Checkers Simulator Manual

Release 11R1
March 2011

Target Compiler Technologies NV
Technologielaan 11-0002
B-3001 Leuven
Belgium
Tel +32 16 38 10 30
Fax +32 16 38 10 49
http://www.retarget.com
mailto:support@retarget.com

Legal notes

Change log

Version	Date	Change
10R1	May 2010	Initial revision. This manual has been largely reworked in view of the new simulator GUI, fully integrated in ChessDE. This new manual replaces the previous ISS user, reference, and simulation models manuals.
11R1	January 2010	Hazard, primitive operations, functional unit, instruction class and *nML* coverage reports. Start Jtalk server from CHESSDE. Align micro-code views in compilation and simulation views. More formatting options for storages. Cloning of register and storages view. Support for `chess_report` functions. Runtime file and graphical I/O linked to breakpoints.

Contents

Chapter 1
Introduction

CHECKERS is an Instruction Set Simulator (ISS) generator. This manual describes how to build and use an ISS.

CHECKERS generates three types of ISSes :

Cycle accurate A cycle accurate (**CA**) ISS simulates every instruction cycle by cycle. The pipeline stages are fully visible. This cycle-accurate model, which is our default simulation model, is fully compatible with the RTL model of the processor, generated by Go [1].

Instruction accurate An instruction accurate (**IA**) ISS makes abstraction of the pipeline stages. The ISS simulates a single instruction at a time. This simulation model obtains higher simulation speeds (a factor ten compared to the cycle-accurate model). However, for most applications, the cycle-accurate model is sufficiently fast.

On-chip debugger The on-chip debugger (OCD) can be used to connect the ISS GUI with a physical processor core, either on an FPGA or in silicon, or with an RTL simulation model of the processor. To interface with the target processor, typically a **JTAG** interface is used.

In this manual, the term ISS refers to both the simulator and the on-chip debugger. Due to the different nature of simulation versus on-chip debugging, some features described in this manual are not available in the debugger context. This will be mentioned when describing the corresponding features.

ISSes may be used as stand-alone program (e.g. a cycle accurate ISS used from within CHESSDE or in batch mode) or embedded in other environments (e.g. in *SystemC* or as black box simulator).

The largest part of this manual, chapters 2 to 7, describes the usage of an ISS. Chapter 8 describes how to build and customize an ISS. Chapter 9 summarizes the processor modeling requirements.

Chapter 2
Running the ISS

2.1 With GUI

The GUI mode of the CHECKERS ISS can be started from CHESSDE. Before an ISS can be started, following conditions need to be met:

- An ISS executable should exist. Information on the generation of ISS executables can be found in section 8.
- For a cycle accurate ISS, instruction accurate ISS or an on-chip debug client, the path to the ISS executable should be configured in the **Debug settings** section of the CHESSDE **Project Settings** (see section 2.3).
- When CHESSDE needs to connect to an embedded simulator as client, the port to connect to can be configured in the **Debug settings** section of the CHESSDE **Project Settings** (see section 2.3).
- Multiple ISSes can be configured in the CHESSDE project settings at the same time. The ISS that should be started needs to be selected in the **Debug** → **Select debugger** menu. Select `Client` to use CHESSDE as client for an embedded ISS.
- The executable target of the project will be loaded in the ISS and therefore needs to be available before the ISS can be started.

Once the settings above have been made, the ISS can be launched using one of the following actions:

- Select the **Debug** → **Start debugging** menu entry.
- Press the **Start debugging** icon in the tool bar.
- Press the **F5** key.

When starting the ISS as on-chip debugger, some additional options can be given. These options are listed in table 2.1. The options can be set in the CHESSDE **Project settings** menu in the section **Debug settings**. The **-u** before the option should be left out when setting the options in the CHESSDE **Project settings** menu.

2.2 Text mode simulation

2.2.1 In CHESSDE

The ISS can also be started in text mode within the CHESSDE console window. To start the ISS that was selected in the **Debug** → **Select debugger** menu entry, do either of the following:

- Select the **Debug** → **Run in console** menu entry.
- Press the **Ctrl+F5** keys.

By default, the program will be loaded in the simulator and a **step -1** will be issued. A custom script can be defined in the **Project settings** menu of CHESSDE under the **Debug settings** entry. Enter the path to the custom script in the **Script for simulation in console** configuration setting.

2.2.2 From the command line

When an ISS is used without GUI, the user can interact with the ISS by typing commands on the command prompt or by sourcing script-files. To start an ISS without GUI, run the executable obtained by compiling and linking the generated C++ source files with the `-T` argument. The name of the executable is typically the same as the name of the modeled processor. To start the ISS of the *tinycore* without GUI, the following command could be used:

```
tinycore -T [-t <tcl-script>]
```

A prompt (by default %) is displayed at which the user can start typing commands. These commands must be valid *Tcl/tk* commands. The command interpreter used is a *Tcl/tk* interpreter extended with commands to interact with the ISS. More information about *Tcl/tk* can be found at `http://www.tcl.tk`.

Information on the creation and use of a script and running the ISS in batch mode can be found in chapter 7. Information on the *Tcl/tk* extensions used to interact with the ISS can be found in [2]. On Windows the ISS should be compiled as a console application. For details, refer to chapter 8.

2.2.3 Debugger

When using the ISS as on-chip debugger, some additional command line arguments are available. These are listed in table 2.1 and correspond to the `-u` option you would specify on the command line.

argument	*description*
`server=<server>`	Specifies JTAG-socket server (default: localhost).
`core=<core-id>`	Specifies core to attach to in case of multi-core debugger (default: 1).
`verbose`	Run debugger in verbose mode.
`jts_verbose`	Run JTAG-socket interface in verbose mode.
`attach`	Attach to (running) core without resetting.
`mask=<mem-mask-file>`	Specifies memory mask file.

Table 2.1: Debugger command line arguments.

memory mask file:

The memory mask file can contain one or more lines formatted as:

```
<memory name> <address> <value>
```

For each specified `address`, the debugger will never read the actual memory, but return the corresponding `value` instead. `address` and `value` must be specified in hexadecimal radix.

2.3 Debug settings

Before starting the ISS, a number of simulation specific settings can be made in the **Debug settings** section of the CHESSDE **Project settings**. These settings can be made in the following places:

- In the compilation settings of the processor project.
- In the project settings of the application that generates the executable to be loaded in the ISS. This level can inherit or override settings made on the processor level.

The following settings can be configured:

Simulators (cycle accurate) Provide the path or paths to one or more cycle accurate ISSes. The ISS to be used should be selected in the **Debug** → **Select debugger** menu.

Simulators (instruction accurate) Provide the path or paths to one or more instruction accurate ISSes. The ISS to be used should be selected in the **Debug** → **Select debugger** menu.

Debuggers (on chip) Provide the path or paths to one or more on-chip debug clients. The OCD client to be used should be selected in the **Debug** → **Select debugger** menu.

Options for on-chip debugger Specify options needed by the OCD client(s). A list of options is given in table 2.1. Leave out the **-u** when specifying options here.

Avoid reading memory ranges (memory start end) Instruct the debug GUI not to read from the specified memory ranges. A memory range must be specified as follows:

```
<memory-name> <lower bound> <upper bound>
```

Name of processor model in simulator When the name of the processor used to build the ISS or OCD is different from the name of the processor used to build the application, the name of the processor used to build the ISS or OCD can be specified here. This name will be used in the `iss::create` command when starting the ISS or OCD.

Server port when connecting to embedded ISS When a port is specified here, the **Debug** → **Select debugger** menu will contain a **Client** entry. Selecting the client entry allows to start an ISS client that connect to an embedded ISS on the given port.

Script for simulation in console This setting allows to specify a Tcl script that will be used when the ISS is started in console mode.

ISS plugins This entry allows to specify plugins for the ISS.

Extra arguments when creating ISS Extra arguments to be used when creating an ISS.

Extra arguments when starting simulator or debugger Extra command line arguments to be added to the command used to start the simulator or debugger executable.

2.3.1 Program load settings

Program load settings are gouped on a separate screen:

Load program arguments (argc/argv) When enabled, the arguments given in the next setting will passed to the application's main function when starting the ISS.

Program arguments (argc/argv) The arguments that should be passed to the main function of the application when starting the ISS can be listed here.

Load extra programs It's possible to load multiple programs into the ISS. The program counter and stack pointer will be set to those defined by the current application which is loaded last. This setting can be used to specify an additional list of programs to be loaded in the simulator.

Only initialize/Do not intialize memory ranges (memory start end) When certain parts of the processor memories should not be written by the loader, these parts of the memory can be specified in the **Do not intialize** entry. Or, by specifying memory ranges in the **Only initialize** entry, it is possible to specify only those parts of the processor memories where the loader can put data or instructions coming from the program being loaded. A memory range must be specified as follows:

```
<memory-name> <lower bound> <upper bound>
```

If both forbidden ranges and allowed ranges are given, an error will be issued. When one or more allowed ranges are specified, all memory locations not covered by the specified allowed ranges will be considered forbidden. This includes memories for which no allowed range was specified. When no allowed ranges are specified, all memory locations will be considered as allowed unless a forbidden range excludes it.

