTs) or nested blocks are allowed. Complex instructions (e.g. array accesses) must be expanded to low-level code. If the input does not conform to this style, the Simple-SUIF library functions will probably fail.

3 The Simple-SUIF Library

The Simple-SUIF interface is designed for implementing back-end optimizations. To reduce the complexity, much of the information in a SUIF file is hidden behind this interface. This means that, although the Simple-SUIF interface could not be used to build a complete code generator, it is quite small and easy to use. There are two include files that define the Simple SUIF data structures and instruction opcodes. The `simple.h` file contains declarations for all of the data structures described in this section. The `simple_ops.def` file contains a list of the instruction opcodes and their formats.

3.1 Types

Types are used in two different places in Simple-SUIF. All Simple-SUIF instructions contain a field to specify the type of the result, although this field is not used if an instruction does not produce a value for the destination operand. Each variable symbol also has a type. Simple-SUIF types are represented by `simple_type` structures.

Each `simple_type` structure contains fields which specify the base type and the size in bits. The base types are:

`VOID_TYPE`
: Used to indicate that there is no value present.

`SIGNED_TYPE`
: Signed integers.

`UNSIGNED_TYPE`
: Unsigned integers.

`FLOAT_TYPE`
: Floating-point values.

`ADDRESS_TYPE`
: Pointers.

`RECORD_TYPE`
: Structures, unions, and arrays.

The `SIGNED_TYPE`, `UNSIGNED_TYPE`, `FLOAT_TYPE`, and `ADDRESS_TYPE` types are all considered to be *scalar* types.

Together, the base type and size constitute the only type information that is available in Simple-SUIF. The rest of the type information is recorded internally in the full SUIF type structures pointed to by the `suif_type` fields in the `simple_type` structures.

You cannot directly create new types or modify existing ones. Pointers to some commonly used types are declared in the Simple-SUIF library. Only use these predefined types or the types that are already present in a procedure. The `get_ptr_type` function may be used to find the type that is a pointer to another type. This may be needed to generate new memory accesses with the correct types.

Because a base type and size do not uniquely define a type, you must compare the addresses of two `simple_type` structures to determine whether or not they represent exactly the same type. However, many Simple-SUIF instructions only require that their operands have *compatible*, but not necessarily equivalent, types. Except for `RECORD_TYPE` types, two `simple_type` structures are compatible if their base types and sizes are equal.

3.2 Symbols

Symbols are used to refer to variables, labels, and procedures. Each symbol is represented by a unique instance of a `simple_sym` structure, which records the kind of symbol (`VAR_SYM`, `LABEL_SYM`, or `PROC_SYM`) and the symbol name. For variable symbols, the `simple_sym` structure includes another field to record the type of the variable. Other information about symbols is not accessible in Simple-SUIF but is recorded in the underlying SUIF symbol structures pointed to by the `suif_sym` fields in the `simple_sym` structures.

Rules for manipulating Simple-SUIF symbols are similar to those for types. As the data in a `simple_sym` structure does not uniquely define the symbol, always refer to symbols by the addresses of their `simple_sym` structures and use these addresses to test for equality. As with types, you cannot directly modify or create new `simple_sym` structures. However, the library provides several functions for creating new symbols. The `new_label` function creates and returns a new label symbol. New local variables are generated by the `new_register` function (see Section 3.3 [Registers], page 6). There is currently no way to create new global variables or procedure symbols.

3.3 Registers

The operands of SUIF instructions are contained in registers. These registers are divided into several classes: pseudo registers, temporary registers, and machine registers. The pseudo registers hold scalar variables that are guaranteed not to be aliased, temporary registers hold intermediate results within basic blocks, and machine registers represent the actual hardware registers in the target architecture. Within each class, each register is given a unique number. Each register contains a particular variable. Variables that are in registers can be used directly as instruction operands. Other variables must be accessed with load and store instructions.

