Dipartimento di Elettronica e Informazione

Politecnico di Milano

20133 Milano (Italia)
Piazza Leonardo da Vinci, 32
Tel. (39) 02-2399.3400
Fax (39) 02-2399.3411

LEON2/3 SystemC Simulator: User Manual

TRAP version 0.58 (revision 822)
LEON2/3 models version 0.3 (revision 822)

Contract Change Notice for ESA contract 20921/07/NL/JD

Contract carried out by Luca Fossati,
Politecnico di Milano (Italy): 2009-2010
European Space Agency/ESTEC: 2010-2012

TABLE OF CONTENTS

1 CD Content

The CD delivered to ESA, in the context of the contract for the developer of the simulators described in this document, has the following structure:

- `cross_compilers`: contains the compilers running both on linux and on windows operating systems, producing code for the *sparc* architecture. In case it is necessary to re-create such cross-compilers, or in case they have to be created for different systems (e.g. cygwin or MacOSX), instructions and scripts for producing them are contained in folder `cross_gcc_scripts.`
- `TRAP`: contains the sources of TRAP itself; this means all the Python files needed to generate the Instruction Set Simulators and the TRAP runtime library (both components are described more in detail in the rest of this document).
- `TRAP_runtime`: contains the pre-compiled TRAP runtime library; such libraries have been compiled for the linux operating system. Use the sources in folder `TRAP` and the instructions given in the rest of the document in case you need to create the libraries for other OSes.
- `LEON2_sim` and `LEON3_sim`, both of them with the same sub-folder structure:
    - `trap_sources`, containing the python files which describe the LEON2/3 architecture according to TRAP's conventions; together with the TRAP library, such sources can be used to re-create the C++ code implementing the Instruction Set Simulators.
    - `sources`, containing the C++ files implementing the simulators; there are different sub-folders for each simulator flavor (Instruction-Accurate, Cycle-Accurate, with Loosely-Timed or Approximate-Timed interfaces, etc., as described in the rest of the document).
    - `binaries`, containing the binaries (compiled for the linux OS) of the different simulator flavors.
- `docs`, containing this user manual and other documentation.

Other documents, the development version of TRAP, and more processor models can be found on the website http://trap-gen.google.com/ .

TRAP, and the scripts for the cross-compiler creation are licensed under the LGPL license of which we report here the main points:

TRAP is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this program; if not, write to the

Free Software Foundation, Inc.,
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA

or see <http://www.gnu.org/licenses/>.

For what concerns the LEON2/3 models, they are delivered to ESA under the following conditions:

The LEON2/3 SystemC models are owned by the Politecnico di Milano, and are delivered to ESA with a non-exclusive license that authorizes ESA to make use of them freely for any ESA desired purposes, without any restrictions. ESA is therefore free to use them internally and also distribute the models to ESA contractors or any

other third parties that ESA considers appropriate, in order to maximize the re-use of these models, improve them with users' feedback and facilitate R&D and commercial developments supported by the use of these models.

2 Transactional Automatic Processor Generator (TRAP)

TRAP (*TR*ansactional *A*utomatic *P*rocessor generator) is a tool for the automatic generation of processor simulators starting from high level descriptions. This means that the developer only needs to provide basic structural information (i.e. the number of registers, the endianness etc.) and the behavior of each instruction of the processor ISA; these data are then used for the generation of `C++` code emulating the processor behavior. Such an approach consistently eases the developer's work (with respect to manual coding of the simulator) both because it requires only the specification of the necessary details and because it forces a separation of the processor behavior from its structure. The tool is written in Python and it produces SystemC based simulators.
TRAP requires in input the information about the behavior of the target processor, and also some (limited) details about the structure in terms of architectural elements and their connectivity.

The tool consists of a Python library: the processor specification is given through appropriate calls to its APIs. The Instruction Set Simulators generated by TRAP are based on the SystemC library and on the new TLM 2.0 standard for modeling the processor's communication interfaces. Depending on the desired temporal accuracy/simulation speed tradeoff, different flavors of simulators can be created.

2.1 Setup

TRAP uses waf (http://waf.googlecode.com/) as build system; from a user point of view, it is similar to using the standard autotools (i.e, Makefiles, etc.): first there is a configuration step, then the compilation and, finally, the installation.

2.1.1 Required Libraries and Tools

- **libELF**: There are various versions of the libELF library, normally you can find it among the packages of your distribution (e.g. *libelf-dev* under Ubuntu) or, else, download it from http://www.mr511.de/software/english.html (at least version 0.8.13) or from https://fedorahosted.org/releases/e/l/elfutils/ as part of the elfutil package (at least version 0.147).
- **SystemC**: downloadable from http://www.systemc.org, it is used to manage simulated time and the communication among the hardware modules. Note that in the current version of SystemC (2.2.0) there is a small bug which prevents compilation with compiler GCC 4.3 or newer: refer to http://www.timfanelli.com/item/2302 for more details. SystemC 2.3 beta release is also supported.
- **TLM 2.0**: downloadable from http://www.systemc.org, it is used to manage connections among the hardware modules. This library is not necessary for the compilation of TRAP itself, but for the compilation of the generated processor models.
- **Boost**: Boost libraries version >= 1.35, downloadable from http://www.boost.org. As, usual, the such libraries are also shipped with Linux distributions, try first to use the package manager of your system (e.g. *apt* for Ubuntu) for installing them; the development package is needed. While it is theoretically possible to have more than one version of the boost libraries installed on the system, it is better to keep only one in system-wide accessible locations (e.g., in `/usr/lib`).
- **Python**: version 2.4.x - 2.7.x

- **networkx**: library used for the description of graphs in Python. This library can be downloaded from http://networkx.lanl.gov/ and it is used during the generation of C++ code implementing the processor models, but it is not necessary if only the runtime library of TRAP is needed. Version 0.36 or later can be used.

2.1.2 Configuration

In the base folder of TRAP run the `./waf configure` command. Several configuration options are available:

- --with-systemc=SYSC_FOLDER specifies the location of the SystemC 2.2 library (which, of course, should have been already compiled). It is mandatory to specify such option.
- --prefix=LIB_DEST_FOLDER specifies the destination folder for installation of the TRAP C++ runtime libraries. If this option is not specified `/usr/local` is used as default
- --py-install-dir=PYTHON_DEST_FOLDER specifies the destination folder for the installation of TRAP python files. If this option is not specified, PREFIX/pythonX.X/site-packages/ is used as default (where PREFIX is the value of the preceding option).
- --with-elf=LIBLEF_FOLDER specifies the location of the libELF library; if this option is not specified TRAP will look for libelf in the standard library search path.
- --boost-includes=BOOSTINCLUDES specifies the location of the include files of the boost lbraries; it has to be specified in case the libraries are not installed in a standard path.
- --boost-libs=BOOSTLIBS specifies the location of the library files of the boost libraries; it has to be specified in case the libraries are not installed in a standard path.

To have the list of all the possible configuration options run `./waf --help`
For example, supposing the *boost* and *binutils* libraries are installed in the default library search path, type:

```
./waf configure --with-systemc=$HOME/systemc-2.2.0 --prefix=$HOME/trap
```

in order to configure TRAP for being installed in the user's home folder.

