

The SimpleScalar Instruction Tool (SSIT) Manual

B.H.H. (Ben) Juurlink Demid Borodin

Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
Mekelweg 4, 2628 CD Delft, The Netherlands
Phone: +31 15 2787362, Fax: +31 15 2784898.
E-mail: {benj,demid}@ce.et.tudelft.nl

1 Introduction

Delft University of Technology is implementing a tool chain for modification of the SimpleScalar Tool Suite version 3.0.

SimpleScalar [1, 2] is a set of processor simulators. According to [3], the SimpleScalar tools are widely used for research and education purposes in the computer architecture field, a large part of papers published in the top computer architecture conferences uses them to evaluate the proposed designs.

Besides the simulators themselves, there is a version of the GNU compiler and binary utilities for SimpleScalar.

SimpleScalar tool set is designed to be extensible, allowing to integrate innovative parts relatively easily. One of the facilities offered by SimpleScalar is a possibility to add new instructions to the existing instruction set. In order to eliminate the need of changing the assembler accordingly, the existing instructions can be annotated to synthesize the new ones.

However, extending the simulators to support new instructions and corresponding functional units is not a trivial task, at least not for novices at SimpleScalar. Neither is writing assembly code with annotated SimpleScalar instructions, since they are much less human-readable than the actual instructions' names, so it is significantly error-prone. To facilitate these tasks, the SimpleScalar Instruction Tool (SSIT) [4] described in this manual has been created.

A user supplies a configuration file, and SSIT performs the following two tasks based on it: first, it modifies some of the source files of SimpleScalar so

that the simulators can execute the new instructions after it is recompiled; second, it replaces readable instructions in assembly source files by corresponding annotated instructions so that the GNU assembler can assemble them. The configuration file lists the new instructions with their functional implementation in the C language, and (if necessary) the new functional units used by these instructions.

Section 2 describes the SSIT configuration file in detail. Section 3 explains how SSIT works with SimpleScalar. Finally, Section 4 presents SSIT's command line interface and gives some usage examples.

2 SSIT Configuration File

SSIT provides a way to introduce new instructions in the SimpleScalar simulators and to extend the architecture with new functional units. In order to do this, a user should supply a *configuration file* specifying the new instruction(s) and, if necessary, functional unit(s).

A configuration file contains a number of sections. Each section begins with the percentage sign (%) after a newline) followed by the name of the section. SSIT supports two sections: (1) **FUNCTIONAL_UNITS** in which the new functional units are described and (2) **MACHINE.DEF** in which the new instructions are specified. There can also be other sections, this will let SSIT give the “unknown section” warning and skip it (possibly with other warnings from the parser). There can be an arbitrary number of **FUNCTIONAL_UNITS** and **MACHINE.DEF** sections in a configuration file, all the information from them will be collected. Everywhere except for inside the **MACHINE.DEF** section, comments, which are skipped by the parser, can appear. A comment starts with the sharp (#) character and lasts until the end of the line.

If SSIT discovers an error in a section, the rest of that section is skipped, and SSIT tries to process the following sections. Then it proceeds further with updating the SimpleScalar and assembly sources. However, this can cause unpredicted application errors because the configuration structure might be damaged.

2.1 The FUNCTIONAL_UNITS Section

This section defines the functional units needed for the new instructions. It is used only in `sim-outorder` simulator for performance simulation. If only functional simulation is important, it is possible to use any existing SimpleScalar resource classes and to skip the **FUNCTIONAL_UNITS**

section in the SSIT configuration file. For example, *FUClass_NA* can be used, which means that the instruction does not use any functional unit.

A new functional unit is described using the following syntax:

```
<name>, <quantity>,  
    <class>, <op_latency>, <is_latency> [,  
    <class>, <op_latency>, <is_latency> [...]]
```

where

name — (a string) — the name of the functional unit.

quantity — (a number) — the number of instances of this unit.

class — (a string) — the resource class. Instructions using this resource class will be able to execute on this functional unit.

op_latency — (a number) — the operation latency of the resource class, i.e., the number of cycles until the result is ready for use.

is_latency — (a number) — the issue latency of the resource class, i.e., the number of cycles before another operation can be issued on this resource.

There can be multiple resource classes in one functional unit. If there is a comma after **is_latency**, another resource class belonging to this functional unit is expected to follow.

The elements should not necessarily be on the same line, and even a new functional unit's description is not required to appear on its own line. However, it is advised as being more readable.

See the Figure 1 for an example of a new functional unit's declaration.

2.2 The MACHINE.DEF Section

This section introduces the new instructions themselves. For each new instruction, the following should be specified:

```
#define INSTRUCTION_NAME_IMPL {                               \  
    <implementation of the new instruction in C>             \  
}                                                                 \  
DEFINST( "<instruction.name>",    <operands>,  
        "<annotated instruction's name>",  
        <FU class>,              <iflags>,  
        <odep1>, <odep2>,        <idep1>, <idep2>, <idep3>  
)
```

where

instruction.name — the readable name of the new instruction. May contain letters, digits and dots.

annotated instruction's name — the name of an existing SimpleScalar instruction with the same number and type of the operands which will be annotated to pass the new instruction through the SimpleScalar assembler.

operands — the instruction operands' specification (see the comments in the beginning of the file `machine.def` of SimpleScalar for more information).