Only load symbol information When loading a program, memories are initialized with program and data from the *Elf* file. This setting can be used to disable loading of the memories.

Only initialize read-only segments With this option, only the read-only segments from an *Elf* file are loaded into the simulator. This includes data segments that are marked as read only and text segments. This option can be used in combination with a bootloader. In such case, the initialization values for the data segments are stored in ROM and copied to the respective data segments by the bootloader.

Do not initialize BSS sections With this option, loading of the BSS sections from an *Elf* file can be disabled.

Load program counter When a program is loaded, the program counter is set to the entry point found in the *Elf* executable or to zero if a micro-code file was loaded. This setting can be used to disable setting of the program counter.

Load stack pointer When a program is loaded, the stack pointer is set to the initial stack pointer found in the *Elf* file or to zero if a micro-code file was loaded. This setting can be used to disable setting of the stack pointer.

Check program memory after program load When this options is enabled, the ISS will compare the contents of the program memory with the expected value as found in the executable.

2.3.2 JTALK server settings

When using an on-chip debug client, CHESSDE will check if a JTALK serer is available on port 41001 of the server specified in the debugger settings. If no JTALK server is available, CHESSDE will try to start one if `localhost` is specified as server to run it on. Otherwise CHESSDE will show a dialog asking to start the JTALK server on the specified server.

CHESSDE start the JTALK server with arguments `-k -c ausb`. Additional arguments can be specified on the JTALK server screen.

Chapter 3
Programs

3.1 Loading programs

When starting the ISS in graphical mode from within CHESSDE, the program is loaded automatically.

Multiple executables can be loaded when a list of elf or mic files is configured in the **Project settings** (see section 2.3). For an overview of the mic format, refer to chapter B.

When starting the ISS in console mode from within CHESSDE, a default script is used that loads the executable and starts the simulation with **run -1**. A custom script for use in text mode can be configured in the **Project settings** (see section 2.3). For more information on scripts and running the simulation in batch mode, refer to chapter 7.

It's possible to specify program arguments via the argc/argv settings in the **Project settings** which are also described in section 2.3.

3.2 Micro-code and source-code windows

When a program is loaded into the ISS, the micro-code and, if the program was generated with *Dwarf* debug info, the source-code is displayed. The location in micro-code and source-code corresponds to the program counter value.

3.2.1 Micro-code window

The micro-code window displays the following information in columns from left to right:

Pipeline position Position of the instructions in the pipeline is displayed by coloring them in different colors. See 3.4 for more information about the pipeline.

Breakpoint indicators Micro-code breakpoint indicators are shown as **red** bullets. See 3.6 for more info on breakpoints.

Source-reference indicators Source-reference indicators are shown as small bullets or triangles. See 3.5 for more info on source-references.

Address of instruction Instruction addresses are printed as integers. The radix used to print the address can be changed through the right-click menu.

Instructions The instructions are printed using the user-defined output-stream operator of the program-memory type (also see 8.6). The radix used to print the instructions can be set using the right-click menu.

Disassembled instructions disassembled instructions are shown in the next column.

Instruction annotations Instruction annotations are shown in a dark colored line above an instruction. The following annotations can appear in this line:

- **.delay_slot** : instruction resides in a delay slot
- **.eol** : instruction is last instruction in a hardware loop
- **.nohwbrkpt** : invalid hardware breakpoint location
- **.noswbrkpt** : invalid software breakpoint location
- **.swstall** <**class name**> : `nop` instruction inserted to fulfill software stall of reported class
- **.rts** : instruction is jump doing a return from sub-routine
- **.tail_call** : instruction is a tail call

Function name Function names and mangled function names are shown in a dark colored line in between the program memory lines. These are located before the first instruction corresponding to the function.

Labels Labels like jump targets can be displayed in-line. For any label to be shown, the executable should be linked using the **-S** flag, which adds the symbol table to the executable. The visibility of the labels column is optional and can be set using the right-click menu.

The micro-code window can be searched using a find dialog or using incremental search.

- To use the find dialog, make sure the micro-code window is the active window. Then press **Ctrl+F** or select **Edit** → **Find**. Enter the search string and press find to jump to the first matching string in the micro-code window. Repeat pressing find to jump to the next matches. Once the end of the micro-code window is reached, the search wraps to the start of the micro-code.

  The find dialog offers options to search backwards, to make the search case sensitive, use regular expressions for the search or to match on whole words only.

- To use the incremental find, press **Ctrl+I** or select **Edit** → **Incr find**. Start typing the search string. The micro-code window will jump to the first string match. Repeat pressing **Ctrl+I** to jump to next string match.

  Use **Ctrl+J** or **Edit** → **Incr find back** to search backwards.

The micro-code window show the same information both in the compilation and in the simulation view. Options to configure the micro-code window are:

Hex address Show addresses using hexdecimal radix iso decimal.

Hex assembly Show constants in assembly using hexadecimal radix iso decimal

Instructions Show instructions

Annotations Show instruction annotations

Labels Show labels. Make sure to link with the `-S` options to be abel to show labels.

Additionaly options in the simulation view:

Goto address Go to a specified address.

Configure breakpoints Set, configure or clear a breakpoint.

Run to cursor Run simulation until this instruction is reached.

Disassemble range Disassemble instructions in an addresses range you can specify in a dialog window, and show the disassembled instructions in the micro-code window.

3.2.2 Source-code windows

Multiple source files can be open, each in its own editor window. A source-code window displays the following information:

Source-reference indicators Source-reference indicators are shown as small bullets. See 3.5 for more info on source-references.

Breakpoint indicators Source-code breakpoint indicators are shown as **red** bullets. In the source-code window, the source-reference indicators are the only locations where breakpoints can be set because a source code breakpoint needs a valid source-reference. See 3.6 for more info on breakpoints.

Line numbers This column displays the line number in the source-code file. The visibility of this column can be set using the right-click menu.

Code folding These widgets can be used to fold sections of the source code. The visibility of this column can be set using the right-click menu.

Source code The source code is displayed in a separate window per file.

The source-code window can be searched using a find dialog or using incremental search. It's also possible to do a find and replace or to grep through the entire set of project files.

- To use the find dialog, make sure the source-code window is the active window. Then press **Ctrl+F** or select **Edit** → **Find**. Enter the search string and press find to jump to the first matching string in the source-code window. Repeat pressing find to jump to the next matches. Once the end of the source-code window is reached, the search wraps to the start of the source code.

  The find dialog offers options to search backwards, to make the search case sensitive, use regular expressions for the search or to match on whole words only.

- To use find and replace, press **Ctrl+H** or select **Edit** → **Replace** to open the **Replace** dialog. Enter the search string and press find to jump to the first matching string in the source code window. Enter the replace string and press the **Replace** button to replace the highlighted search target with the replace string. Or press the **Replace all** button to replace all occurrences of the search string with the replace string.

  The replace dialog offers the same additional options as the find dialog.

- To use the incremental find, press **Ctrl+I** or select **Edit** → **Incr find**. Start typing the search string. The source-code window will jump to the first string match. Repeat pressing **Ctrl+I** to jump to next string match.

  Use **Ctrl+J** or **Edit** → **Incr find back** to search backwards.

- To grep through all of the project source and header files, press **Ctrl+G** or select **Edit** → **Grep**. A dialog similar to the find dialog pops up. The search string entered will be searched for in all project files. The results are displayed in the **Console** window.

- To jump directly to a specific line in the current source file, press **Ctrl+L** or select **Edit** → **Go to line**. Enter the line number in the dialog and press **Ok**. The cursor will move to the line number specified.

3.3 Executing the loaded program

The ISS interface in CHESSDE provides the following tool-bar icons to control the simulation:

Go / Continue Simulate until an event happens. If an event is encountered, the simulation stops and control is returned to the user.

Break Pause the simulation and return control to the user.

Stop debugging Stop the ISS and return to the compilation mode of CHESSDE.

Restart Clicking the restart button will first save the current setup, than reset the ISS and afterwards load the setup.

Step in Simulate one cycle and return control to the user.

When stepping through the source-code, this button simulates until the next source reference is found and return control to the user. If an event happens before reaching the next source reference, simulation is stopped and control is returned to the user regardless of the fact the ISS reached a source reference or not.

Using this button, you will *step into* any called function.

Step N in Simulate the number of cycles specified by the user and return control to the user afterwards. When this button is clicked, a pop-up window appears that allows to specify the value of the step count **N**. After setting the value for **N**, the **Ok** button can be pressed and the simulation will start for the number of cycles specified.

The **Ignore break and watch point** check in the pop-up window can be deselected. In that case, if a break or watch point is hit before reaching the specified number of cycles, simulation is stopped and control is returned to the user.

To avoid the pop-up menu from appearing, hold the **Ctrl** key while clicking the button.

Using this button, you will *step into* any called function.

In source code stepping mode, this button is disabled.

Step over Simulate one cycle in the current call frame and return control to the user. If the instruction being executed is a jump to subroutine instruction, execution will continue until the called function returns. If an event happens before reaching the next instruction in the current call frame, simulation is stopped and control is returned to the user regardless of the call frame in which the event occurs.

