

SimpleScalar Instruction and Architecture Tool (SSIAT) Manual

Dusty Lefevre
dlefevre@student.ua.ac.be

March 31, 2006

1 Introduction

The SimpleScalar [?] tool set is a software suite build for performance analysis. The The Delft University of Technology (TU Delft) is implementing a tool chain for extending this suite and crosscompiling the extended assembler to form compatible with SimpleScalar. This tool chain includes the SimpleScalar Instruction Tool[?] (SSIT), for adding new instructions, and the SimpleScalar Architecture Tool[?] (SSAT), for adding and extending register files. More information on these tools can be found on [?]. SSIAT is a merge between these two tools and provides, apart from a unified interface, a backup mechanism and support for one configuration file.

2 Inner working

SSIT and SSAT are part from a tool chain and thus need to be executed in sequence. After making the necessary backups and restorations (more on this in the SimpleScalar section) SSIAT will perform these actions in two phases.

First, the SSIT code is executed which will update the SimpleScalar source and change the mnemonics in the assembler file, depending on which options were specified. Next the SSAT code will update the SimpleScalar source and process the assembler file if necessary. This again depends on the specified options. The SSAT code will only be executed when a configuration for the SSAT component was found or the *-a* or *-all* options were specified.

SSIAT uses a unified configuration file. The configuration uses the same syntax rules as the SSIT configuration with an additional “%REGISTERS” section for the SSAT configuration and a “%ALIASES” section in which aliases to instructions can be defined. Both the SSIT and SSAT component will read the completely read the configuration and each others sections.

3 The configuration

The configuration file is divided in four sections. A new section is identified by a “%” and a section identification (*REGISTER*, *FUNCTIONAL_UNITS*, *MACHINE.DEF* and

ALIASES). Each section may be defined multiple times and all instances will be accumulated. All text behind a “#” is considered comment, except in the “*MACHINE.DEF*” section, and will be ignored by the configuration parser.

3.1 %REGISTERS

This section contains the SSAT component configuration. In other words, this is where the directives to add, extend and alias registers should be put.

NEW_REGTYPE <name>, <size>, <instances>

Creates a new registry file with <instances> instances. The registers are named by appending the instance number behind <name> (i.e. for instance 0 with name 'f' the register would be named 'f0'). The registers consist of <size> bytes, and should always be a multiple of 8, smaller than 256.

ALIAS_REGTYPE <name>, <alignment>, <instances>, <source>

Creates an alternative way of accessing registers (<source>) through an alias (<name>). Each alias consists of <alignment> source registers.

EXTEND_REGTYPE <name>, <instances>

Extends an existing register file <name> with <instances> instances. The total number of instances should not exceed the 256 limit. In case of extending the SimpleScalar integer register file, the special registers are not moved, i.e. when extending from 32 to 64 instances, the special register \$31 is not moved to \$63!

NEW_CTRL_REG <name>, <size>

Extends the SimpleScalar control register file with a register of <size> bytes and with name <name>. The total number of registers should not exceed 256. Because \$hi, \$lo and \$fcc are already taken, 253 more can be added.

3.2 %FUNCTIONAL_UNITS

Functional units are used by sim-outorder to define the latency of instructions. The corresponding section contains directives that define these functional units which each can multiple resources. Such a directive has following signature:

```
<name>, <quantity>,  
    <class>, <operation latency>, <issue latency> [,  
    <class>, <operation latency>, <issue latency>][,  
    ...]
```

<name>: The name of the functional unit.

<quantity>: The number of instances.

<class>: The resource class.

<operation latency>: The number of cycles before the result is available.

<issue latency>: The number of cycles before another instruction can use this resource.

Apart from defining the latencies in the configuration file it is also possible to redefine them when SimpleScalar is started (by means of command line parameters). To create a command line parameter for a resource class, this class must be exported with

the `E_` prefix¹. This will create a command line parameter with the following syntax: `-lat:<class> <operation latency> <issue latency>`².

3.3 %MACHINE.DEF

This section is used for defining the new instructions. Each new instruction is specified by two parts. First, a implementation that is specified as a C macro. Next, an instruction definition. It is important that the `INSTRUCTION_NAME` part of `INSTRUCTION_NAME_IMPL` matches instruction name. The annotated instruction 's name should exist in `simplescalar` and a definition should always follow the appropriate implementation. The signature of a single instruction looks like this [?]:

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

`<instruction name>`: The name of the new instruction.

`<operands>`: Operands of the new instruction (see `machine.def` for more information).

`<annotated instructions name>`: The 'existing' SimpleScalar instruction that should be annotated.

`<FU class>`: The resource class needed for the execution. This can be a SimpleScalar resource or a resource defined in the `FUNCTIONAL_UNITS` section.

`<iflags>`: instruction flags (see `machine.def` for more information).

`<odep1>`, `<odep2>`: Output dependency designators. These define which operands will be modified and are used to define data dependencies (see `machine.def` for more information).

`<idep1>`, `<idep2>`, `<idep3>`: Input dependency designators. These define which operands will be read and are used to define data dependencies (see `machine.def` for more information).

Registers that were added, aliased or extended can be accessed in the implementations using special macros. The definitions of these macros are added by the SSAT component. In the following definitions the `XXX` should be replaced by the instruction name.