The `simple_reg` structure is used to represent Simple-SUIF registers. This structure records the kind of the register (`PSEUDO_REG`, `TEMP_REG`, or `MACHINE_REG`), the register number, and the variable that is held in the register. The Simple-SUIF library keeps the `simple_reg` objects in a table, so each register is represented by a unique `simple_reg` instance and you can just refer to registers by the addresses of their `simple_reg` structures. The `NO_REGISTER` constant should be used as the address whenever a register field is empty.

The `new_register` function may be used to create new registers. You need to specify the register kind and the type of the variable to be stored in the register. The function then creates a new variable of the specified type. A new register is entered in the library's register table using the next available register number. You cannot create new machine registers. Note that the `new_register` function can also be used to create local variables even if they cannot be stored in registers. In that case, you can ignore the resulting registers and just use the variables that are created along with the registers.

In general, you may not modify any of the fields in a `simple_reg` structure. However, there is one exception to this rule. The register kind may be changed between `PSEUDO_REG` and `TEMP_REG`. The current implementation of the library guarantees that none of the registers in these two classes will ever have the same register number, so a register can be moved back and forth between the two classes without changing its number. Of course, if you change the kind of a register, it is up to you to make sure that it is used according to the conventions described below.

3.3.1 Pseudo Registers

Pseudo registers can only contain scalar variables that cannot be aliased. This means that global variables, local variables that have their addresses taken, and aggregates (e.g. structures and unions) cannot be stored in pseudo registers, and thus they must be accessed with separate load and store instructions. Because of the guarantee that pseudo registers are not aliased, they are ideal targets for optimization. Most basic optimizations cannot handle variables that are potentially aliased, so they ignore any variables that are not in pseudo registers. Only `cpy` (copy) instructions are allowed to assign to pseudo registers. This makes it easier to do value numbering and common subexpression elimination.

Another Simple-SUIF convention is that the pseudo registers with the lowest register numbers are the formal parameters of the procedure. For this reason, do not be surprised to find references to pseudo registers that have not yet been defined within the procedure.

3.3.2 Temporary Registers

Temporary registers are used to hold intermediate results of computations within a basic block. Each temporary register must be defined in exactly one place and used in exactly one place. It is illegal to assign a value to a temporary register that is never used, and you may not use the value in a temporary register more than once. The Simple-SUIF library checks to make sure that these conventions are followed.

3.3.3 Machine Registers

Machine registers correspond directly to the processor registers in the target architecture. They are only needed when doing register allocation. The machine registers are divided into general-purpose and floating-point registers. Each of these groups is further divided into callee-saved and caller-saved registers. The values in callee-saved registers are preserved across procedures but you have to explicitly save the caller-saved registers at each call site where they are live. The different classes of machine registers are identified by the following constants:

`CALLEE_GPRS`
: Callee-saved general-purpose registers.

`CALLER_GPRS`
: Caller-saved general-purpose registers.

`CALLEE_FPRS`
: Callee-saved floating-point registers.

`CALLER_FPRS`
: Caller-saved floating-point registers.

The `num_machine_regs` array contains the number of machine registers in each class. For example, `num_machine_regs[CALLER_GPRS]` is equal to the number of caller-saved general-purpose registers. Pointers to the actual `simple_reg` structures for machine registers are stored in the two-dimensional `machine_regs` array. The first array index identifies the machine register class and the second index identifies a particular register within the class. Note that the second array index ranges from zero to `num_machine_regs[class]`; it is *not* the register number.

The current library only handles machine registers for MIPS-based architectures. The general-purpose registers contain signed 32-bit variables, and the floating-point registers contain 64-bit floating-point variables. There is no way to access a floating-point register as a pair of 32-bit values. Support for other architectures may be added in the future.