In source-code stepping mode, the simulation will continue until the next source reference is found in the current call frame and control is returned to the user. If the instruction being executed is a jump to subroutine instruction, execution will continue until the called function returns.

Using this button, you will *step over* any called function.

Step out Simulate until the function in the current call frame returns. If an event happens before the function in the current call frame returns, simulation is stopped and control is returned to the user regardless of the call frame in which the event occurs.

In source-code stepping mode, the simulation will continue until the first source reference after the function in the current call frame returns.

Using this button, you will *step out* of the current function.

Source code stepping This is a checkbutton to switch between micro code and source code stepping. By default, micro code stepping is active. When *Dwarf* 2 information is available, this button can be checked. From then on, the stepping controls described above will operate in source-stepping mode.

3.3.1 Run to cursor

The simulation can also be run from within the source-code or micro-code windows using the **Run to cursor** entry of the right-click menu.

- In the micro-code window, hover the cursor over a micro-code instruction line and right-click. In the pop-up menu, select **Run to cursor**. The ISS will place a breakpoint on the micro-code instruction line and will start the simulation.
- In the source-code window, hover the cursor over a source-code line that has a source-reference (indicated by a small triangle or a small bullet in front of it). In the right-click pop-up menu, select **Run to cursor**. The ISS will place a breakpoint on the corresponding micro-code instruction and will start the simulation.

3.3.2 Differences between ISS and Debugger

A fundamental difference exists between simulation and debugging on chip. An ISS usually offers greater controllability and observability than a debug interface. Therefore some features described in this manual aren't available in the debug client. For example:

- A cycle accurate ISS contains a pipeline window and can be stopped every cycle. The pipeline stages can be stepped and followed cycle by cycle. When a breakpoint is hit, the simulation stops right before the instruction with the breakpoint enters the breakpoint stage. This breakpoint stage is typically set to the main execute stage of the processor, i.e., the stage where most registers are updated.
- A debug client connected to a core can issue a stop action to halt the core or the core can be stopped by a breakpoint. In both cases, the pipeline is flushed and there is no view on the pipeline. When a breakpoint is hit, the core simply stops in front of the instruction with the breakpoint.
- An instructon accurate ISS has no pipeline view, and regarding breakpoints, it behaves like a debug client.

3.4 Pipeline

When stepping through a loaded program, the micro-code and source-code windows show which instruction and source-statements are being executed. The color-coded pipeline positions in the micro-code window give stage information for each instruction. More detailed stage information is given in the pipeline window.

The pipeline window is only available in cycle-accurate simulators. On-chip debuggers do not have access to pipeline information, as this pipeline is always flushed when the processor enters debug mode. The visibility of the pipeline window can be toggled from the **View** menu in the ISS mode of CHESSDE.

The different instructions in the pipeline are shown from top to bottom with the address of each instruction shown at the left and right side of the window. The instructions being in the pipeline for the longest time are shown above instruction inserted into the pipeline at a later cycle. The radix for the addresses can be chosen in the right-click pop-up menu.

Each stage of an instruction is shown separately from left to right. Stage names are printed for each instruction part. Each instruction part is positioned in the cycle where it will be executed. Cycles are ordered from left to right. Cycles numbers are shown at the top and at the bottom of the pipeline window. The current cycle is shown a vertical green bar. Cycles from the past are shown to the left of the current cycle. Cycles in the future are show to the right of the current cycle. Cycles are shown as system cycles, not user cycles. User cycles may differ from system cycles when inserting wait-states or when extra cycles are added to the number of user-cycles without making this visible in the nML, e.g. from the memory-hooks. The radix for cycle counts and data in the information window below can be chosen in the right-click pop-up menu. This menu also allows to show additional **Action details** in the lower part of the pipeline window.

In the pipeline view, the program counter labels, the cycle count labels and the instruction pipeline stages that contain operations are clickable. When one of these labels is clicked, the information window in the lower part of the pipeline view window will show details corresponding to the selected label. In this window, following elements are displayed:

- The first block of the information window displays the instruction address, the decode index and the assembly that corresponds to the instruction. If the information window is called from the label of a cycle count which has multiple instructions in the pipeline, the first block will show the information corresponding to the most recent instruction issued.
- The second block lists the nML operations that correspond to the specific pipeline stage mentioned in the first line of this block.

- In case a program counter value was clicked and the corresponding instruction contains active operations in multiple pipeline stages, there will be additional blocks to list the operations for each of the pipeline stages.
- In case a cycle count value was clicked and the corresponding cycle contains active pipeline stages from multiple instructions, the first two blocks will be repeated for each of the instructions that have or had an active pipeline stage during this cycle.

3.5 Dwarf2: source line references

When a program is loaded into the ISS and it contains *Dwarf* debugging information, the ISS will display the link between the source-code and the micro-code. When source line references are found, each source-line referring to one or more micro-code line is preceded by a bullet. Also each micro-code line referenced by at least one source-line is preceded by a small bullet or a triangle.

One source-line may refer to one or more micro-code lines. Also one micro-code instruction can be referred by one or more source-lines. When a micro-code line is referenced by a source-line and it is the start of a source code statement, it is preceded by a small triangle. When a micro-code line is referenced by a source-line but it is not the start of a statement, is preceded by a small bullet.

The micro-code instructions referenced by a source-line can be displayed by clicking the source-code reference bullet to the left of the source-line. This will highlight the source line and all micro-code lines referenced by the source-line. The source-lines referencing a micro-code instruction can also be displayed by clicking the source-code reference bullet or triangle to the left of the micro-code instruction. This will highlight the micro-code instruction and the source-code location or locations (usually not the entire lines) corresponding to the micro-code instruction. Highlighting can be removed by clicking on the same source reference again.

To see the link between micro-code instructions and source code, the micro-code window is also available in the compilation view.

3.6 Breakpoints

Breakpoints can be put on both micro-code instructions and source-lines. To be able to put a breakpoint on a source-line, it must reference at least one instruction.

During execution, the ISS will check if a breakpoint was set on the micro-code instruction currently in the breakpoint-stage. If a breakpoint was set, the breakpoint is triggered. A breakpoint put on a source-line referencing one or more micro-code instruction will be triggered for the micro-code instruction marked by the CHESS compiler as being the beginning of the statement. When a breakpoint is triggered, the break conditions (see further) are examined. If these break conditions are fulfilled, the breakpoint is said to be hit. When a breakpoint is hit, an event is generated, simulation is interrupted and control is returned to the user.

Micro-code and source-code breakpoints are related to each other. If a source-code breakpoint is set, an open red circle will be shown in front of the corresponding micro-code instruction. The source-code breakpoint itself is marked with a red bullet. When the micro-code breakpoint marker is double-clicked, the red circle will become a red bullet to indicate the micro-code breakpoint. The red bullet in front of the source-code line disappears.

Temporary breakpoints are indicated with a blue bullet.

Breakpoints for hosted I/O, `chess_report` functions and file I/O triggered by breakpoints are indicated with a blue circle.

3.6.1 Adding, editing and removing breakpoints

The fastest way to add or remove a micro-code breakpoint is to double click the breakpoint indicator column next to the instruction where the breakpoint should be added. The fastest way to add or remove a source-code breakpoint is to double-click a source-line reference in the source-code window.

For more configuration details, a pop-up window for breakpoint configurations can be opened from the right-click menu in the micro-code and source-code windows. Right-click the line where a breakpoint should be added or modified and select **Configure breakpoint**.

The following items make up the breakpoint configuration pop-up window.

Hitcount entry In this entry, it is possible to set the number of times the breakpoint must be triggered before the break condition evaluates to true. The special hit count zero will make sure the breakpoint is never hit although it is triggered. With hit count zero simulation will never be interrupted for the breakpoints with hit count zero, while still printing the breakpoint is triggered in the console window.

Current hitcount entry This read-only entry shows the number of times the breakpoint was triggered since the last time it was hit.

Execution count entry This read-only entry shows the number of times the instruction was executed since startup or since the last reset of the profiling information. This entry is only available in the micro-code window.

Verbose check button Lets the user choose how the breakpoints which are triggered must be reported. If verbose is set to true, each breakpoint being triggered and hit is reported. If set to false, breakpoints being triggered or hit are never reported while still the breakpoint event is generated to interrupt simulation.

Multi-core breakpoint check button When this button is checked, the breakpoint hit will be exported to other cores in a multi-core debug environment.

Software Make the breakpoint a software breakpoint. This radiobutton is only available when using an on-chip debugger with software breakpoint support. Is software breakpoints are available, they are the default.

Hardware Make the breakpoint a hardware breakpoint. This radiobutton is only available when using an on-chip debugger with software breakpoint support.

Ok button Button used to set or modify the breakpoint as specified by the user. Clicking this button will also close the breakpoint pop-up.

Remove button Clicking this button will clear the breakpoint and close the breakpoint pop-up.

Cancel button Clicking this button will close the breakpoint pop-up without applying the changes made by the user.