2.1.3 Compilation and Installation

Once the project is correctly configured execute the `./waf` command to trigger the compilation. Once this is done, command `./waf install` copies the python files and the generated runtime libraries in the chosen destination folders (previously specified through the --prefix option).

2.2 Runtime Library

The C++ code composing the generated simulators is partly based on auto-generated code and partly from TRAP's runtime library. Its source code is contained in folder *trap/runtime*; in particular we can identify the following parts:

- **debugger**: contains the GDB server used for the communication with the GDB debugger of your target architecture (i.e. in case we are simulating a LEON3 processor with the *sparc-elf-gdb* debugger).
- **osEmulator**: enables the execution of software applications (on top of the Instruction Set Simulator) without the need to also execute an operating system; calls made to the OS (e.g. for writing to *stdout*) will be sent to the emulator and, from this, forwarded to the host OS.
- **elfFrontend**: C++ wrapper around the libELF library used to parse the application being executed with the generated simulator. The elfFrontend library is used by the *loader*, the

osEmulator, and the *profiler*. In synthesis, it provides methods for reading all the symbols in the application, for determining their correspondence with the addresses in the application executable file, and for parsing the different sections of the application to extract the machine code, the static data, etc.

- **loader**: given the application executable file, it extracts the text segment (i.e. the assembly instructions which implement the application behavior), the data segment (containing the static data, global variables, etc.), it determines the entry point (the first instruction to be executed) and all the other information necessary for application execution.
- **profiler**: computes statistics about the application program: the number of times each routine is called and the time spent into it, and the number of times each single assembly instruction is called and the time spent in its execution.

When compiling TRAP, all the just mentioned tools are linked together in a runtime library (called *libtrap*) provided both in the form of static (*libtrap.a*) and dynamic library (*libtrap.so*); during TRAP installation such libraries are copied into folder PREFIX/lib, while the corresponding header files into PREFIX/include. Note that, in a 64 bit environment, dynamic linking is possible only if all files are compiled using the `-fPIC -DPIC` compilation flags; so, in case SystemC and/or libELF have not been created with such flags (as it is the default case for SystemC 2.2, not anymore for SystemC 2.3) the dynamic TRAP library will not be created.

2.2.1 Operating System Emulator

Operating System Emulation is a technique which allows the execution of application programs on an Instruction Set Simulator (ISS) without the need to also simulate a complete OS. The low level calls made by the application to the OS routines (*system calls*, SC) are identified and intercepted by the ISS, and then redirected to the host environment which takes care of their actual execution. Suppose, for example, that the application program, that we want to execute on top of the simulated architecture, contains a call to the *open* routine to open file ``*filename*''. Such a call is identified by the ISS and routed to the host OS, which actually opens ``*filename*'' on the PC's filesystem. The file handle is then passed back to the simulated environment for the use by the application program.

Having an Instruction Set Simulator with Operating System Emulation capabilities allows the application developers to start working as early as possible, even before a definite choice about the target OS is performed. These capabilities are also used for ISS validation, by enabling fast benchmark execution.

Operating System emulation is enabled by default in TRAP's created simulators, as the OSEmulator tool (see below for more details) is always added to the processor's generated code; note, anyway, that the emulator is not intrusive in the ISS core, and it can be disabled by commenting the following line in the *main.cpp* file of the generated processor:

```
01. procInst.toolManager.addTool(osEmu);
```

In addition to simply emulating system calls, the OS emulator fakes a linux-like environment for program execution; such environment can be specified through the following command line options of the generated ISS (for more details look inside the *main.cpp* generated file and read Section 3.4):

- `--arguments:` comma separated list of the simulated application arguments (which are passed to the `main` routine of the simulated application); note that, even if no argument is specified by the user, the name of the application program is always passed to the `main` routine of the simulated application as first parameter.
- `--environment:` comma separated list of the environmental variables that we want to make visible to the application program; they are in the form *option=value,option=value*. From the application program, such variables can be read by calling the `getenv` routine.

- `-- sysconf:` comma separated list of the environmental variables that we want to make visible to the application program; they are in the form *option=value,option=value*. From the application program, such variables can be read by calling the `sysconf` routine.

2.2.2 GDB Debugger

A debugger is a tool used for the analysis of software programs in order to catch and, possibly, help correcting errors. Being used for software development, Instruction Set Simulators often feature a debugger; TRAP was integrated with the GNU/GDB debugger using its capabilities of connecting to remote targets through TCP/IP or serial interfaces (the same mechanisms used to debug software running on physical boards). The specifications on the GDB remote protocol are present at http://sourceware.org/gdb/current/onlinedocs/gdb_33.html According to this protocol a TCP/IP server has been implemented inside TRAP; by default it listens for connections on *port 1500*; from GDB the directive for connecting to such remote target is: `target remote localhost:1500.`
The standard GDB commands (including all the ones mandatory for being compliant with the GDB protocol) are supported. In addition, through the *monitor* command, we add functionalities to GDB in order to manage the flow of simulated time (of course these commands can only be used in a simulator created with SystemC enabled, which has the notion of time):

- `go` *n*: specifies that, starting from the current time, simulation has to proceed for *n* nanoseconds; this command only sets the length of the simulation time, but simulation is not resumed until the `cont` command is issued.
- `go_abs` *n*: specifies that simulation has to proceed until time *n* (in nanoseconds); this command only sets the length of the simulation time, but simulation is not resumed until the `cont` command is issued.
- `hist n`: prints the history of the last *n* executed instructions, up to a maximum of *n = 1000*.
- `status`: returns the status of the simulation, i.e. the elapsed simulation time and if it is running for an indefinite period of time or if simulation is running only for a specified amount of time, the latter situation happening after a `go` or `go_abs` command has been issued.
- `time`: returns the current simulation time in microseconds.
- `help`: prints the list, with a short explanation, of the just listed commands.

From the GDB console, the syntax for issuing such commands is "`monitor` *`command`*" (e.g. `monitor help`).

When the simulator is started, the `--debugger` command line option activates the use of GDB; refer to the next Chapter for more details and for a small tutorial on the use of GDB together with the Instruction Set Simulators.

2.2.3 Application Loader

The application loader is a simple piece of software which takes an executable file (usually in the ELF format, but not limited to it), it parses such file and it extracts the different sections of the executable program (the code, the data, and it determines the entry point). The loader is not intrusive at all in the Instruction Set Simulator core as it is instantiated in the *main.cpp* file and simply used to initialize the memory with the code and data of the application program to be simulated and to initialize the processor with the size of the whole application program and with the value of the entry point. The following snippet of code performs such actions:

```
01. ……
02. ExecLoader loader(application_name);
03. //Lets copy the binary code into memory
04. unsigned char * programData = loader.getProgData();
05. for(unsigned int i = 0; i < loader.getProgDim(); i++){
```

```
06.     procInst.dataMem.write_byte_dbg(loader.getDataStart() + i,
programData[i]);
07. }
08. procInst.ENTRY_POINT = loader.getProgStart();
09. procInst.PROGRAM_LIMIT = loader.getProgDim() + loader.getDataStart();
10. procInst.PROGRAM_START = loader.getDataStart();
11. ……
```

The `--application` command line option of the generated simulator is used to specify the software application which has to be simulated.