3.4 Immediate Values

The `ldc` (load constant) instruction requires an operand that is a constant value stored in a `simple_immed` structure. These constant values may be integers or floating-point numbers or symbolic addresses. A `simple_immed` contains a union of these fields along with a tag to identify which field is used. The tag value may be `IMMED_INT`, `IMMED_FLOAT`, or `IMMED_SYMBOL`.

A symbolic address is specified by a symbol and a constant integer offset. The offset is specified in terms of bits. For example, to get the address of a field within a structure, the `simple_immed` would contain a pointer to the `simple_sym` of the variable with the offset equal to the number of bits between the beginning of the structure and the desired field.

3.5 Instructions

Instructions are the primary data structure in Simple-SUIF. Each procedure is represented by a doubly-linked list of instructions. These instructions contain references to the various types, symbols, and registers. Simple-SUIF instructions resemble assembly language instructions. Each one performs a relatively simple operation specified by its opcode. Most instructions have several operands in registers.

Instructions are stored in `simple_instr` structures. All instructions share several fields: the opcode, the result type, and pointers to the next and previous instructions. Since different instructions require different kinds of operands, the rest of the fields in the `simple_instr` structure are specific to the instruction format. The `simple_op_format` function may be used to identify the format used for a particular opcode. The possible format values are:

`BASE_FORM`
: This format provides a destination register and two source registers and is used for most of the opcodes. Not all of the registers are always used. For example, `nop` (no operation) instructions use none of the registers, and `cpy` (copy) instruction only use the destination and one of the source registers.

`BJ_FORM` Branch and jump instructions use this format which includes a reference to the target label and an optional source register. The source register is only used with conditional branch instructions.

`LDC_FORM` Load constant (`ldc`) instructions use this format. It includes a destination register and a `simple_immed` structure to hold the immediate value. See Section 3.4 [Immediate Values], page 9.

`CALL_FORM`
This format is only used for procedure `call` instructions. There are at least two register operands: the destination and the address of the procedure to be called. The format also includes a field to record the number of arguments and an array of the argument registers.

`MBR_FORM` Multi-way branch (`mbr`) instructions use this format. The fields include a source register, an integer offset, a default target label, and an array of other target labels.

`LABEL_FORM`
Label (`lab`) instructions identify the position of a label in the instruction list. They use this format which only contains a reference to the label symbol.

The result type of an instruction identifies the type of the value produced for the destination register. If the instruction format does not contain a destination register or if the particular opcode does not use the destination, the result type should always be a `VOID_TYPE`. Otherwise, the result type must always be supplied and should generally be the same as the type of the variable in the destination register. In rare cases, the destination register field may be set to `NO_REGISTER` even though the instruction produces a result. When this happens, the result type should indicate the type of the value that is actually produced even though that value is unused.

Unlike the other Simple-SUIF data structures, instructions can be freely modified and rearranged. The library includes functions to help create new instructions and deallocate them when they are no longer needed. The `new_instr` function creates a new `simple_instr` object given the opcode and result type. The other fields must be filled in separately. Note that this function is merely a convenience; nothing special is required when creating new instructions, and users are free to create them by calling `malloc` and setting the opcode and result type manually. The `free_instr` function deallocates the storage used by an instruction. Again, this function does nothing magic and is only provided for convenience.

The following table lists all of the Simple-SUIF opcodes. Each opcode has an internal representation that is a member of the `simple_op` enumeration. The `simple_op_name` function may be used to produce a textual representation for an opcode. The table lists both forms. Unless indicated otherwise, the opcodes use the `BASE_FORM` format.

`nop NOP_OP`
: Do nothing at all. All of the register operands for these instructions should be set to `NO_REGISTER`, and the result type should be a `VOID_TYPE`.

`load LOAD_OP`
: Load the value at the address contained in the `src1` register and put it in the `dst` register. The result type may be any type and indicates the type of the value being loaded. The type of the variable in `src1` must be a pointer to the result type. The `src2` register is not used.