All breakpoints can be removed using the **Remove all breakpoints** entry in the **Debug** menu.

3.6.2 Breakpoints overview

The break and watch points windows offers an overview of all active micro-code breakpoints, source-code breakpoints and watchpoint. Break and watch points can also be edited and remove from this window.

When simulation stops due to a break or watch point, those break or watch points are highlighted.

3.7 Simulation statistics and instruction history

The **Statistics** window can be opened from the **View** menu. While a simulation is in progress, this window shows the cycle count and the instruction count. Both are updated at fixed intervals.

When the simulation is halted, this window shows more information on the on-going simulation.

ISS command Command used to start the ISS

ISS mode Mode of the ISS, can be:

- Cycle accurate
- Hardware debugger
- Instruction accyrate

Cycle count The total number of cycles simulated so far.

Instruction count The total number of instructions processed so far. When dealing with multi-cycle instructions, the instruction count will be lower than the cycle count.

PC The program counter value for the instruction that is being fetched and will be issued in the next cycle.

Stack properties Information about the stack pointer and the stack area is shown here.

Instruction history The **Pipeline position** column of the micro-code window allows to see the most recent instruction history. When the same instructions is executed multiple times or when no instruction is executed at all, this is difficult to see in the micro-code window. To be able to see the exact order in which instructions are executed, a history buffer is kept for instructions and their addresses.

Each line contains the following information:

- Stage names are colored with the same colors as used in the micro-code.
- Program counter value
- Instruction, including multi-word instructions
- Assembly instruction

The instruction on top just entered the fetch pipeline. The number of stages kept can be configured when building the ISS. For more information, see section 8.4.1.

For all address values in the **Statistics** window, the radix can be set from the right-click menu.

Chapter 4

Storage elements

Several windows show insight in the processor storages. These windows can be grouped in following categories.

Registers Registers, transitories, ports and internal storages can be displayed in the **Registers** and in the **Storages** windows. These windows are described in section 4.1.

Memories Each memory or memory alias has an associated window in the ISS. These windows are described in section 4.2.

Variables Two windows show information on application variables. These windows are described in section 4.4.

4.1 Registers

Both the **Registers** and the **Storages** windows can be opened from the **View** menu in CHESSDE. Both windows have the same functionality. The difference between these windows is that the contents of the **Registers** window are defined on the processor level and the contents of the **Storages** window are defined on the project level.

By default, the **Registers** window contains an auto-generated list of all processor registers. The layout of this window can be edited only if the processor model is not in **Protect** mode. To disable **Protect** mode, open the **Project explorer** window, right-click the processor model name and deselect **Protect**. After editing the layout of the **Registers** window, the layout can be stored to the `<processor>.prx` file with the **Save all** command from the **File** menu.

The **Storages** window is empty by default. The contents of this window can be edited regardless of the processor protection. After editing the **Storages** window, the layout can be stored to the `<project>.prx` file with the **Save project** command from the **File** menu.

When coloring of storages on read and write is enabled, storage locations will be colored red when they are written, blue when they are read and purple when they are both read and written in the previously executed cycle.

When storage location values have changed since the previous time stepping was stopped, the background of those storages is colored dark.

To edit the **Registers** or the **Storages** window, open the right-click menu in the window and choose **Edit layout**. For information about the layout editor, refer to section 4.1.1.

In the right click menu of the **Registers** or the **Storages** window you find the following items:

Vector elements Vector registers can be displayed in a separate window. This window can be opened through the **View** menu or via this right-click menu item while hovering the mouse cursor over a vector register.

The vector window behaves like a memory view window. For more information on the use of a vector window, therefore refer to section 4.2.

Data format The basic data formats you can select in the right-click menu of a register window are **Hexadecimal**, **Signed**, **Unsigned**, and **Default**. The latter format is a decimal format where the signedness depends on the signedness of the nML data type of the storage.

Edit layout/Add storage Make changes to the current layout. Check section 4.1.1 for more information.

Configure watchpoint This menu item opens a pop-up window that allows to configure a watchpoint on the storage location under the cursor. Once configured, the background of the storage location will turn green.

Register groups When using collapsible register groups in the layout (see 4.1.1), these groups are listed here to let the user toggle the displaying of the groups.

All Show all collapsible register groups.

None Hide all collapsible register groups.

Copy Copy the highlighted string.

4.1.1 Editing a storages list

To edit the contents and the layout of the **Registers** window or the **Storages** window, open the right-click menu and select **Edit layout**. A new window will pop up. This is the **Edit register layout** window. This window has the following sections.

help A short overview of how to edit the layout.

storage selection A pull-down menu listing storages that can be inserted in the layout. The contents of the pull-down menu are controlled by a set of check buttons for each of the following items.

- Registers
- Transitories
- Ports
- Internal

Layout editing area When the layout editor is empty, all storages that are added will be added in a single column. Storage names should be separated by a space or a newline. To join storages on a single line, group them in a list using braces. Nested lists will alternate as rows and columns.

To insert an empty row or column, use **<empty>**.

To insert a label, use **<label> LABELTEXT**.

To insert a collapsible register group, use **<clabel> LABELTEXT**. These labels can only be added at the outermost level.

Ok button Clicking the **Ok** button closes the layout edit window and applies the new layout settings.

Default button Clicking the **Default** button closes the layout edit window. Existing or new layout settings are ignored. Instead, the default register layout is applied.

Cancel button Clicking the **Cancel** button closes the layout edit window without applying the modified layout settings.

To illustrate the configurations that can be edited, a simple example is given below. Both of the following configurations will result in a similar layout.

The first configuration defines two rows which are ordered in a column.

```
{<label> " " <label> "Column header"}
{<label> "Row header" PC}
```

The second configuration defines two columns which are grouped in a row.

```
{
{<label> " " <label> "Row header"}
{<label> "Column header" PC}
}
```

For both configurations, the generated layout looks as follows.

```
              Column header
 Row header   PC =        0
```

4.1.2 Value formatting

Besides the basic data formats you can select in the right-click menu of a register window (**Hexadecimal**, **Signed**, **Unsigned**, or **Default**), it is possible to select a more specific formatting when editing the register layout (§4.1.1).

When the compiler header file `<processor>_chess.h` defines fractional or floating point types (like `fract_t` or `float`), then you can force the corresponding fractional format by putting this type after the register name with an intermediate dot, e.g., when `f0` is a floating-point register, you can add `f0.float` to the register layout.

When having defined a user print function $t = f$ (§8.6), you can force this format for a register of type t by adding $.f$ after the register name in the register layout.

When not specifying a formatting type, values have other formatting options. Values can be given a fixed radix by adding `.bin`, `.dec` or `.hex` after the storage name. A value can be splitted in different parts by specifying the number of bits of each part separated by slashes after the storage name (e.g `A.4/16/16`). Radix and part bit width specifiers can be combined. When using part bit width specifies, it is also possible to use the `.hexonly` radix specifier. This indicates to only use the part bit width specifiers in when the **Registers** or the **Storages** window data format is set to hexadecimal radix, and to print the value as a whole in the other cases.

4.2 Memories

Windows for memories, memory aliases and vector registers can be opened from the **View** menu. This window is commonly called a memory view window. A memory view window shows the contents of a memory or vector register. The left column shows the memory addresses in increments of the number of columns displayed. The top row show the incremental memory address. The rightmost column optionally shows **ASCII** code corresponding to the memory contents. The width of the table can be configured through the right-click menu.

When coloring of storages on read and write is enabled, memory locations will be colored red when they are written, blue when they are read and purple when they are both read and written in the previously executed cycle.

When memory location values have changed since the previous time stepping was stopped, the background of those locations is colored dark.

The right-click menu contains the following items.

Values on line Configure the number of columns used to display the memory values. By default, this is set to **auto**. The **auto** setting will fill up the window and will adapt when the window is resized. Other possible values are 1, 2, 4, 8, 10, 16 and 32.

When displaying vector memories or vector registers and the **Vector elements** configuration is set, the maximum number of columns displayed is be equal to the number of vector elements. If a higher number is chosen, this will have no effect. If a lower number is chosen, the horizontal scroll bar of the memory view window allows to scroll through the vector elements.

Show ascii This setting controls whether the **ASCII** interpretation of the memory contents is shown at the right side of the memory view window. When the **Values on line** setting is **auto**, this control influences the number of columns shown in the memory view.

Vector elements This setting is only available when vector registers or vector memories are displayed. This setting controls whether the number of vector elements is used for the number of columns to display.

Data format This sub-menu allows to display the data values as hexadecimal values, as signed integers or as unsigned integers.

Grouping/User format function As in the **Registers** window or the **Storages** window, memory values can also be formatted by the user in the popup you get when selecting this item. Values can be splitted in different parts by specifying the number of bits of each part separated by slashes in the popup. Alternatively, user print functions can be specified in the popup.

Hex addresses This setting controls whether addresses are displayed in hexadecimal notation or in unsigned decimal notation.

Goto address This menu item opens a pop-up window which allows to specify a memory address to which the view should jump.

Goto symbol When a list of symbols is available for the memory, this menu item opens a pop-up window that allows to locate the object in memory.