2.2.4 Profiler

The profiler is used to compute and produce statistics on the software application being executed on the simulator; in particular it produces statistics about single assembly instructions and function calls:

- *assembly instructions*: for each single assembly instruction as defined in the Instruction Set Simulator (note that not necessarily there is a one-to-one correspondence between the instructions in the processor manual and the instructions in the simulator) it computes the *number of calls*, the *percentage of the number of calls* on the total number of instructions executed, the *total SystemC time spent* in executing this instruction, and the *SystemC time per call*.
- *routines*: for each routine of the application program, the following information is computed: the *number of calls*, the *percentage of the number of calls* on the total number of routines executed, the *number of assembly instructions* executed inside this routine and the subroutines called from it, the *number of assembly instructions* executed exclusively inside this routine (not considering sub-routines), the *number of assembly instructions* per call, the *SystemC time* spent inside this routine and the subroutines called from it, and the *SystemC time* spent exclusively inside this routine (not considering sub-routines).

The profiler is enabled with the `–profiler` ***`file`*** command line option of the generated simulator; ***`file`*** represents the name of the file in which the profiler output is saved; in particular two files are produced: ***`file`***`_instr.csv` and ***`file`***`_fun.csv`. Such files are in the CSV (comma separated value) format and they have the following structure:

- ***`file`***`_instr.csv`: *name;numCalls;numCalls %;time;Time per call*, where the meaning of the different fields is explained above; here is an example of such file:

```
name;numCalls;numCalls %;time;Time per call
ORcc_reg;1;0.0254000508001016;0;0
LDSH_imm;16;0.4064008128016256;0;0
SMUL_reg;1;0.0254000508001016;0;0
FLUSH_imm;1;0.0254000508001016;0;0
LDUH_imm;13;0.33020066040132079;0;0
LD_imm;57;1.4478028956057911;0;0
LD_reg;40;1.0160020320040639;0;0
```

- ***`file`***`_fun.csv`: *name;numCalls;numCalls %;totalNumInstr;exclNumInstr;NumInstr per call;totalTime;exclTime;Time per call*, where the meaning of the different fields is explained above; here is an example of such file:

```
name;numCalls;numCalls %;totalNumInstr;exclNumInstr;
              NumInstr per call;totalTime;exclTime;Time per call
memset;3;4.225352112676056;156;156;52;0;0;0
_fwrite_r;1;1.408450704225352;386;35;35;0;0;0
__do_global_ctors_aux;1;1.408450704225352;7;7;7;0;0;0
software_init_hook;1;1.408450704225352;129;38;38;0;0;0
f2;1;1.408450704225352;2512;53;53;0;0;0
f5;1;1.408450704225352;2285;87;87;0;0;0
f8;1;1.408450704225352;2024;87;87;0;0;0
```

Note how the different elements of such files are separated by a semi-colon **';'**.
There are two command line options which affect the way profiling is performed:

- `-prof_range` *`start-end`*: computes the profiling statistics only among the assembly instructions contained at addresses *start* and *end*; such addresses can be both specified as decimal of hexadecimal numbers or also giving the name of the symbol they correspond to (e.g. the *main* routine).
- `-disable_fun_prof:` it disables statistics gathering on software routines: the only statistics that the profiler computes are the ones on the single assembly instructions. Using this options consistently accelerates execution speed with respect to a fully fledge profiling; moreover, in a few corner situations it might happen that the profiler (and, hence, the whole simulation) fails because of problems in tracking function calls: using this option prevents such failures from happening.

2.2.5 Creating New Tools

A Tool, in TRAP, is a software element which needs to be called during the simulation and which performs actions in correspondence of particular values of the program counter. Examples of Tools are the OSEmulator (in correspondence of the program counter identifying an operating system call, it forwards the call to the host OS), the Debugger (in correspondence of the program counter identifying a break-point it pauses execution), and the profiler (for each different value of the program counter it updates the ISA statistics, while for the PC values corresponding to the entry/exit of a routine it updates the related routine statistics).

Other tools, depending on the user needs, might be easily created by extending the `ToolsIf` class contained in the *ToolsIf.hpp* file (part of TRAP's runtime library):

- Method `bool newIssue(const issueWidth &curPC, const InstructionBase *curInstr)` takes in input the current program counter and a point to the corresponding instruction of the processor Instruction Set; inside this method the tool can then take the appropriate actions depending on the tool itself. `true` is returned in case the current instruction have to be skipped by the processor (i.e. not executed, as it is the case of OS routines for the OSEmulator tool), `false` otherwise.
- Method `bool emptyPipeline(const issueWidth &curPC)` is used only inside the cycle-accurate processor and it specifies whether, before calling the `newIssue` method of the tools, the pipeline has to be emptied: for example, for the LEON3 processor, this method is checked in the fetch stage; if it returns `true`, the current instruction (the one in the fetch stage) is not propagated, but a series of 6 `nop`s is propagated instead in the pipeline. Then the `newIssue` method is called: since there has been the propagation of the 6 `nop`s, now the pipeline is empty.

More advanced tools (such as the debugger) may also need to perform actions when particular memory addresses are read/or written: such tools need to extend the `MemoryToolsIf` class contained in the *ToolsIf.hpp* file:

- Method `void notifyAddress(addressType address, unsigned int size)` is called in correspondence of every memory write; the tool can then take appropriate actions depending on the address to be written and on the size (in bytes) of the datum to be written. For instance, the debugger uses this method to check for watchpoints.

Once a tool has been written, it can be *inserted into* the processor by calling method `procInst.toolManager.addTool(toolInst)`, where `procInst` is the instance of the processor and `toolInst` is the instance of the tool to be added to the processor.

2.2.6 Communication Between the Processor and Tools

Of course, in order to be able to perform useful work, the tools have to be able to access and modify the processor status (its registers, the memory, etc.): this is provided by the ABI Interface, which can be accessed through there `getInterface()` method of the Processor class.

2.2.7 Miscellaneous Runtime-Library Elements

The runtime library also contains some miscellaneous elements not used by the simulator core itself, but which are anyway helpful for the development:

- `MemoryAT.hpp` and `MemoryLT.hpp`: SystemC IP-models representing generic memories with respectively Accurate-Timed and Loosely-Timed interfaces; such IP-models are used to check the correct working of the generated simulators which make use of the TLM interfaces for connecting with memories
- `PINTarget.hpp`: used to connect with the TLM PIN ports of the generated simulators for testing them (all the TLM ports which are not used for interrupts and for memory communication are called PINs). In the LEON models such ports are used for interrupt acknowledgement.
- `irqTester`: it is a folder containing compiler scripts, a TLM interrupt generator, and simple applicative software used to check that the generated LEON2 and LEON3 processor simulators correctly behave when interrupts are encountered.
- `benchmarks` and `testsuite`: containing real-world benchmark programs and test programs used to check the correct behavior of the generated simulators; instructions on how to use such programs are contained in those folders.