`str STR_OP`
: Store the value in the `src2` register at the address contained in the `src1` register. Both registers must be specified. The `src2` register may have any type. The `src1` register should contain a variable that is a pointer to the type of the variable being stored. The `dst` register is not used.

`mcpy MCPY_OP`
: Memory-to-memory copy. Load the value from the address in the `src2` register and store it at the address in the `src1` register. Objects of any type may be copied. Both of the source registers must be pointers to the type of the object that is being copied. The `dst` register is not used.

`cpy CPY_OP`
: Copy the `src1` register to the `dst` register. The `src2` register is not used. The result type must be compatible with the type of the source register but need not necessarily be equivalent. Only scalar types are allowed here.

`cvt CVT_OP`
: Convert the `src1` register to the result type and put it in the `dst` register. The `src2` register is not used. Conversions should be performed in steps, changing only one attribute at a time. For example, when converting from an unsigned 8-bit type to a signed 32-bit type, use one `cvt` instruction to change the size and another to make it signed. Conversions between integer and floating-point types should always be made using types that are as close as possible in size. Nothing can be converted to or from a `VOID_TYPE` type or a `RECORD_TYPE`. `ADDRESS_TYPE` types can only be converted to and from integer types.

`ldc LDC_OP`
: Load a constant value. This instruction uses the `LDC_FORM` format. The `value` operand is an immediate constant stored in a `simple_immed` structure (see Section 3.4 [Immediate Values], page 9). If the immediate value is an integer, the result type must be a `SIGNED_TYPE`, `UNSIGNED_TYPE` or `ADDRESS_TYPE` type. The result type should be a `FLOAT_TYPE` type if the immediate is a floating-point value. And finally, if the immediate value is a symbolic address, the result type should always be an `ADDRESS_TYPE` type.

`neg` `NEG_OP`

Negation. Change the sign of the value in the `src1` register and put the result in the `dst` register. The `src2` register is unused. The result type and the type of the registers must be compatible integer or floating-point types.

`add` `ADD_OP`

Add the values in the `src1` and `src2` registers and put the result in the `dst` register. Except for pointer additions, the result type and the types of the registers must be compatible integer or floating-point types. Pointer addition is a special case. One of the source registers may have an `ADDRESS_TYPE` type, as long as the other source register contains a variable with an integer type of the same size as the pointer; the result type must also be an `ADDRESS_TYPE` type.

`sub` `SUB_OP`

Subtract the value in the `src2` register from the value in the `src1` register and put the result in the `dst` register. Except for pointer subtractions, the result type and the types of the registers must be compatible integer or floating-point types. There are two special cases for pointer subtractions. In either case, the `src1` register must have an `ADDRESS_TYPE` type. First, the `src2` register may have an integer type of the same size as the pointer to produce an `ADDRESS_TYPE` value in the `dst` register. Second, the `src2` register may be another `ADDRESS_TYPE` value to produce a value in the `dst` register with an integer type that is the same size as the pointers.

`mul` `MUL_OP`

`div` `DIV_OP`

Multiply or divide the value in the `src1` register by the value in the `src2` register and put the result in the `dst` register. The result type and the types of the registers must be compatible integer or floating-point types. Integer multiplication and division are defined according to the rules for ANSI C.

`rem` `REM_OP`

`mod` `MOD_OP`

Remainder and modulus. These two instructions are very similar. Both divide the value in the `src1` register by the value in the `src2` register to find the remainder or modulus. The `rem` instruction is identical to the modulus operator in ANSI C, and the `mod` instruction is the same except that its result is always guaranteed to be positive. The result type and the types of the destination and source registers must be compatible integer types.

`not` `NOT_OP`

Bit-wise inversion. Compute the one's complement negation of the value in the `src1` register and put the result in the `dst` register. The `src2` register is not used. The result type and the types of the registers must be compatible `UNSIGNED_TYPE` types.