Edit Edit the value over which you right clicked. Alternatively, double click the value to edit it.

Read Read the value over which you right clicked again.

Configure watchpoint This menu item opens a pop-up window that allows to configure a watchpoint on the memory location under the cursor. Once configured, the background of the memory location will turn green.

Clone Create a new window with displaying the same storages.

Copy Copy the highlighted string.

4.3 Watch points

Watch points can be used to interrupt simulation based on read or write actions. They are the data equivalent of breakpoints.

A watch point is triggered when a storage element field is read and/or written. If triggered, the watch conditions are evaluated. As with breakpoints one of the watch conditions is the hit count. It is possible to set a number of times the watch point must be triggered before it will effectively be considered as hit. In addition to the hit count, it is possible to specify an expression which must evaluate to true before the watch condition can be fulfilled. If no expression is used, the hit count is incremented each time the storage is read and/or written (as specified for the watch point). If an expression is used, the storage element must be read and/or written as specified for the watch point and the expression must evaluate to true before the hit count is incremented.

A watchpoint can cover multiple consecutive fields starting from the storage element field it was placed on.

4.3.1 Expressions

A logical expression is evaluated by the ISS using the *Tcl/tk* `expr` command. Any expression understood by this `expr` command is allowed. When entering a watch point expression, the storage element value must be replaced by a $ sign. Some valid watch point expressions are:

```
$ < 100
$ & 0xf
$ < 100 || $ >= 407
abs($) < 10
pow($,2) * 3.1415 > 100
```

Expressions are evaluated in *Tcl/tk* and are as such slower than logical operations.

4.3.2 Adding, editing, and removing watch points

To set a watch point, open the right-click menu and select **Configure watchpoint**. A new dialog window will open with the following contents.

Watch read This checkbutton indicates if the watchpoint triggers on a read access. By default, this checkbutton is off.

Watch write This checkbutton indicates if the watchpoint triggers on a write access. By default, this checkbutton is on.

Hitcount In this entry, it is possible to set the number of times the watch point must be triggered and all watch conditions must be fulfilled before the watch condition evaluates to true. The special hit count zero will make sure the watch point is never hit although it is triggered. With hit count zero simulation will never be interrupted for the watch points with hit count zero while still printing the watch point is triggered in the console window. The default value of this entry is 1.

Current hitcount This read-only entry shows the number of times the watch point was already triggered and all watch conditions were fulfilled since the last time it was hit.

Verbose Lets the user choose how the watch points which are triggered and all watch conditions are fulfilled must be reported. If verbose is set to true, each watch point being triggered and hit is reported. If set to false, watch points being triggered or hit are never reported while still the watch point event is generated to interrupt simulation. By default, this checkbutton is on.

Remove when hit When checked, this option removes the watchpoint as soon as it has been hit. By default, this checkbutton is off.

Expression An expression which will be evaluated each time the watch point is triggered can be entered in this entry. See also section 4.3.1.

4.4 Variables

The ISS provides two windows where variable information can be viewed and edited. Both of them can be controlled through the **View** menu.

Local variables In the **Local variables** window, variable information about local variables in current scope is displayed automatically. These are both variables with automatic and with static allocation. The stack trace is displayed at the top of this window.

In general, local variables with automatic allocation are displayed as soon as the function stack frame is allocated (the stack pointer is updated). At that moment the variables might not yet be initialized. Local variables with static allocation are displayed during the entire function.

When compiling in **Release** mode, local scalar variables are allocated to registers by default. These variables only show up during their actual lifetime, starting once they are initialized. The amount of variables displayed here can depend on the compilation. Some variables might not be available because of compiler optimizations.

When compiling in **Debug** mode, local scalar variables are allocated to the function stack frame and show up as soon as the function stack frame is allocated. Compiling in **Debug** mode ensures all variables in the source code are available for display.

Variable info The content of the **Variable Info** window doesn't change dynamically. This window can be used to display information on global and local variables. The initial window is empty and variables must be added manually by selecting them from a list. When a local variable being displayed is not in scope, the line displaying this variable is grayed out. Global variables have no scope and will never be grayed out. They can only be visualized in this window, not in the local variables window.

To add a variable to the **Variable info** window, open the right-click menu and select **Add variable**. A new dialog window opens in which variables can be selected to be added to the **Variable info** window.

4.5 Stack information

When the ISS models a processor using a stack-pointer register, some stack-pointer information is displayed in the **Statistics** window.

The **Statistics** window shows the lowest valid, the current and the highest valid value for the stack-pointer register. The lowest and highest values are taken from the *Elf* executable loaded into the ISS.

The **Dir** field indicates in which direction the stack grows. If the direction is **Up**, the stack starts at the lowest valid address and can grow up to the highest valid address. If the direction is **Down**, the stack starts at the highest valid address and grows down to lowest valid value.

If enabled in the CHECKERS configuration file, the ISS can report errors when the stack exceeds its valid range.

Stack unwinding information can be found in the **Profiling** window (see 6).

Chapter 5

File input and output

CHECKERS provides following ways to do file I/O in a simulation session :

- **Hosted I/O.** Using the hosted I/O functionality of CHECKERS, you can use the `<stdio.h>` functions (`fopen()`, `fprintf()`, `fscanf()`, `fwrite()`, `fread()`, ...) in your C source code to do file I/O also on the target processor.
- `chess_report` **output.** When using `chess_report` functions in your C source code, this functionality of the ISS will write values passed to those `chess_report` functions to a file.
- **Runtime I/O.** Using the runtime file I/O functionality of CHECKERS, you can bind text files to specific memory locations of the target processor. Inside the simulator, each time the processor reads or writes such a memory location, the next element in the file is read or written.
- **Snapshot I/O.** Different from the three previous methods, snapshot I/O can only be used when the ISS is halted. It is used to initialize an address range in memory with the values in a file, or to dump the current memory contents for a given address range to a file. It can be setup to trigger when a breakpoint is hit.

This chapter describes how to setup runtime I/O (§5.1) and snapshot I/O (§5.2) in the CHESSDE GUI. Both I/O types are configured in the **I/O** window, which is opened from the **View** menu, when not already visible. Next to file output, runtime and snapshot I/O also provide graphical output, which is displayed in the **Graphical output** window.

Hosted file I/O is discussed in [3, §3.4], where it is compared to runtime I/O, both having their merits. Appendix A describes how hosted I/O is implemented by the ISS.

Note that runtime I/O is only available in simulation mode and not in debug clients.

5.1 Runtime I/O

Runtime I/O is typically configured to trigger every time the corresponding storage is read or written, but it is also possible to configure runtime file input to trigger every cycle, in combination with interval lists.

5.1.1 Runtime file input

Runtime file inputs are connected to specific storage elements (register or memory fields, or input ports), and will put a new value from the data file on the corresponding storage during simulation. Runtime file inputs can be configured to trigger every cycle (in combination with interval lists) or when the application program reads from the corresponding storage element.

Each file input configuration can be controlled by one or more *interval lists*. Such an interval list contains information about when the ISS should actually read from the data-file. When the file input is configured to

read from the data-file every cycle and is controlled by an interval-list, the ISS will only read a new value from the data-file after an interval specified in the interval-list has expired (and an event is generated). This can be used to mimic fixed sample intervals or random interrupts. More information about interval-list can be found in section 5.4.

To add a runtime file input, go to the **I/O** window and click on **Runtime I/O** → **File input** → **<new>**. Fill in the name of the storage. The drop-down menu can be used to select the storage name from a list. Then click **Ok**. A new settings window appears in which further details can be configured.

Storage The previously entered storage name is fixed. A storage name can't be edited for a given file input.

Address If the chosen storage contains multiple entries, the address can be entered here. Once chosen, the address remains fixed for a given file input. The radix of the entered value can be chosen on the right. The address range entry can be done in octal, decimal or hexadecimal radix.

Data format Configure the value formatting of the input values.

Data radix Select the radix of the data in the input file.

File name Choose the name of the file containing the input data or click the browse button to the right of the entry to select a file using a file-selection dialog.

Initial position Enter the position in the data-file were reading must start. When connecting the runtime file input to the storage element, the runtime file input will read the specified number of values from the file without writing them to the storage element.

Filter command The name of an executable used to convert the contents of the specified data file to a format understood by the ISS can be entered here or click the browse button to the right of the entry to select one using a file-selection dialog. When connecting the runtime file input to the storage element this executable will be called like this:

```
<filter executable> <data-file> <data-file>.checkers_flt
```

The ISS will use the contents of `<data-file>.checkers_flt` as data-input for the runtime file input.

Interval file The names of the interval-lists can be entered here. Multiple lists are separated by a space. When the interval-list file-name contains spaces, put it between braces. It is also possible to select the interval-lists using a file-selection dialog by clicking the browse button to the right of the entry.

Input file contains structured data This option specifies whether the input file is structured or not (§5.3).

Bytes per word (binary) In binary mode, values are read from the input file without any formatting. There are no spaces or newlines. This setting allows to specify how many bytes should be read per value. The number of bytes specified can be different from the type width, e.g. when the same input file is shared between an ISS using 24 bit integers and a native simulation where integers are 32 bit.