2.3 Simulator Generation

Once TRAP itself is correctly installed, simply execute the main python script of your processor model to generate the C++ code implementing the simulator. For the LEON3 processor, for example, go into the folder containing its source files (LEON3Arch.py LEON3Coding.py LEON3Isa.py LEON3Methods.py LEON3Tests.py LEONDefs.py) and run the command `python LEON3Arch.py`. The C++ files implementing the different flavors of simulators should be generated. Note that, in case the TRAP python files are installed in a custom folder and not in the Python default search path, the first lines of the main script (LEON3Arch.py in this case) should be modified to specify this path.

There are different commands and options which can be specified in the LEON3Arch.py file to customize the generated simulator; such options are specified in two places: in the constructor of the *processor* class and when calling the *write* method to start the actual processor creation.
The main architectural file (LEON3Arch.py, for example) is indeed organized as follows:

```
01. ……
02. processor = trap.Processor('LEON3', version = '0.2.0', systemc = True,
instructionCache = True, cacheLimit = 256)
03. ……
04. ……
05. processor.write(folder = 'processor', models = ['accLT', 'funcAT', 'accAT',
'funcLT'], dumpDecoderName = 'decoder.dot', trace = False, combinedTrace = False,
forceDecoderCreation = False, tests = True, memPenaltyFactor = 4)
06. ……
```

- **Line 02** contains the processor constructor:
  - The first parameter specifies the name of the processor being generated.

- The second parameter is a string specifying its version.
- The third parameter specifies whether SystemC should be used or not for keeping track of time: note that it is possible to avoid using SystemC (thus consistently speeding up simulation) with the standalone Instruction-Accurate model (called later *funcLT*) and when no TLM ports are specified.
- The fourth parameter (`instructionCache`) specifies whether the instruction buffer should be used or not: generally it should be used as it delivers high simulation speed without introducing any drawback in the generated models.
- The fifth parameter (`cacheLimit`) specifies the threshold with which instructions are added to the instruction buffer: only after an instruction has been encountered `cacheLimit` times it is added to the buffer. The value of this parameter is a tradeoff among having many instructions in the buffer (thus avoiding to decode them every time) and the effort due to adding the instructions to the buffer and searching for instructions in a huge buffer.
- Other less important parameters can be used; refer to the source code for more details.

- **Line 05** contains the `write` method; this method is the one actually triggering the creation of the C++ code implementing the simulator:
  - `folder` parameter: specifies the folder, relative to the current one, in which the simulator's C++ code will be created.
  - `models` parameter: the models which will be created; four different types of models can be created: *funcLT, funcAT, accLT, accAT*; they respectively represent the Instruction-Accurate with loosely-timed TLM interfaces, the Instruction-Accurate with accurate-timed TLM interfaces, the Cycle-Accurate with loosely-timed TLM interfaces and the Cycle-Accurate with accurate-timed interfaces. Note that the Instruction-Accurate models can also be created standalone (so not able to connect with any external IP), by specifying an internal memory instead of TLM port(s).
  - `dumpDecoderName` specifies the file in which the tree representing the decoder is saved using the dot format; this is useful just for debugging purposes, if no name is specified, no file is printed.
  - `trace` specifies that the processor is created with tracing capabilities: after the execution of each instruction (or at each clock cycle for the cycle accurate processor), a dump of the whole processor status is printed on standard error.
  - `combinedTrace`, if used in combination with `trace`, enables the creation of the same structure for the dump of both the functional and cycle accurate processors (by default these processors have a different dump format): this can be used to ease the development of the Cycle-Accurate processor, if an Instruction-Accurate version already exists.
  - `forceDecoderCreation`: as generating the decoder is an expensive operation, it is usually executed only once and saved in cache; when this option is set to true, the cache is discarded and the decoder re-created.
  - `tests:` when set to True, it enables the creation of the executable running the tests for the single instructions; note that such tests are created only when SystemC is not used, so they can only be created for the *funcLT standalone simulator*.
  - `memPenaltyFactor:` this parameter affects the way the decoder is created: higher values give preferences to `if` structures in the decoder, lower values to `switch/case` statements.

Other minor options can influence the generated code, as shown in the following snippet of code again taken from file LEON3Arch.py:

```
01. ……
02. processor.addTLMPort('instrMem', fetch = True)
```

```
03. processor.addTLMPort('dataMem')
04. processor.setMemory('dataMem', 10*1024*1024)
05. processor.setMemory('dataMem', 10*1024*1024, debug = True, programCounter =
'PC')
06. ……
```

Lines 02-03/04/05 are exclusive with each other, and they determine how the processor connects to memory:

- Lines 02-03 specify that the generated simulator will instantiate two TLM ports, with the first one being the one from which instructions are fetched (two memory ports are declared as the LEON3 processor features a Harward architecture).
- Line 04 declares an internal memory: both instructions and data will be fetched from this memory; the use of an internal memory is only suitable for standalone simulators, not connected with any other SystemC IP.
- Line 05 declares a debugging memory: it is an internal memory which also tracks each location written and the time at which each write operation happened; in case the fourth parameter is specified (containing the name of the register representing the program counter), the value of the program counter in correspondence of each memory write is saved. The resulting memory dump is saved in a binary file always called *memoryDump.dmp*; such file can be parsed with the *memAnalyzer* program created during the compilation of TRAP and contained, after TRAP's installation, in folder PREFIX/bin.

Next Chapter shows how to compile and use the simulator generated with the just seen commands; a brief overview of how it is possible to connect the simulator with other, externally provided, SystemC IP-models is also given.

3 LEON2/3 Simulator

This Chapter gives an overview of the LEON2/3 instruction set simulators generated with TRAP and it provides detailed instructions, also with simple tutorials, on how to use them. This Chapter is based on the LEON3 processor simulator, but the same considerations also apply to the LEON2 processor simulator.

3.1 Overall Structure

According to the instructions given in the previous Chapter, simulators are created with the following command:

```
python LEON3Arch.py
```

where *LEON3Arch.py* is the main file of the processor description (the one containing the call to the `write` method). Supposing that all the different flavors of simulators are created inside folder *processor* (which means that the call to the write method is `write(folder = 'processor', models = ['accLT', 'funcAT', 'accAT', 'funcLT'])`), the following folder structure is created:

```
processor
    accLT
    funcAT
    accAT
    funcLT
```

Such folders directly contain the C++ source code and the compilation scripts for the Instruction Set Simulators. In case the `funcLT` model has been created without SystemC and with the tests enabled, there will also be a sub-folder containing the C++ sources of the testsuite (for testing the decoder, the indivisual instructions, the interrupt behavior, etc.).

The different folders *funcLT, funcAT, accLT, accAT* respectively represent the Instruction-Accurate with loosely-timed TLM interfaces, the Instruction-Accurate with accurate-timed TLM interfaces, the Cycle-Accurate with loosely-timed TLM interfaces and the Cycle-Accurate with accurate-timed interfaces.

3.2 Compiling the Simulator

TRAP uses `waf` (http://waf.googlecode.com/) as build system; from a user point of view it is similar to using the standard autotools (e.g. Makefiles, etc.): first there is a configuration step, then the compilation and, finally, the installation.