`and` `AND_OP`

`ior` `IOR_OP`

`xor` `XOR_OP`

Compute the bit-wise AND, inclusive OR, or exclusive OR of the values in the `src1` and `src2` registers and put the result in the `dst` register. The result type and the types of the registers must be compatible `UNSIGNED_TYPE` types.

`asr` `ASR_OP`

`lsr` `LSR_OP`

`lsl` `LSL_OP`

Shift the value in the `src1` register right or left by the amount specified in the `src2` register. The variable in the `src2` register must always have an `UNSIGNED_TYPE` type. The `asr` instruction performs sign extension and requires that the result type and the types of the `dst` and `src1` registers be compatible `SIGNED_TYPE` types. The `lsr` instructions does not perform sign extension and requires that the result type and types of the `dst` and `src1` register be compatible `UNSIGNED_TYPE` types. Sign extension is not an issue for left shifts, so the `lsl` instruction only requires that the result type and the types of the `dst` and `src1` register be compatible integer types.

`rot` `ROT_OP`

Rotate the value in the `src1` register left or right by the amount specified in the `src2` register. The variable in the `src2` register must always have an `SIGNED_TYPE` type. If the shift amount is positive, the value is rotated to the left; if it is negative, the value is rotated to the right. The result type and the types of the `dst` and `src1` registers must be compatible integer types.

`seq` `SEQ_OP`

`sne` `SNE_OP`

`sl` `SL_OP`

`sle` `SLE_OP`

Comparison instructions. If the `src1` register is equal, not equal, less than, or less than or equal, respectively, to the `src2` register, assign the integer value one to the `dst` register. Otherwise, set the `dst` register to zero. The result type must always be a `SIGNED_TYPE` type. The source registers must have compatible scalar types.

`jmp` `JMP_OP`

Unconditional jump. This instruction uses the `BJ_FORM` format, but the `src` register is unused. The flow of control is unconditionally transferred to the code at target label.

`btru` `BTRUE_OP`

`bfls` `BFALSE_OP`

Branch if true or false. This instruction uses the `BJ_FORM` format. If the `src` register, which must have an integer type, contains a true (non-zero) or false (zero) value, respectively, the flow of control is transferred to the code at the target label. Otherwise, it continues with the next instruction in sequential order.

`mbr` `MBR_OP`
: Multi-way branch. This instruction uses the `MBR_FORM` format, which includes an array of target labels. The number of labels in the array is identified by the value of the `ntargets` field. Control is transferred to one of these labels depending on the value in the `src` register. The variable in the `src` register must have a `SIGNED_TYPE` or `UNSIGNED_TYPE` type. The integer value in the `offset` field is subtracted from the value in the `src` register and the result is used to index into the array of target labels. If the index is within the range of the array, the instruction branches to the label at that position in the array; otherwise, it branches to the default label in the `deflab` field.

`lab` `LABEL_OP`
: Label pseudo-instruction. This instruction uses the `LABEL_FORM` format. No operation is performed by a label instruction. Its only purpose is to mark the location of a label symbol in the instruction list. The `lab` field must be a pointer to the `simple_sym` for the label.

`call` `CALL_OP`
: Call a procedure. This instruction uses the `CALL_FORM` format. The `proc` register must hold a pointer to the procedure to be called. The type of the variable in that register must be a pointer to the type of the procedure. Note: Simple SUIF currently cannot represent procedure types, so you must use the pointer types that are used in the input list when referring to procedure addresses. The result type of the `call` instruction must match the return type of the procedure. If the result type is not a `VOID_TYPE` type, the `dst` register will be assigned the value returned by the procedure. The `nargs` field in the `call` instruction indicates how many arguments are in the `args` array. Each entry in the `args` array is a register holding the value of an argument to the procedure.

`ret` `RET_OP`
: Return from a procedure. Only the `src1` register is used and it is optional. If specified, it is the return value and may contain a variable of any type.