LSB first (binary) Specify the endianness of the bytes read in binary mode.

Stop simulation when input file wraps This entry allows the user to choose between stopping or continuing the simulation when all data-values from the data-file are read. If the entry is set to **On**, simulation will stop. This can be used to stop simulation when all data found in the data-file is processed. If the check button is disabled, simulation will continue and the runtime file input will start reading from the beginning of the data-file again.

Read from file This entry lets the user decide when the runtime file input will read from the data-file. The available options are:

When read When this entry is selected, the runtime file input will only be read when the ISS reads from the storage element the runtime file input is connected to. When one or more interval-lists are used, the ISS will only read from the data-file when its read from the storage element the runtime file input is connected to and at least one of the interval-lists has triggered an event.

Each cycle When this entry is selected, the runtime file input will read from its data-file each cycle. When one or more interval-lists are used, the ISS will only read from the data-file when at least one of the interval-lists has triggered an event.

Ok Click the **Ok** button to confirm the entries made for the file input configuration.

Cancel Click the **[**Cancel] button to cancel the file input configuration.

To edit an existing runtime file input, go to the **I/O** window and click on **Runtime I/O** → **File input** → **<NAME>**. Then click the **Edit** button.

The input file and the interval file can be opened from the **Open input file** and **Open interval file** buttons.

To remove a runtime file input, go to the **I/O** window and click on **Runtime I/O** → **File input** → **<NAME>**. Then click the **Remove** button.

5.1.2 Runtime file output

Runtime file outputs are connected to specific storage elements (register or memory fields, or output ports), and will write the value on the corresponding storage to the data file during simulation. Runtime file outputs can be configured to trigger every cycle (not useful) or when the application program writes to the corresponding storage element.

Runtime file outputs can also be used to compare simulation results of the ISS with RTL simulation results, done on the hardware model generated by GO.

To add a runtime file output, go to the **I/O** window and click on **Runtime I/O** → **File output** → **<new>**. Fill in the name of the storage. The drop-down menu can be used to select the storage name from a list. Then click **Ok**. A new settings window appears in which further details can be configured.

Storage The previously entered storage name is fixed. A storage name can't be edited for a given file output.

Address range If the chosen storage contains multiple entries, the address or address range can be entered here. The radix of the entered value can be chosen on the right. The address range entry can be done in octal, decimal or hexadecimal radix.

Data format Configure the value formatting of the output values. Can be:

- Integer
- Unsigned
- VHDL testbench format
- VHDL testbench format (all hex)
- Verilog testbench format
- Verilog testbench format (all hex)
- Value change dump

When the storage has fractional and floating point types it can be converted to, those types are added to the list of selectable formats.

Data radix Select the radix of the data in the output file.

Address radix Select the radix of the address in the output file.

Printf format string For the formats integer and unsigned, a custom format string can be defined here.

File name Choose the name of the file to which the output data should written or click the browse button to the right of the entry to select a file using a file-selection dialog.

Write This entry lets the user decide when the runtime file output will be written. The available options are:

When written When this entry is selected, the runtime file output will only be written when the ISS writes to the storage element the runtime file output is connected to.

Each cycle When this entry is selected, the runtime file output will be written each cycle.

Verbose (storage name and address) If verbose is enabled, the storage name, address (for multiple-fields storage elements) and value will be printed. If not enabled, only the address (for multiple-fields storage elements and only when the file output has more than one field) and the value are printed separated by a space.

Print initial value If enabled, the initial values of each storage element in the file output will be printed into the file.

Print file header If enabled, a file header is printed at the top of the output file.

Print cycle header If enabled, a cycle-header is printed each cycle when at least one file output needs to be printed.

Bytes per word (binary) In binary mode, values are written to the output file without any formatting. There are no spaces or newlines. This setting allows to specify how many bytes should be written per value. The number of bytes specified can be different from the type width, e.g. when the same output file is shared between an ISS using 24 bit integers and a native simulation where integers are 32 bit.

LSB first (binary) Specify the endianness of the bytes written in binary mode.

5.1.3 Go file output

The **Go file output** option logs the writes to all static processor storages needed to compare ISS simulations with an RTL simulation.

To add a Go file output, go to the **I/O** window and click on **Runtime I/O** → **Go file output**. A new settings window appears in which further details can be configured.

File name Choose the name of the file to which the output data should written or click the browse button to the right of the entry to select a file using a file-selection dialog.

Format Choose the HDL testbench output format that should be matched by the ISS. For more information on the HDL testbench output format, refer to [1, §5.1]. Possible choices are:

- VHDL testbench format
- VHDL testbench format (all hex)
- Verilog testbench format
- Verilog testbench format (all hex)
- Value change dump

Write This entry lets the user decide when the runtime file output will be written. The available options are:

When written When this entry is selected, the runtime file output will only be written when the ISS writes to the storage element the runtime file output is connected to.

When changed When this entry is selected, the runtime file output will be written when the written value is different from the previous value.

Print registers If enabled, register values are printed to the output file.

Print memories If enabled, memory values are printed to the output file.

Skip list Storages entered in this list will not be written to the output file.

5.1.4 `chess_report` output

The name of the file to write `chess_report` values to can be specified in the `chess_report` settings window.

5.1.5 Runtime graphical output

A runtime graphical output is similar to a runtime file output, but instead of writing the values on the connected storage element to a file, the values are plot in the **Graphical output** window. Graphical runtime outputs are used to display the evolution of the value of a storage element over a simulated time interval.

To add a runtime graphical output, go to the **I/O** window and click on **Runtime I/O** → **Graphical output** → **<new>**. Fill in the name of the storage. The drop-down menu can be used to select the storage name from a list. Then click **Ok**. A new settings window appears in which further details can be configured.

Address If the chosen storage contains multiple entries, the address can be entered here. The radix of the entered value can be chosen on the right. The address range entry can be done in octal, decimal or hexadecimal radix.

Data format Configure the value formatting of the output values.

Write This entry lets the user decide when the graphical output will be updated. The available options are:

When written When this entry is selected, the runtime graphical output will only be updated written when the ISS writes to the storage element the runtime graphical output is connected to.

Each cycle When this entry is selected, the runtime graphical output will be updated each cycle.

X-axis Following modes are available to configure the X axis:

- samples : the value on the X axis corresponds to the sample index.
- cycles : the value on the X axis corresponds to the cycle count.
- instructions : the value on the X axis corresponds to the instruction count.

Click the **Plot** button to plot the data to the **Graphical output** window.

Use the **Edit** and **Remove** buttons to edit or remove the settings for the graphical output.

5.2 Snapshot I/O

Snapshot I/O can be used whenever the ISS is halted. It can be used to initialize a range of memory fields, or the capture or display the current contents of a range of memory fields.

5.2.1 Snapshot file input

A snapshot file input is used to initialize a range of memory (or register file) fields, as input of the program being simulated.

To add a snapshot file input, go to the **I/O** window and click on **Snapshot I/O** → **File input** → **<new>**. Fill in the name of the storage. The drop-down menu can be used to select the storage name from a list. Then click **Ok**. A new settings window appears in which further details can be configured.

Storage The previously entered storage name is fixed. A storage name can't be edited for a given file input.

Address range If the chosen storage contains multiple entries, the address or address range to be initialized can be entered here. The radix of the entered value can be chosen on the right. The address range entry can be done in octal, decimal or hexadecimal radix.

Data format Configure the value formatting of the input values.

Data radix Select the radix of the data in the input file.

File name Choose the name of the file containing the input data or click the browse button to the right of the entry to select a file using a file-selection dialog.

Initial position Enter the position in the data-file were reading must start. When connecting the snapshot file input to the storage element, the snapshot file input will read the specified number of values from the file without writing them to the storage element.

Filter command The name of an executable used to convert the contents of the specified data file to a format understood by the ISS can be entered here or click the browse button to the right of the entry to select one using a file-selection dialog. When connecting the runtime file input to the storage element this executable will be called like this:

```
<filter executable> <data-file> <data-file>.checkers_flt
```

The ISS will use the contents of `<data-file>.checkers_flt` as data-input for the snapshot file input.

Input file contains structured data This option specifies whether the input file is structured or not (§5.3).

Bytes per word (binary) In binary mode, values are read from the input file without any formatting. There are no spaces or newlines. This setting allows to specify how many bytes should be read per value. The number of bytes specified can be different from the type width, e.g. when the same input file is shared between an ISS using 24 bit integers and a native simulation where integers are 32 bit.

LSB first (binary) Specify the endianness of the bytes read in binary mode.

Automatically load upon init/restart When this setting is enabled, the snapshot input will automatically be loaded into the ISS when the ISS is started or restarted. Else, the snapshot file input must be loaded manually.

Load when breakpoint hit Enable this setting to load the snapshot input when the breakpoint specified below is hit.

Breakpoint add Breakpoint to trigger the snapshot input.

To edit an existing snapshot file input, go to the **I/O** window and click on **Snapshot I/O** → **File input** → **<NAME>**. Then click the **Edit** button.