3.2.1 Required Libraries and Tools

- **libELF**: There are various versions of the libELF library, normally you can find it among the packages of your distribution (e.g. *libelf-dev* under Ubuntu) or, else, download it from http://www.mr511.de/software/english.html (at least version 0.8.13) or from https://fedorahosted.org/releases/e/l/elfutils/ as part of the elfutil package (at least version 0.147).
- **SystemC**: downloadable from http://www.systemc.org, it is used to manage simulated time and the communication among the hardware modules. Note that in the current version of SystemC (2.2.0) there is a small bug which prevents compilation with compiler GCC 4.3 or newer: refer to http://www.timfanelli.com/item/2302 for more details. SystemC 2.3 beta is also supported.
- **TLM 2.0**: downloadable from http://www.systemc.org, it is used to manage connections among the hardware modules.
- **Boost**: Boost libraries version >= 1.35, downloadable from http://www.boost.org. As, usually, the such libraries are also shipped with Linux distributions, try first to use the package manager of your system (e.g. *apt* for Ubuntu) for installing them; the development package is needed.

3.2.2 Configuration

In the base folder of the simulator (folder `processor`, according to the example above) run the `./waf configure` command. Several configuration options are available:

- --with-systemc=SYSC_FOLDER specifies the location of the SystemC 2.2 library (which, of course, should have been already compiled). It is mandatory to specify such option.
- --with-tlm=TLM_FOLDER specifies the location of the TLM 2.0 library. It is compulsory to specify such option.
- --with-trap=TRAP_FOLDER specifies the location of the TRAP runtime libraries and headers. It must be specified if TRAP was installed in a custom folder (so not in the standard gcc library search path)
- --boost-includes=BOOSTINCLUDES specifies the location of the include files of the boost lbraries; it has to be specified in case the libraries are not installed in a standard path.
- --boost-libs=BOOSTLIBS specifies the location of the library files of the boost libraries; it has to be specified in case the libraries are not installed in a standard path.
- --with-elf=LIBLEF_FOLDER specifies the location of the libELF library; if this option is not specified TRAP will look for libelf in the standard library search path.

- --static specifies that a fully static executable have to be built. Such executable can be redistributed standalone and, once created, it does not require any library to be presented on the system on which is going to be executed.
- --enable-history activates the compilation of the data structures allowing to keep track of the instructions executed by the processor model.

To have the list of all the possible configuration options run `./waf –help.` Note that other compilation options can be present depending on the processor model being compiled; it is, indeed, possible to specify custom compilation options from the main processor description file (*LEON3Arch.py* for instance) through the `processor.setPreProcMacro(option_name, macro_name):` it adds a compilation option called *option_name* which, when triggered, has the effect of defininf macro *macro_name*. Such macro can then be used in the ISA behavior.
So far LEON2 and LEON3 models contain the `--tsim-comp` option, which controls the definition of macro `TSIM_COMPATIBILITY`; such macro activates controls, initializations, etc. used to make the simulator behave like Aeroflex Gaisler's TSIM.

For example, supposing the *boost* and *binutils* libraries are installed in the default search path, type:

```
./waf configure --with-systemc=$HOME/systemc-2.2.0 --with-tlm=$HOME/TLM-2009-07-15 --with-trap=$HOME/trap
```

in order to configure the processor.

3.2.3 Compilation

After the configuration successfully ends, compilation can be started with the `./waf` command; the executables files of the processor simulator an of its testing scripts (in case test generation was enabled) are contained inside the `_build_/default` folder.

3.2.4 Running the Tests

The tests of the decoder, of the instruction set, and of the interrupts (if they are declared) are located into folder `funcLT/tests` of the generated code; note that the tests are created only for the functional simulator and if SystemC is not used in the generated simulator. The tests are based on the *boost.test* library and they are composed of four main files:

- *main.cpp*: main file driving the tests; it simply instantiates the tests and runs them.
- *isaTests.cpp*: tests of the instruction set behavior; it contains all the tests specified by the developer to test the correct behavior of each instruction of the instruction set; as there tend to be many tests, this file has been split into multiple parts.
- *decoderTests.cpp*: tests of the decoder; at processor creation we pick up random bit strings exercising each instruction (i.e. the variable parts of each instruction are randomly chosen, the parts that identify each instruction of course are not) and we verify that the decoder is able to correctly decode them.
- *irqTests.cpp*: tests checking the correct behavior of the interrupt; they mimic the arrival of an interrupt and check that the processor correctly responds.

Running the tests is simple: just run the `tests` executable found under the `_build_/default/funcLT/tests` folder.

3.3 Using the Simulator

The executable file implementing the functional simulator has the following command line parameters:

- `--help`: prints the available command line parameters together with a brief explanation on how to use them.
- `--debugger`: enables the debugger for the debugging of the applications running on the simulator; the debugger simply consists of a GDB stub. In order to use it open GDB (of course the one relative to the architecture being simulated, if we are simulating a sparc-based processor, we need to open the GDB debugger for that architecture), open the application being simulated and connect to the GDB stub as a remote host using GDB command `target remote localhost:1500` (since the GDB Stub is waiting for connections on port 1500). Then you can normally debug the application as you would do using a standard GDB debugger; only note that, as normally happens when connecting to a remote target, you start execution with the `cont` command and not with `run`.
- `--application` *`arg`*: application which has to be simulated on the simulator; of course this application has to be compiled with a compiler for the architecture being simulated. Some cross-compilers are provided at page http://home.dei.polimi.it/fossati/downloads.html and on the delivered CD; note when using these cross-compilers, the `-specs=osemu.specs` command line option have to be passed to GCC in order to enable the use of the Operating System emulator tool.
- `--frequency` *`arg`*: this option is available only if the generated processor model is based on SystemC and it specifies the processor clock frequency; the frequency is expressed in MHz, the default value is 1 MHz.
- `--profiler` *`arg`*: activates the use of the software profiler, specifying, as argument the name of the output file; two files are created: *`arg`*`_instr.csv` and *`arg`*`_fun.csv.`
- `--disassembler`: prints to standard output the disassembly of the application; it works in a similar way to the `objdump -d` program (part of the binutils tools).
- `--arguments` *`arg`*: comma separated list of the simulated application arguments (which are passed to the `main` routine of the simulated application); note that, even if no argument is specified by the user, the name of the application program is always passed to the `main` routine of the simulated application as first parameter.
- `--environment` *`arg`*: comma separated list of the environmental variables that we want to make visible to the application program; they are in the form *option=value,option=value*. From the application program, such variables can be read by calling the `getenv` routine.
- `--sysconf` *`arg`*: comma separated list of the environmental variables that we want to make visible to the application program; they are in the form *option=value,option=value*. From the application program, such variables can be read by calling the `sysconf` routine.
- `--prof_range` *`arg`*: specifies, in the form *start-end*, the addresses among which profiling is enabled; such addresses can be represented by decimal or hexadecimal numbers or by the name of the symbol corresponding to the address (be it a function name of a variable).
- `--cycles_range` *`arg`*: specifies, in the form *start-end*, the addresses among which the count of the elapsed simulation cycles if performed; such addresses can be represented by decimal or hexadecimal numbers or by the name of the symbol corresponding to the address (be it a function name of a variable).
- `--history` *`arg`*: saves the history of all the executed instructions on the file specified in *arg*; note that this option is not available if the model has been compiled without instruction history (i.e., if the model has been compiled without the *--enable-history* option)
- `--disable_fun_prof`: it disables statistics gathering on software routines: the only statistics that the profiler computes are the ones on the single assembly instructions. Using this options consistently accelerates execution speed with respect to a fully fledge profiling; moreover, in a few corner situations it might happen that the profiler (and, hence, the whole simulation) fails because of problems in tracking function calls: using this option prevents such failures from happening.