`DSSAT_XXX` and `DSSAT_XXX(N)`

For each new, alias and control register a decoder macro is defined. The `N` should be replaced with a specifier. Control register decoder have no specifier argument.

`SSAT_XXX` and `SSAT_XXX(N)`

This is an accessor macro for reading the content of the registers. The `N` should be replaced with a specifier (doesn't apply to control registers). Accessors return the content as an array of bytes (`unsigned char []`).

¹This new name, `E_` included, will have to used throughout the configuration file.

²The resource class does not contain the `E_` prefix.

SET_SSAT_XXX(Expr) and SET_SSAT_XXX(N, Expr)

This is an accessor macro for writing to a register. The *N* should be replaced with a specifier except when dealing with control registers. *Expr* should be an array of bytes (*unsigned char [I]*).

SSAT_XXXS, SSAT_XXXD and SSAT_XXXT

These are register specifiers for the *XXX* register file. The *S*, *D* and *T* stand for source, destination and target register.

More information of the *MACHINE.DEF* section can be found in Chapter 2.2 of [?].

3.4 %ALIASES

This section can be used to define aliases to instructions. These aliases can reshuffle the operands of existing instructions or assign constant literals to some of them. An alias is defined on one line using this format:

```
<alias> [<lnk>][, <lnk>]* = <instr> [<lnk>|<lit>]
                                     [, <lnk>|<lit>]*
```

<alias>: The new instruction or alias.

<instr>: The instruction that is to be aliased.

<lnk>: Define a linkage between operands. A link always begins with the “&” symbol.

<lit>: A literal (e.g. a number).

Aliases can be useful when defining pseudo-ops that are normally implemented by the compiler or assembler. An example of this are the 2 operand pseudo-ops that simplify the *cmpps* and *cmpps* SSE instructions. The *cmpeqps* alias can be defined like this:

```
cmpeqps &a, &b = cmpps &a, &b, 0
```

4 Usage

The configuration can be applied on an instance of SimpleScalar and assembler code by using the *ssiat* tool. By default SSIAT updates the assembler code, but this can be disabled by using the “-n” option. In order to update SimpleScalar use the “-u” option. As mentioned before, the SSAT component will be used if no *REGISTERS* section was found. This behavior can be overwritten with the “-a” option. Below is a list of all the available command line options.

```
usage: ssiat <options>
```

with options:

```
-i, --input <input assembly file>
The input assembly file that should be updated.
(mandatory unless the -n option is given).
```

```
-o, --output <output assembly file>
The output assembly file, where the updated version
should be saved.
(mandatory unless the -n option is given)
```

```

-c, --config <configuration file>
The configuration (mandatory)

-s, --sspath <simplescalar source dir>
The SimpleScalar 3.0 source path.
(optional, defaults to '.')

-u, --update
update SimpleScalar source.
(optional)

-n, --nocompile
don't update the assembler file.
(optional)

-b, --nobackup
disable backup routines.
(optional, not recommended)

-a, --all
force ssat phase
(optional)

-h, --help
display the help message.
(will block other options)

-v, --version
display version.
(will block other options)

```

5 SimpleScalar

SSIT and SSAT, and thus SSIAT, are designed to work on SimpleScalar version 3.0. Before SSIAT can make any changes to the SimpleScalar source code it will have to be configured for the Portable Instruction Set Architecture (PISA). This is done by typing “*make config-pisa*”. After this SSIAT will change, once executed with the “-u” option, the following files: *loader.c*, *machine.def*, *machine.h*, *machine.c*, *regs.h*, *resource.h* and *sim-outorder.c*. SSIAT will make backups from these files before any changes are made. These backups can be recognized by their *.backup* file extention.

Once the source has been updated it will have to be recompiled. Because only *sim-outorder* is supported it is best to use “*make sim-outorder*” instead of just “*make*” since in the latter case it will most likely not compile. After recompiling the SimpleScalar the *sim-outorder* binary is ready for use on the modified assembler.

6 Converting configurations

SSIT and SSAT use separate configuration files. However SSIAT uses a unified configuration file. It is easy to convert the former to this new format using the *convert-cfg.sh* Bourne again shell script.

```

usage: convert-cfg.sh -i <ssit config> -a <ssat config> \
      [-o <ssiata config>]

```

<*ssit config*>: The SSIT configuration file (mandatory)
<*ssat config*>: The SSAT configuration file (mandatory)
<*ssiat config*>: The output file (optional)

As described above, a *ssit* and *ssat* configuration file should always be given. However should no output file be specified then the output will be written to */dev/stdout*.

References

- [1] Juurlink B.H.H., Borodin D., Meeuws R.J., Aalbers G.Th., Leisink H. *The SimpleScalar Instruction Tool (SSIT) and the SimpleScalar Architecture Tool (SSAT)*
- [2] Juurlink B.H.H., Borodin D. *The SimpleScalar Instruction Tool (SSIT) Manual*
- [3] Meeuws R.J., Aalbers G.Th. *SSAT manual*
- [4] SimpleScalar LLC website <http://www.simplescalar.com>
- [5] SSIT, SSAT and SSIAT website <http://ce.et.tudelft.nl/~demid/SSIAT>