To load the snapshot input, click the **Load** button. The input file can be opened in the text editor using the **Open input file** button.

To remove a snapshot file input, go to the **I/O** window and click on **Snapshot I/O** → **File input** → **<NAME>**. Then click the **Remove** button.

5.2.2 Snapshot file output

A snapshot file output is used to write the contents of a range of memory (or register) fields to a file. This writing happens on user request, when the ISS is halted.

To add a snapshot file output, go to the **I/O** window and click on **Snapshot I/O** → **File output** → **<new>**. Fill in the name of the storage. The drop-down menu can be used to select the storage name from a list. Then click **Ok**. A new settings window appears in which further details can be configured.

Storage The previously entered storage name is fixed. A storage name can't be edited for a given file output.

Address range If the chosen storage contains multiple entries, the address or address range to be dumped can be entered here. The radix of the entered value can be chosen on the right. The address range entry can be done in octal, decimal or hexadecimal radix.

Data format Configure the value formatting of the output values.

Data radix Select the radix of the data in the output file.

Address radix Select the radix of the address in the output file.

Printf format string For the formats integer and unsigned, a custom format string can be defined here.

File name Choose the name of the file to which the output data should written or click the browse button to the right of the entry to select a file using a file-selection dialog.

Verbose (storage name and address) If verbose is enabled, the storage name, address (for multiple-fields storage elements) and value will be printed. If not enabled, only the address (for multiple-fields storage elements and only when the file output has more than one field) and the value are printed separated by a space.

Print file header If enabled, a file header is printed at the top of the output file.

Bytes per word (binary) In binary mode, values are written to the output file without any formatting. There are no spaces or newlines. This setting allows to specify how many bytes should be written per value. The number of bytes specified can be different from the type width, e.g. when the same output file is shared between an ISS using 24 bit integers and a native simulation where integers are 32 bit.

LSB first (binary) Specify the endianness of the bytes written in binary mode.

Dump when breakpoint hit Enable this setting to dump the snapshot output when the breakpoint specified below is hit.

Breakpoint add Breakpoint to trigger the snapshot output.

5.2.3 Snapshot graphical output

A snapshot graphical output is similar to a snapshot file output, but instead of writing to file, it displays the values in the **Graphical output** window.

To add a snapshot graphical output, go to the **I/O** window and click on **Snapshot I/O** → **Graphical output** → **<new>**. Fill in the name of the storage. The drop-down menu can be used to select the storage name from a list. Then click **Ok**. A new settings window appears in which further details can be configured.

Address range If the chosen storage contains multiple entries, the address or address range to be dumped can be entered here. The radix of the entered value can be chosen on the right. The address range entry can be done in octal, decimal or hexadecimal radix.

Data format Configure the value formatting of the output values.

Overwrite previously plotted data When enabled, the graphical output previously written will be overwritten. When disabled, the graphical output is appended to the previously written data.

Draw when breakpoint hit Enable this setting to draw the graphical output when the breakpoint specified below is hit.

Breakpoint add Breakpoint to trigger the graphical output.

Click the **Plot** button to plot the data to the **Graphical output** window.

Use the **Edit** and **Remove** buttons to edit or remove the settings for the graphical output.

5.3 Input file format

Input files can be *structured* or *unstructured*. Unstructured input files contain one value per line. These unstructured files are not loaded into memory, but instead, are read line per line, which reduces the memory usage by the ISS, in case of large input files. On the other hand, structured input files can contain repeat statements, and are processed by the ISS when added. So, structured input files are more compact when values are repeated several times at the cost of a higher memory usage by the ISS.

5.3.1 Structured input files

A structured input file has a simple syntax with which you can specify input sequences for storage elements. It is possible to repeat parts of the input file. The syntax to be used is:

Structured input file:
    ValueList

ValueList:
    Value
    ValueList Value

Value: one of
    integer
    hexadecimal integer
    string
    `<cycle-count>`
    `<no-value>`
    `(` *ValueList* `)` *repeat_count*

repeat_count:
    integer

The repeat count should always be a decimal integer value.

Values specified as strings should be placed between single or double quotes. These quotes are removed by the ISS, and only the resulting string is passed to the stream input method, selected for this file input. For example, when selecting the fractional format for file input, floating point literals should be specified between quotes :

```
"0.25" "0.55"
```

The string `<no-value>` is reserved. It is used as a place holder in the input file and will not be written to the storage element the input file is attached to. This allows for input files specifying values on specific cycles while keeping the value in the storage element in other cycles.

Using the reserved string `<cycle-count>` will write the cycle count to the storage attached to the file input.

5.3.2 Unstructured input files

An unstructured input file should contain one value per line. The ISS will read the unstructured input file line by line and pass each line unmodified to the input method specified for the storage element the file input is attached to. No repeat statements are allowed.

Different from Section 5.3.1, quotes should not be used around non-integer input values. For example fractional input values can be specified as follows :

```
0.25
0.55
3e-3
```

5.4 Interval list file format

An interval-list file contains timing information used for file-inputs. The file contains a list of intervals. When an interval ends, an event occurs, and a new value will be read from the data-file and this value will

be written to the storage element the data file and the interval-list file are attached to. The syntax to be used is:

IntervalList file:
    IntervalList

IntervalList:
    Interval
    IntervalList , Interval

Interval: one of
    RegularInterval
    WaitInterval
    RandomInterval
    EndlessRepeatSequence
    RepeatSequence

RegularInterval
    StepCount

WaitInterval
    StepCount !

RandomInterval
    MinimumStepCount : *MaximumStepCount*

EndlessRepeatSequence
    * { *IntervalList* }

RepeatSequence
    RepeatCount * { *IntervalList* }

StepCount
    integer

MinimumStepCount
    integer

MaximumStepCount
    integer

RepeatCount
    integer

The integer number specified for a *RegularInterval* specifies after how many steps an event must be generated. After generating the event, stepping continues with the next interval-list element.

The integer number specified for a *WaitInterval* specifies after how many steps to wait. When the specified number of wait step is expired, no event is generated.

The two numbers specified for a *RandomInterval* specify the minimum and maximum number of steps to be taken before generating an event. The actual number of steps will be randomly chosen each time a random interval is encountered and will be in the given range ([`MinimumStepCount` .. `MaximumStepCount`]).

The *EndlessRepeatSequence* will keep on repeating the interval-list it contains. The *RepeatSequence* will repeat the interval-list it contains the specified number of times.

When all intervals in the interval-file are processed, stepping starts again from the beginning of the file.

The following interval-list will generate event after 100, 200 and 300 steps. Then it will wait until cycle 500 and generate two more events after 700 and 1000 steps. After 1000 steps, the list ends and stepping will continue at the beginning again generating events after step 1100, 1200, 1300, wait until step 1500, ...

```
100, 100, 100, 200!, 200, 300
```

The following interval list will generate five events with 100 steps intervals and 10 event with 200 steps intervals:

```
5 * { 100 }, 10 * { 200 }
```

The following interval list will wait for 10000 cycles and afterwards generate an event every 1000 cycles:

```
10000!, * { 1000 }
```

The following interval list will also wait for 10000 cycles and afterwards generate an event every 990 to 1010 cycles:

```
10000!, * { 990 : 1010 }
```

Chapter 6

Profiling and execution tracing

Profile information is available both at the instruction and function level. Profiling tells you in which parts of the application program most cycles are spent. Functional unit, primitive operation and instruction class profiling can help in analysing your instruction set. Hazard profiling gives data about which hazards are triggered by your application. Also storage accesses can be profiled.

All profiling and execution tracing information can be accessed through the **Profiling** window. This window can be made visible from the **View** menu.

6.1 Collecting profile information

Profiling is only available in simulation mode. An ISS will collect profile information when the corresponding simulator settings are enabled (§8.4.4). For instruction-level profiling, the ISS records when instructions enter the profile stage. For function-level profiling, the ISS records when call/return instructions are executed (called execution tracing).

6.2 Instruction profiling

6.2.1 Textual representation

Profiling information can be viewed by right clicking on a line in the micro-code window and selecting **Configure breakpoint**. The pop-up window shows the execution count for that instruction in the **Execution count** field.

Profiling information can also be saved into a file or it can be seen as a bar chart. To save profiling information into a file, go to the **Profiling** window and click **Instructions** → **Instructios report**. The following settings for the generated report can be configured:

File name Enter the file name for the profile report or select an existing file using the **Select file** dialog.

Show user cycle count When this option is enabled, the user cycle count will be printed in the profile report.

Show source references Add source references to the micro-code.

Hide instruction bits When this option is enabled, the instruction bits will be hidden from the generated profile report.

Generate XML When this option is enabled, the instruction profile report will be generated in **XML** format.

Assembly width This setting limits the width of the assembly instructions printed in the profile report.

Press the **Create report** button to generate the profiling report. The report will be opened in a new editor window.

6.2.2 Graphical representation

To display profiling information in a bar chart, go to the **Profile** window and click **Instructions** → **Instructions chart**. A graph representing the cycle and instruction counts for each of the program memory locations in use will be displayed.