For examples on how to use the different features of the simulator, refer to Section 3.4.

3.4 Tutorials

This Section aims at guiding you step by step in the cross-compilation of an application program, and in running it with the LEON instruction set simulators, fully exploiting their capabilities. In the following we will use the LEON3 simulator, but the tutorial also applies to the LEON2 simulator.

3.4.1 Cross-Compiling

Cross-compiling is the first step in running an application program on the generated instruction set simulators (actually it is also the first step when running any piece of code on a board); it consists in taking the source code of your application, *passing* it to the GCC cross-compiler which, in turn, creates the executable code which can be executed on the simulator. This step is very similar to the standard use of GCC for compiling programs, with the only difference that normally both the compilation and the execution steps are performed on the same machine, while now GCC runs on one machine (your PC), but it produces code for another machine (the LEON2/3 simulator). From this heterogeneity comes the name *cross*-compilation.

So, let's start with a simple program (which we save in file *test.c*):

```
01. #include <stdlib.h>
02. #include <stdio.h>
03.
04. void foo1(){
05.     printf("inside foo1\n");
06. }
07.
08. void foo2(){
09.     printf("calling foo1\n");
10.     foo1();
11.     printf("called foo1\n");
12. }
13.
14. int main(int argc, char * argv[]){
15.     foo2();
16.     foo1();
17.     return 0;
18. }
19.
```

Now, get the cross-compiler for sparc architectures: go into folder `cross_compilers` of the CD and de-compress file *sparc-elf-4.3.3.tar.bz2* in **any location** on you filesystem. Supposing you have it in folder */home/fossati/*, in order to cross-compile the previous program simply issue the command:

```
/home/fossati/sparc/bin/sparc-elf-gcc -o test –g –specs=osemu.specs test.c
```

The syntax is very similar to the normal use of GCC (`-g` is the normal flag to enable the production of debugging information and `–o` specifies the output file name); the only difference is the `–specs=osemu.specs` flag: it specifies that the BSP (Board Support Package) for the Operating System Emulation has to be used; as such, in case you are not using emulation but your own operating system, such flag does not have to be employed. The source files, together with some documentation, of the *osemu* BSP created in the context of this contract are contained in folder `cross-compilers /cross_gcc_scripts.`

Now that we have produced the first cross-compiled application program we can proceed to the rest of the tutorial.

3.4.2 Running a Simple Program

Running programs on the generated simulators is simple: in case we are using the standalone Instruction-Accurate simulator (whose executable file is called `funcLT`), the command to run the `test` application program is:

```
funcLT –a test
```

The result of the execution should look like:

```
            SystemC 2.2.0 --- Dec 15 2008 10:29:20
             Copyright (c) 1996-2006 by all Contributors
                         ALL RIGHTS RESERVED

     calling foo1
     inside foo1
     called foo1
     inside foo1

     Program exited with value 0

     SystemC: simulation stopped by user.
     Elapsed 0 sec.
     Executed 3481 instructions
     Execution Speed inf MIPS
     Elapsed 4119 cycles
```

As you can see, the `printf` instructions have been forwarded to the host OS which has executed them, printing their argument to the Linux shell. After such prints, line `Program exited with value 0` indicates the program return value, i.e. the return value of the `main` routine of the applicative program. The simulator then prints some statistics about the execution, in particular: host elapsed time, number of assembly instructions executed, Simulator Execution Speed, simulated elapsed time (SystemC time in case SystemC is employed, simply the approximated number of cycles for the Standalone non-SystemC-base simulator).

3.4.3 Exploiting the OS Emulator Capabilities

Actually the OS Emulation capabilities have already been exploited in the previous example for redirecting the `printf` instructions to the host Operating System: it is clear that the emulator is totally transparent to the user, a part from using the `–specs=osemu.specs` compilation flag no special actions have to be taken. In this paragraph we show how to write simple programs that read the command line arguments, the environmental variables, and the system configuration information; note that such programs do not use any special instruction, but the standard C directives for performing such tasks:

```
01. #include <stdlib.h>
02. #include <stdio.h>
03.
04. int main(int argc, char * argv[]){
05.     int i = 0;
06.     printf("There are %d arguments\n", argc);
07.     for(i = 0; i < argc; i++){
08.         printf("The %d-th argument is %s\n", i, argv[i]);
09.     }
10.     return 0;
11. }
```

```
12.
```

Let's now cross-compile the program with the instructions given above into the executable file *test*; then we can simulated it as:

```
funcLT –a test ––arguments one,two,three,four
```

The output of the run is:

```
             SystemC 2.2.0 --- Dec 15 2008 10:29:20
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED

There are 5 arguments
The 0-th argument is test
The 1-th argument is one
The 2-th argument is two
The 3-th argument is three
The 4-th argument is four

Program exited with value 0

SystemC: simulation stopped by user.
Elapsed 0 sec.
Executed 9664 instructions
Execution Speed inf MIPS
Elapsed 10965 cycles
```

Note the use of the `--arguments` command line argument to specify the application command line arguments.

```
01. #include <stdlib.h>
02. #include <stdio.h>
03.
04. int main(int argc, char * argv[]){
05.     printf("The env ONE is -%s-\n", getenv("ONE"));
06.     printf("The env TWO is -%s-\n", getenv("TWO"));
07.     if(getenv("THREE") == NULL){
08.         printf("Not found THREE");
09.     }
10.     else{
11.         printf("Found THREE");
12.     }
13.
14.     return 0;
15. }
16.
```

Let's now cross-compile the program with the instructions given above into the executable file *test*; then we can simulated it as:

```
funcLT -a test --environment ONE=foo,TWO=fii
```

The output of the run is:

```
             SystemC 2.2.0 --- Dec 15 2008 10:29:20
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED

The env ONE is -foo-
The env TWO is -fii-
Not found THREE
Program exited with value 0

SystemC: simulation stopped by user.
Elapsed 0.01 sec.
Executed 3907 instructions
Execution Speed 0.3907 MIPS
Elapsed 4577 cycles
```

Note the use of the `--environment` command line argument to specify the environment.

```
01. #include <stdlib.h>
02. #include <stdio.h>
03. #include <unistd.h>
04.
05. int main(int argc, char * argv[]){
06.     printf("The _SC_NPROCESSORS_ONLN value is %ld\n", sysconf(_SC_NPROCESSORS_ONLN));
07.     printf("The _SC_CLK_TCK value is %ld\n", sysconf(_SC_CLK_TCK));
08.
09.     return 0;
10. }
11.
```