FU class — the resource class which is needed for the instruction execution. This can be an existing SimpleScalar resource class or a new one defined as `class` in the section “FUNCTIONAL_UNITS” (see Section 2.1).

iflags — instruction flags (see `machine.def`).

odep1, odep2 — output dependency designators. They specify which registers are modified by the instruction and are used to determine data dependencies. See `machine.def` for more information. `DNA` is used to specify that there is no dependency.

idep1, idep2, idep3 — input dependency designators. They specify which registers are read by the instruction. See `machine.def` for more information. `DNA` is used to specify that there is no dependency.

SSIT performs a validation of an instruction's definition as much as possible at this stage. Except the syntax check, SSIT performs the following tests:

- Makes sure that every `#define INSTRUCTION_NAME_IMPL` is followed by an appropriate `DEFINST`.
- Checks if the part `INSTRUCTION_NAME` of `#define INSTRUCTION_NAME_IMPL` above matches the `instruction.name`. Because the dots are allowed in an instruction's name but not in identifiers in the C language, in place of dots in `instruction.name` underscores must appear in `INSTRUCTION_NAME`.

- Makes sure that the instruction chosen to be annotated exists in SimpleScalar.
- Checks if the operands' specification of the new instruction matches that of the annotated instruction. If they are not the same, gives a warning, leaves the specification as is, and continues processing the current section (this is not considered as a severe error).
- Checks if the resource class specified exists among the standard SimpleScalar resource classes and those defined up to this point by SSIT. This means that if a functional unit is defined later in the same configuration file (has not been processed yet), this test will fail.

In the instructions' definition **DEFINST** of SimpleScalar in `machine.def`, there are also fields *enum* and *opcode* that are not present in SSIT **DEFINST**. SSIT generates them automatically. The field *enum* is obtained based on the instruction's name. The *opcode* is generated by searching for possible unused values from 0 to *MD_MAX_MASK*. *MD_MAX_MASK* is equal to 2048 by default (taken from the SimpleScalar file `machine.h`). If there is a large number of new instructions in a SSIT configuration file, there could be not enough available opcodes for all of them, and SSIT would give the error "No masks (opcodes) available any more...". In this case the value of *MD_MAX_MASK* can be changed in the SimpleScalar source (the file `machine.h`) and, accordingly, in the SSIT source (the file `config.h`).

Some extra text is allowed before **#define INSTRUCTION_NAME_IMPL**. This could be some useful *#define*-s, C-style comments etc. It goes to `machine.def` as is, without any validation.

The implementation of an instruction is defined as a C-style line (until a newline which is not escaped), it also goes to `machine.def` as is.

Figure 1 presents an example of SSIT configuration file for the new instruction `avg.ssit`. The instruction uses the resource class *Avg* from the new functional unit *AVG*, whose operation latency is 2 clock cycles, the issue latency is 1 cycle, and one instance of which is available. `avg.ssit` uses the SimpleScalar instruction `add` as the annotated instruction and has corresponding operands. The input of `avg.ssit` depends on the registers *RS* and *RT* (these are SimpleScalar macros, see `machine.def` for more information), and the output dependence is the register *RD*. The extra text before **#define AVG_Ssit_IMPL** contains the comments and the definition of *DEBUG_STREAM* which is used in the *fprintf* function to print the debug information at the execution stage of the instruction.

```

1  %FUNCTIONAL_UNITS

    AVG, 1, Avg, 2, 1

5  %MACHINE.DEF

    /* This is "extra text" for the instruction "avg.ssit" */
    #define DEBUG_STREAM  stdout

10     /***** avg.ssit *****/

    #define AVG_SUIT_IMPL {
        sword_t result = (GPR(RS)+GPR(RT)) >> 1;
        fprintf( DEBUG_STREAM, "avg: %d\n", result );
15     SET_GPR(RD, result);
    }
    DEFINST( "avg.ssit",  "d,s,t",  "add",
        Avg,              F_ICOMP,
        DGPR(RD),DNA,     DGPR(RS),DGPR(RT),DNA
20 )

```

Figure 1: SSIT configuration file for the instruction `avg.ssit`.

3 How Does SSIT Work?

SimpleScalar instruction has a 16-bit opcode and 16-bit annotation fields. To avoid the need to change the assembler for support of new instructions, they are compiled with the opcodes of existing SimpleScalar instructions, and the annotation is used to distinguish them.

SimpleScalar uses the enumeration *md_opcode* where the simulator's inner opcodes of the instructions are listed (these opcodes are different from those given in the `machine.def` file). The enumeration is defined in the file `machine.h`, it is initialized using the file `machine.def`. SSIT adds the new instructions' definitions into `machine.def` after the definition of the last instruction `ctc1`. In this way it populates the enumeration *md_opcode* with the new instructions. SSIT changes the source file `loader.c` so that when loading a program, for each annotated instruction SimpleScalar obtains its inner opcode by adding the annotation value to the largest existing opcode (of the instruction `ctc1`).

