

The SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT)

B.H.H. (Ben) Juurlink Demid Borodin Roel J. Meeuws Gerard Th. Aalbers
Hugo Leisink

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,rmeeuws,gtaalbers}@ce.et.tudelft.nl

Abstract

The SimpleScalar is one of the most often used simulator tool sets in computer architecture research. In this paper we present two tools that substantially simplify synthesizing new instructions and architectural units in SimpleScalar. The tools enable users to extend the simulators without a deep knowledge of the SimpleScalar source code. Besides that, the tools allow using the real names of the new instructions and registers instead of the unreadable instructions with annotations in SimpleScalar assembly code.

Keywords: Processor simulator.

1 Introduction

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 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 Section 2 of this work has been created.

SSIT provides a way to introduce new instructions in the SimpleScalar simulators and to extend the architecture with new functional units. However, it can work only with the registers existing in SimpleScalar. Sometimes it is desirable or even necessary to introduce new registers or aliases to the existing registers. This is not an easy task to do by hand, so

the SimpleScalar Architecture Tool (SSAT) [4] described in Section 3 has been developed to facilitate this task.

2 The SimpleScalar Instruction Tool (SSIT)

SimpleScalar allows to synthesize new instructions without having to change and recompile the assembler, by attaching annotations to the existing instructions' opcodes. However, instructions with annotations are hard to read and, therefore, error-prone. The SimpleScalar Instruction Tool (SSIT) allows to use readable instructions in the assembly files and performs two tasks: first, it replaces readable instructions by corresponding annotated instructions; second, it modifies some of the source files of SimpleScalar so that the simulators can execute the new instructions.

In order to synthesize new instruction(s), a user should supply a *configuration file* specifying the new instruction(s) and, if necessary, functional unit(s). The 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.

2.3 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

```

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_SSIT_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`.

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.

2.4 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` to 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.

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.

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. See the comments in the file `ssit_config.h` for more information.

3 The SimpleScalar Architecture Tool (SSAT)

SSIT can be used to synthesize new instructions and add new functional units as long as these new instructions process data contained in existing registers. In addition, it is not possible to use different names for existing registers. This would be useful for synthesizing MMX, for example, because the MMX registers correspond to the floating-point registers. The SimpleScalar Architecture Tool (SSAT) extends SSIT by providing ways to define new registers, to allow accessing a register through a different name, and to extend an existing register file.

New registers and register files, register aliases, and register file extensions are specified in the SSAT configuration file. A configuration file contains several directives. The possible directives are:

`NEW_REGTYPE <name>, <size>, <number>`. This directive creates a new register file with name `<name>`. Each register is `<size>` bytes wide, and there are `<number>` registers in the register file. In the assembly file, individual registers are addressed by appending their number to the name of the register file. E.g., if `name=newrf`, then the third register is addressed as `newrf2`. In the simulator each register is represented as an array of `<size>` unsigned bytes (`unsigned char[]`).

`ALIAS_REGTYPE <name>, <alignment>, <number>, <name source>`. This directive provides a different name for the existing register file with name `<name source>`. `<alignment>` indicates the number of existing registers that correspond to one register alias. For example, if `<alignment>=4`, then the first four existing registers correspond to the first register alias, the next four to the second register alias, and so on. `<number>` indicates the number of aliased registers. It should be less (TODO: or equal?) than the number of registers in the existing register file divided by `<alignment>`.

`EXTEND_REGTYPE <name>, <number>`. This directive extends the existing register file with name `<name>` with `<number>` registers.

`NEW_CTRL_REG <name>, <size>` This directive adds a new control register with name `<name>` and width `<size>` bytes.

An example configuration file is depicted in Figure 2. All text behind a sharp (`#`) symbol is comment. The first directive creates an MMX-like register file. MMX registers are 64 bits wide and correspond to the floating-point registers. The alignment is 2, indicating that two

```

1      # MMX-like register file aliased to floating-point register file
      ALIAS_REGTYPE mm, 2, 16, f

      # SSE-like register file
5      NEW_REGTYPE   xmm, 16, 32

```

Figure 2: Example SSAT configuration file.

32-bit single-precision floating-point registers correspond to one MMX-like register. The same could have been achieved with