Let's now cross-compile the program with the instructions given above into the executable file *test*; then we can simulated it as:

```
funcLT -a test --sysconf _SC_NPROCESSORS_ONLN=2,_SC_CLK_TCK=500
```

The output of the run is:

```
             SystemC 2.2.0 --- Dec 15 2008 10:29:20
        Copyright (c) 1996-2006 by all Contributors
                    ALL RIGHTS RESERVED

The _SC_NPROCESSORS_ONLN value is 2
The _SC_CLK_TCK value is 500
```

```
Program exited with value 0

SystemC: simulation stopped by user.
Elapsed 0.01 sec.
Executed 3637 instructions
Execution Speed 0.3637 MIPS
Elapsed 4319 cycles
```

Note the use of the `--sysconf` command line argument to specify the system configuration information; with respect to the other two examples, which can take an arbitrary environment and arbitrary command line parameters, the system configuration can only be specified for the `_SC_NPROCESSORS_ONLN` and `_SC_CLK_TCK` parameters, which respectively identify the number of online (available) processors and the number of clock ticks per second. In case it is necessary to consider other configuration parameters, the file *sysCallB.hpp*, part of TRAP's runtime in folder *runtime/osEmulator*, must be accordingly modified.

3.4.4 Using GDB Debugger

The standard GDB debugger can be used for debugging programs running on the generated simulators, so a more in depth guide on the use of GDB can be found on the internet; anyway this Section aim at showing in brief a sample debugging session. Note that, as for the GCC compiler, the plain debugger of the host system cannot be used, but the cross-debugger (running on your host system, able to debug sparc architectures) for sparc architectures have to be employed; such debugger is contained in the same folder of the cross-compiler.
For this example we will use the application program written in Section 3.4.1: lets run it on the simulator (for this example we do not use the standalone simulator anymore, but the one featuring the use of SystemC to keep track of time) with the `--debugger` command line option:

Now the simulator is stopped waiting for the connection of the GDB debugger:

```
                SystemC 2.2.0 --- Dec 15 2008 10:29:20
            Copyright (c) 1996-2006 by all Contributors
                        ALL RIGHTS RESERVED

GDB: waiting for connections on port 1500
```

Let's now start GDB in a different terminal:

```
sparc-elf-gdb test
```

GDB can be connected to the *remote target* (i.e. the simulator) by typing the following instruction in the GDB command prompt:

```
(gdb) target remote localhost:1500
```

Now simulation is still stopped, but the connection between the debugger and the simulator has been successfully performed. At this time it is possible to set breakpoints, watchpoints, etc. or simply to resume simulation with the `cont` GDB command. Let's set a breakpoint on *main* and then resume simulation:

`(gdb) break main` (which responds with *`Breakpoint 1 at 0x940: file prova.c, line 15.`*)
`(gdb) cont` (which responds, when the main routine is encountered, with *`Breakpoint 1, main (argc=1, argv=0x13da4) at test.c:15, 15 foo2();`*)

Now simulation is stopped at the beginning of the *main* routine: we can use the *monitor* commands, which can be listed with the *monitor help* command at the GDB command prompt:

```
(gdb) monitor help
```

which responds with:

```
Help about the custom GDB commands available for TRAP generated simulators:
   monitor help:       prints the current message
   monitor time:       returns the current simulation time
   monitor status:     returns the status of the simulation
   monitor go n:        after the 'continue' command is given, it simulates for n (ns)
starting from the current time
   monitor go_abs n:   after the 'continue' command is given, it simulates up to instant
n (ns)
   monitor history n:  prints the last n (up to a maximum of 1000) instructions
```

Let's use `monitor time` to examine the simulated flow of time:

`(gdb) monitor time` (which responds with `1501 (us)`)

We can also inspect the value of some variables, for example the *argv* parameter of the *main* function:

```
(gdb) p argv[0]
```

and we see that it correctly is a string containing the name of the application program being simulated:

```
$1 = 0x13dac "test"
```

Finally we resume simulation, which runs until the end:

```
(gdb) cont
```

which responds with:

```
Program Correctly Ended

Program exited normally.
```

3.4.5 Using the Profiler

For this example we will use the application program written in Section 3.4.1: lets run it on the simulator with the `--profiler` command line option:

```
funcLT –a test --profiler prof_out
```

After the execution, profiling results will be saved in files *`prof_out_instr.csv`* and *`prof_out_fun.csv`*; such files have the format already presented in Section 2.2.4.

3.5 Integration in a SystemC Environment

In order to allow the integration of the simulator into a SystemC and TLM based Virtual Platform (or simply in order to enable its connection with other SystemC IP-models) the simulators must be created with SystemC enabled and with TLM interfaces instead of using an internal memory; refer to Section 2.3 for details on how to generate different simulator flavors; anyway C++ sources of already generated simulators are available in folders *LEON2/sources* and *LEON3/sources*. There are three different kind of ways of communicating with the external world: through *TLM memory ports*, through *TLM interrupt ports*,

and through *TLM pins*. In the following we will show how to isolate the processor core from the rest of the simulator and how to use such TLM ports.

Note that communication through the TLM interfaces takes place according to the processor's endiannes: *big-endian* processors (as the LEON processor) send data through the interface using big-endian ordering, while *little-endian* ones (as the ARM processor) send it with little-endian ordering, independently of the byte-ordering of the host machine. All the processor models, anyway, internally work with the host endiannes, in order to ease the processor description and to improve execution speed.

3.5.1 Isolating the Processor Core

The generated simulators, the tools, and the external memories are connected together in the generated *main.cpp* file; the processor core is, thus, already perfectly isolated from the rest of the system, to which it can be connected as follows:

- `Processor procInst("LEON3", sc_time(1, SC_US));` : instantiation of the processor object, the first parameter being its name and the second the clock latency (the inverse of the frequency).
- `procInst.instrMem.initSocket.bind(mem_port);` and `procInst.dataMem.initSocket.bind(mem_port);` : binding of the TLM memory ports toward the interconnection medium (e.g. bus) or towards a cache, etc.
- `procInst.ENTRY_POINT`, `procInst.PROGRAM_LIMIT`, and `procInst.PROGRAM_START` must be set respectively to: (a) the application program entry point (i.e. the address of the first instruction which has to be executed), (b) the highest address of the application program (including data, i.e. the lowest address of free memory after program loading), and (c) the lowest address of the code of the application program (sometimes this is the same as the entry point, but not always).
- `procInst.toolManager.addTool(tool);` : called to add tools to the processor. In case the specific tool also needs to monitor memory ports (as it is for a debugger) for each memory port which needs monitoring it is necessary to call `procInst.instrMem.setDebugger(tool);`
- In case there is the need to also use the interrupt ports (even if interrupts are declared in the processor they do not necessarily need to be connected to an interrupt source, the port can be left unconnected), they can be connected as follows: *init_port.bind(*`procInst.IRQ_port.socket`*);*, where *init_port* is the TLM port of the interrupt source.
- Finally if the design features PIN ports (remember, a any TLM port which is not a memory or IRQ port is a PIN port), and for example the LEON2/3 models feature a port for the interrupt acknowledgment (called `irqAck`), they can be connected with the PIN target as `procInst.irqAck.initSocket.bind(pin_target_port);`