To integrate the new functional units and corresponding resource classes into SimpleScalar, SSIT adds them to the enumeration *md_fu_class* in the

file `machine.h` and to the array `md_fu2name` in the file `machine.c`. It also extends the resource pool definition `fu_config` in the file `sim-outorder.c`. Besides that, SSIT changes the maximum allowed number of resource classes which is defined in the file `resource.h` according to the number of new resource classes.

4 Usage

SSIT is a command line tool. To get the SSIT usage instructions, it is possible to use the command line option `-h` (`--help`). Alternatively, SSIT can be run without any arguments. This lets SSIT print the usage instructions and try to process the default configuration file.

The SSIT usage follows:

```
USAGE:  ssit [options]
```

Options:

```
-c <filename> or --config <filename>
    Specify the configuration file name
    Default: "ssit_config.cfg"
-i <filename> or --infile <filename>
    Specify the input file name
    Default: "infile.s"
-o <filename> or --outfile <filename>
    Specify the output file name
    Default: "outfile.s"
-u <true/false> or --update <true/false>
    Update the SimpleScalar source files
    Default: "false"
-s <dirname> or --ssdir <dirname>
    Specify the SimpleScalar source directory
    Default: "<default path>/"
-h or --help
    Print usage instructions
```

SSIT updates the SimpleScalar source files only if the `-u` (`-update`) option is set to "true", and processes the assembly file only if at least one of the options `-i` (`-infile`) or `-o` (`-outfile`) is explicitly set in a command line.

If SSIT is invoked without any arguments, it prints the usage instructions (assuming that the user might not know how to use it) and tries to proceed

using the default arguments. Because by default the option `--update` is *false* and SSIT processes an assembly file only if one of the options `--infile` or `--outfile` is explicitly set, SSIT actually only parses the default configuration file *ssit_config.cfg* in this case. In this way (or by setting only the option `--config`), a configuration file can be tested for syntax errors.

4.1 SSIT Usage Examples

```
ssit
or
ssit --help
```

Print usage instructions and try to parse the default configuration file *ssit_config.cfg* if it exists. Test it for errors that can be deduced by SSIT.

```
ssit --config filename.cfg
```

Parse the configuration file *filename.cfg*, test it for errors that can be deduced by SSIT.

```
ssit --infile asm.s
```

Use the default configuration file *ssit_config.cfg*. Process the assembly file *asm.s*, producing the default output assembly file *outfile.s*.

```
ssit --ssdir $HOME/simplesim-3.0 --update true
```

Use the default configuration file *ssit_config.cfg*. Update the SimpleScalar source files in the directory *\$HOME/simplesim-3.0*.

```
ssit --outfile asm.s -u true
```

Use the default configuration file *ssit_config.cfg*. Process the default assembly file *infile.s*, producing the output assembly file *asm.s*. Update the SimpleScalar source files in the default directory.

```
ssit -c conf.cfg -i in.s -o out.s -u true -s simplesim-3.0
```

Use the configuration file *conf.cfg*. Process the assembly file *in.s*, producing the output assembly file *out.s*. Update the SimpleScalar source files in the directory *simplesim-3.0*.

5 Configuration of SSIT

SSIT needs the file `machine.def` from SimpleScalar to be compiled. For this, the value of the variable `SIMPLESCALAR_DIR` in the `Makefile` of SSIT must be set to the directory where the SimpleScalar sources are. Another way is to copy the file `machine.def` into the SSIT source directory.

To update the SimpleScalar source files, SSIT needs to know the SimpleScalar source directory. By default, the directory specified as `SIMPLESCALAR_DIR` in the `Makefile` is used. To change the default value, the command line option `-s (--ssdir)` can be used.

A user may want to change something in the default SSIT configuration. This is possible via editing the source file `ssit_config.h` and recompiling SSIT. Among the configurable settings, there are the default input/output/configuration filenames, debug/error/log streams, the maximum length of a word that the parser can recognize, the maximum possible length of the “extra text” before an instruction’s implementation, the maximum possible length of an instruction’s implementation, the maximum length of a line in a processed file etc. The “maximum possible length” values play an important role. SSIT was designed to allow reasonable sizes for a general case. The sizes are large enough to handle expected (general) cases, but keep the memory consumption (and hence the application’s execution time) at the minimum level. In some special case, the maximum size of, for example, an instruction’s implementation, can be not sufficient. Then it should be changed accordingly in the file `ssit_config.h`. Via changing the “maximum possible length” values, the memory consumption and the execution speed of SSIT can be altered. See the comments in the file `ssit_config.h` for more information.

References

- [1] Doug Burger and Todd M. Austin. The SimpleScalar Tool Set, Version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.
- [2] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An Infrastructure for Computer System Modeling. *Computer*, 35(2):59–67, 2002.
- [3] SimpleScalar LLC webpage. <http://www.simplescalar.com/>.
- [4] SSIT, SSAT and SSIAT webpage. <http://ce.et.tudelft.nl/~demid/SSIAT/>.