```
ALIAS_REGTYPE mm, 1, 16, d
```

MMX supports only 8 registers, but here we have assumed 16.

SSE registers, on the other hand, are 128 bits wide. The second directive, `NEW_REGTYPE`, defines a set of 16 SSE-like registers named `xmm`.

SSAT not only defines new register files and provides different names for existing ones, it also defines macros to access the new register state, dependency designators, etc. These should be used when describing the semantics of new instructions. The predefined macros are (`XXX` stands for the new register file name):

`SSAT_XXX`, `SSAT_XXX(N)`. These macros read the new registers. If `XXX` is a single control register, then no number needs to be supplied. If `XXX` is a register file, then `N` indicates the register number. Both macros return an array of unsigned bytes.

`SET_SSAT_XXX(EXPR)`, `SET_SSAT_XXX(N,EXPR)`. These macros write the new registers. `EXPR` is an array of unsigned bytes that matches the size of the registers.

`DSSAT_XXX`, `DSSAT_XXX(N)`. Dependency designators, just as for example the dependency designator `DGPR(N)` defined in `sim-outorder.c`.

`SSAT_XXXS`, `SSAT_XXXD`, `SSAT_XXXT`. Register specifiers for the `XXX` register file. These specifiers are used to reference the source, destination or target `XXX` register. For example, when an instruction’s implementation wants to assign the destination `XXX` register, it uses `SET_SSAT_XXX(SSAT_XXXD)`, `array_name`. These specifiers are directly mapped to the SimpleScalar `RS`, `RD` and `RT` register specifiers, accordingly. Hence, the latter specifiers can be safely used instead of them.

As an example of the usage of these macros, Figure 3 depicts the SSAT configuration file for the MMX-like instruction `PADDB` (addition of packed bytes). We say “MMX-like” because most MMX instructions have two operands and the second operand can be an MMX register or a memory address. The SimpleScalar ISA is RISC and such instructions have three operands all of which are registers. The first section defines two new functional units: `SIMD-ALU` that executes instructions belonging to the `SimdALU` resource class, and `SIMD-MULT/DIV` that executes instructions from the `SimdMULT` resource class (with a latency of 3 cycles and a throughput of 1) and from the `SimdDIV` resource class (latency of 10, issue latency of 9 cycles). As can be seen, the implementation of `PADDB` uses the macro `SSAT_MM` to read the `mm` register file, the macros `SSAT_MMS`, `SSAT_MMT` and `SSAT_MMD` to reference the registers, and the macro `SET_SSAT_MM` to write the destination register. The definition of the new instruction uses the dependency designator `DSSAT_MM`.

```

1  %FUNCTIONAL_UNITS
   SIMD-ALU, 1,
     SimdALU, 1, 1

5  SIMD-MULT/DIV, 1,
     SimdMULT, 3, 1,
     SimdDIV, 10, 9

%MACHINE.DEF
10 #define PADDB_IMPL { \
     unsigned char[] _source1 = SSAT_MM(SSAT_MMS); \
     unsigned char[] _source2 = SSAT_MM(SSAT_MMT); \
     unsigned char[8] _result; \
     int _i; \
15     for (_i=0; _i<8; _i++) \
         _result[_i] = _source1[_i] + _source2[_i]; \
     SET_SSAT_MM(SSAT_MMD, _result); \
}
20 DEFINST( "paddb",          "d,s,t", "add",
     SimdALU,          F_ICOMP,
     DSSAT_MM(SSAT_MMD), DNA,
     DSSAT_MM(SSAT_MMS), DSSAT_MM(SSAT_MMT), DNA
)

```

Figure 3: SSIT configuration file to synthesize addition of packed bytes.

(TODO: Is this correct, can I use the integer instruction add even though MMX registers are aliases of integer instructions?)

4 Conclusions

In this work two tools facilitating the extension of the SimpleScalar instruction set and architecture are proposed. Provided with the configuration files, these tools help to automatically update the source code of the SimpleScalar simulators and to substitute the readable instructions in assembly sources with the SimpleScalar annotated instructions. After (re)compilation of SimpleScalar the simulators support the new instructions and architectural units, and the SimpleScalar GCC is able to compile the assembly source files for the simulators. The user only needs to provide a description of the new architectural units (registers and functional units) and a C implementation of the new instructions' functionality in the configuration files. For the latter, the user needs some basic knowledge of SimpleScalar.

Up to now SSIT and SSAT have been used in [5] to evaluate the performance of the Modified MMX (MMMMX) technology compared to the MMX. The MMX and MMMMX instruction sets were integrated into SimpleScalar. Performance simulation of several common multimedia kernels has been accomplished. In this work SSIT and SSAT played an essential role facilitating the integration of the new instruction sets and architectural units into SimpleScalar and processing the assembly files containing the kernels.

SSIT and SSAT are two separate tools for historical reasons, they were developed inde-

pendantly. In future it is planned to combine them into a single tool (the projected name is SSIAT, SimpleScalar Instruction and Architecture Tool).

Besides that, currently there is a certain inconvenience caused by the need to work with assembly files to utilize new instructions. In [5], the multimedia kernels were first compiled from the C language into assembly, then the appropriate parts were transformed by hand into the code containing the new (and existing) instructions, after that the sources were assembled into the executables. It would be considerably easier, for example, to introduce macros for the new instructions, so that C source code can be directly compiled into the executable form with annotated instructions. To make this possible, the SimpleScalar GCC must be updated.

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/>.
- [5] Asadollah Shahbahrami, Ben Juurlink, Demid Borodin, and Stamatis Vassiliadis. Avoiding Conversion and Rearrangement Overhead in SIMD Architectures. *International Journal of Parallel Programming*, pages 237–260, June 2006.