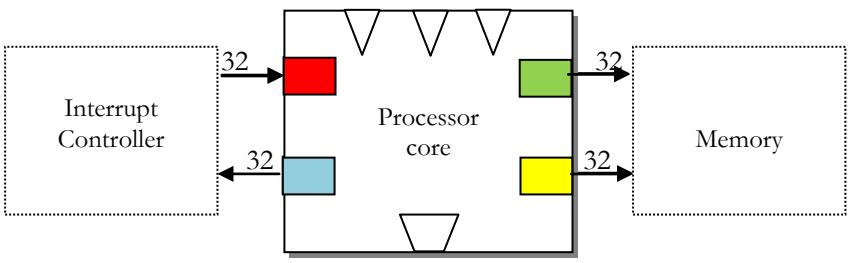

Figure 1: Diagram of the processor core together with its external interfaces. Square boxes represent TLM 2.0 interfaces, represents the acknowledgment port, represents the incoming interrupt port, while and the ports for memory mapped communication represents the `addTool` method, for adding to the processor the necessary tools (e.g. OS emulator, debugger, etc.); finally stands for the variables which give to the processor core information about the executable application being simulated (i.e. `ENTRY_POINT`, `PROGRAM_LIMIT`, and `PROGRAM_START`).

3.5.2 TLM Loosely-Timed Memory Interfaces

The loosely-timed memory interface is produced for the *funcLT* and *accLT* processor flavors when SystemC is employed and no internal memory is used; refer to Section 2.3 for the description of the different processor models and of the specification of TLM interfaces into the processor models.

The memory interface is contained into the `externalPorts.hpp` and `externalPorts.cpp` files; it uses the concepts of blocking transports interface (for standard communication), direct memory interface (dmi) to improve simulation speed by directly accessing the target memory storage, and the debug interface for non-standard communication (such as the memory traffic generated by the Operating System emulator and by the GDB debugger).
Instruction-Accurate processors also make use of the *tlm_quantumkeeper* concept to improve simulation speed by reducing the synchronization points with the SystemC scheduler. Cycle-Accurate processors, instead, do not employ the *tlm_quantumkeeper* as each pipeline stage is composed of a different SystemC thread (SC_THREAD) and such stages must be always synchronized with respect to each other.
For more details on the inner working of the cited elements, refer to the TLM 2.0 user manual (contained in folder doc of the TLM 2.0 distribution)

The TLM port itself is implemented with a *simple_initiator_socket* (as defined in the *tlm_utils* namespace) with a width of 32 bits.

3.5.3 TLM Accurate-Timed Memory Interfaces

The accurate-timed memory interface is produced for the *funcAT* and *accAT* processor flavors; refer to Section 2.3 for the description of the different processor models and of the specification of TLM interfaces into the processor models.

The memory interface is contained into the `externalPorts.hpp` and `externalPorts.cpp` files; it uses the concepts of non-blocking transports interface (for standard communication) and the debug interface for non-standard communication (such as the memory traffic generated by the Operating System emulator and by the GDB debugger). Even though the non-blocking interface is used, the port itself is blocking, which means that calls to the write or read methods of the ports do not return until the memory transaction has not correctly completed.

The TLM port itself is implemented with a *simple_initiator_socket* (as defined in the *tlm_utils* namespace) with a width of 32 bits.

3.5.4 TLM Interrupt Ports

The interrupt ports are implemented in the `irqPorts.cpp` and `irqPort.hpp` generated files; they use the concept of blocking interface for receiving the interrupts.

The port itself is implemented with a *multi_passthrough_target_socket* which allows the connection of at most 1 initiator socket (this means that there might even be no interrupt source connected with the processor if it is not needed).

The interrupt port defined in the LEON2/3 models has a width of 32 bits; two different data values can be sent through this port: if the data value is different from 0 then the address of the transaction can have a value between 1 and 15 and it corresponds to the interrupt priority (1 being the lowest and 15 the highest priority level). The interrupt is level triggered: once an interrupt has been raised, to lower it is

needed to write a transaction with a data value equal to 0. Note that interrupts are not buffered: if the interrupt source lowers the interrupt before the processor has been able to service it, the interrupt is lost, it will not be serviced.

When an interrupt is sent to the processor through the interrupt port, the *IRQ* variable of the processor class is set to the interrupt priority level (i.e. to the address of the received interrupt transaction): then, at each cycle (for the Cycle-Accurate processor) or before issuing a new instruction (for the Instruction-Accurate processor) there is a check to see if the interrupt has to be serviced:

```
(IRQ != -1) && (PSR[key_ET] && (IRQ == 15 || IRQ > PSR[key_PIL]))
```

if it is the case, the interrupt behavior is executed.

3.5.5 TLM PIN ports

PIN ports are defined in files `externalPins.hpp` and `externalPins.cpp` generated files; they use the concept of blocking interface for communicating with the external world. A PIN port is a TLM port not related to interrupts or to memory-mapped communication; it is possible to declare an arbitrary number of PIN ports in TRAP's based designs.

In general PIN ports can be both initiator or target ports (i.e. inbound or outbound) and they can transfer any arbitrary value, depending on the declaration in the main architectural file of the processor model (e.g. LEON3Arch.py). In the LEON2/3 case, a single initiator port was declared to be used for interrupt acknowledgement, transporting an integer value of 32 bits. Such value represent the priority level (1 to 15) of the interrupt being acknowledged. Internally the port is declared as a *multi_passthrough_initiator_socket* allowing the connection of at most 1 target socket (this means that there might even be no pin target connected with the processor if it is not needed).
The blocking transport method is used for communication through this port.

3.5.6 Additional Glue

The implementation of some parts of the processor model is not finished, and it can only be completed together with the rest of the SystemC-on-Chip, in particular together with the system bus.
The following details still need to be implemented:

- `SWAP` Instruction, implemented in the classes `SWAP_imm`, `SWAP_reg`, `SWAPA_reg` and in the ISA instructions with the same name. For the whole duration of the instruction the bus has to be locked so that no other instruction can access memory and, thus, SWAP holds the property of atomicity. As the TLM 2.0 standard does not give any guidelines on how to do it, such operation has to be implemented ad hoc for the particular bus model which shall be connected to the processor.
- In all load and store alternate instructions, the ones that make use of the ASI codes (to work on alternate memory spaces) do not consider such codes and the instructions behave as normal load/store instructions.
  As the TLM 2.0 standard does not given any guideline on how to work with alternate memory spaces, such instructions have to be modified to take ASI codes into account accordingly to the bus behavior.
- Instructions `STBAR` and `FLUSH` (implemented in `STBAR`, `FLUSH_reg` and `FLUSH_imm` classes) still have to be completed implemented, as their behavior only affects the bus and the memory status and how they perform such actions depends on the internal implementation of the bus and the memory.

Instruction classes are contained in files `instructions.hpp` and `instructions.cpp`, files which are directly generated from the *LEON3Isa.py* file. Even though it is perfectly fine to modify the generated

C++ files, I suggest that, in order to complete the gluing between the processor and the rest of the SoC, file *LEON3Isa.py* is modified and the simulator implementations re-created using TRAP.