The drop-down box on top of the graph can be used to control the contents of the graph. By default, all functions are shown. Individual functions can be selected in the drop-down box.

6.2.3 Functional units report

To save profiling information for functional units into a file, go to the **Profiling** window and click **Instructions** → **Functional units report**. The following settings for the generated report can be configured:

File name Enter the file name for the profile report or select an existing file using the **Select file** dialog.

Include function details Enable this to include function details in the generated report.

Generate XML When this option is enabled, the instruction profile report will be generated in **XML** format.

6.2.4 Primitive operations report

To save profiling information for primitive operations into a file, go to the **Profiling** window and click **Instructions** → **Primitive operations report**. The following settings for the generated report can be configured:

File name Enter the file name for the profile report or select an existing file using the **Select file** dialog.

Include function details Enable this to include function details in the generated report.

Generate XML When this option is enabled, the instruction profile report will be generated in **XML** format.

6.2.5 Instruction classes report

To save profiling information for instruction classes into a file, go to the **Profiling** window and click **Instructions** → **Instruction classes report**. The following settings for the generated report can be configured:

File name Enter the file name for the profile report or select an existing file using the **Select file** dialog.

Include function details Enable this to include function details in the generated report.

Generate XML When this option is enabled, the instruction profile report will be generated in **XML** format.

6.2.6 Hazards report

To save profiling information for hazards into a file, go to the **Profiling** window and click **Instructions** → **Hazards report**. The following settings for the generated report can be configured:

File name Enter the file name for the profile report or select an existing file using the **Select file** dialog.

Include function details Enable this to include function details in the generated report.

Generate XML When this option is enabled, the instruction profile report will be generated in **XML** format.

To see data about software stalls, a class name needs to be speficied with the software stall hazard as described in the *nML*.

6.2.7 *nML* coverage report

To save *nML* coverage information into a file, go to the **Profiling** window and click **Instructions** → **nML coverage report**. RISK is used to calculate the *nML* coverage based on the simulation. The following settings for the generated report can be configured:

File name Enter the file name for the profile report or select an existing file using the **Select file** dialog.

6.3 Function profiling and execution tracing

6.3.1 Execution tracing

Logging

Execution tracing will log all call-to-subroutine and return-from-subroutine commands. The logged information can be used to show the current execution point (call tree) or to show a complete execution trace.

Execution tracing must be enabled in the CHECKERS configuration file. See section 8.4.4 for more information on how to do this.

Execution trace

The execution trace shows the call and return history of the complete simulation since startup or since the last reset. This can be used to investigate how the program executed, which functions were called and how often a function was called.

Each row represents one entry of the execution-trace log. In this dialog all entries of this log are shown. This allows to user to reconstruct the execution path of the program being simulated.

The execution trace is not available during on-chip debugging.

Execution point

The execution point shows the call tree for the current simulation position. It shows which function is currently executing and from which function it was called. Each row represents one log entry of the execution-trace log.

The execution point is not available during on-chip debugging.

Back trace

The back trace shows the same information as the execution point. However, back tracing uses stack unwinding instead of execution tracing. This makes back tracing work in the OCD where execution tracing isn't possible.

6.3.2 Textual represenation of function profiling

To save function profiling into a file, go to the **Profiling** window and click **Functions** → **Report**. A dialog opens in which following settings for the generated report can be configured.

File name Enter the file name for the profile report or select an existing file using the **Select file** dialog.

Info per call When enabled, the report will list the number of cycles every call did take. This can make the report very big.

Caller/callee info When enabled, an additional report will be generated listing every caller and callee for each function.

Entry PC Define the entry point of the execution. This is the start of the main function.

Generate XML When this option is enabled, the instruction profile report will be generated in **XML** format.

Press the **Create report** button to generate the profiling report. The report will be opened in a new editor window.

6.3.3 Graphical representation of function profiling

A number of charts displaying function profiling information are available from **Functions** → **Charts** in the **Profiling** window:

Total func time Barchart showing total time spent in a function

Total func+desc time Barchart showing total time spent in a function and its descendants.

Number of calls Barchart showing the number of time a funtion was called.

Min/Avg/Max func time Barchart showing minimum, average and maximum time spent in a function.

[Min/Avg/Max func+desc time Barchart showing minimum, average and maximum time spent in a function and its descendants.

6.4 Storage profiling

Storage profiling gives information about how many times a certain storage file or a certain storage field is read or written. The ISS will collect storage profiling information if this is enabled in the CHECKERS configuration file. Each time a storage element field is read or written, its read or write count is incremented.

When a storage element is written with the same value as the one it already contains, the write count is still incremented.

Storage profile information is collected for the storages as used in the nML. If a read or write operation uses a range alias, a record alias or a record register/memory, the read or write count of the alias or record is incremented. To obtain the total read or write count of the physical storage elements, the read or write counts of the physical storage elements and its aliases or of the records it is part of must be added.

To enable the collection of storage access profiling, set the **Storage access history logging** setting in any of the **Storages** dialogs to **On**.

6.4.1 Storage profiling access summary

The storage profiling accesses allows to generate a report containing a summary of the number of accesses per storage. No information about the cycle count the accesses occurred in is included.

Report

To save an access report, go to the **Profiling** window and click **Storages** → **Accesses** → **Report**. A dialog opens in which following settings for the generated report can be configured.

Storage access history logging This entry is identical for all **Storages** dialogs. It should be set to **On** to collect storage access history.

File name Enter the file name for the profile report or select an existing file using the **Select file** dialog.

Include function details Enable this to include function details in the generated report. When this setting is disabled, the following three settings are inactive.

Hide instruction bits Enable this setting to hide reporting of the instruction bits.

Include field details Enable this setting to include details of the storage fields in the generated report. When this setting is disabled, no reporting for individual storage fields is included.

Include function summary Enable this setting to include a summary per function in the generated report.

Generate XML When this option is enabled, the instruction profile report will be generated in **XML** format.

Press the **Create report** button to generate the report. The report will be opened in a new editor window.

6.4.2 Storage profiling access history

The storage profiling access history allows to generate a report containing a detailed history of all storage accesses, including the cycle information on the accesses.

Report

To save an access report, go to the **Profiling** window and click **Storages** → **Access history** → **Report**. A dialog opens in which following settings for the generated report can be configured.

Storage access history logging This entry is identical for all **Storages** dialogs. It should be set to **On** to collect storage access history.

File name Enter the file name for the profile report or select an existing file using the **Select file** dialog.

Storage Select the storage for which to generate the access history report.

Address range Select the address or address range for which to generate the access history report.

Generate XML When this option is enabled, the instruction profile report will be generated in **XML** format.

Press the **Create report** button to generate the report. The report will be opened in a new editor window.

Chart

To show the access history in a chart, go to the **Profiling** window and click **Storages** → **Access history** → **Chart**. A dialog opens in which following settings can be configured.

Storage access history logging This entry is identical for all **Storages** dialogs. It should be set to **On** to collect storage access history.

Storage Select the storage for which to generate the access history chart.

Address range Select the address or address range for which to generate the access history chart.

Access per: Configure whether the X axis shows the cycle count or the program counter value.

Chapter 7

Running a simulation in batch mode

This chapter explains how the ISS can be used from the command line and how scripts can be used to drive batch simulations. The console is only available in text mode and not in GUI mode.

To start an ISS in textual mode, use the `-T` command line argument.

```
<iss-executable> -T
```

For batch simulations, multiple commands can be placed in a file. This file can be specified when starting the ISS using the `-t` argument:

```
<iss-executable> -T [-t <script.tcl>]
```

It's also possible to pass arguments for the script to the ISS like this:

```
<iss-executable> -T [-t "<script.tcl> [arg ..]"]
```

Before the ISS starts executing the commands in the file, the arguments are placed in the interval variable **::iss::tcl_script_args**. You can use this variable to access the script parameters.

The file can also be sourced from within the textual command prompt like this:

```
% source <script.tcl>
```

Arguments can't be passed when sourcing a file. Variables used by the script can be created before sourcing the file.

Information about the commands you can use from the command line or in batch mode can be found in [2, §7.6.2].

7.1 Black box simulation

It is possible to embed the simulator or use the batch mode without requiring a license. This is called black box simulation. Functionality is limited in black box simulation. After creating an ISS with the `iss::create` statement (or equivalent C statement) you can load a program with following restrictions:

- no *Dwarf* debugging information
- no disassembling
- no hosted I/O
- no program arguments (`argc/argv`)
- no initialization of program counter
- no initialization of stack pointer

Stepping through the program is possible, but only micro-code stepping is allowed.

Reading and writing of memories and I/O ports is allowed. All other functionality will require a `tctsim` license.

When building an ISS for use as black box simulator, make sure to disable:

- hosted I/O
- program arguments (`argc/argv`)
- runtime file input
- runtime file output
- runtime graphical output
- breakpoints
- watchpoints
- all checks (address ranges, write conflicts, stack range, uninitialized transitories, reserved instructions)
- coloring of storages when read or written
- instruction, operation and storage profiling

Chapter 8

Simulator settings in CHESSDE

Simulators are configured and built using CHESSDE, as explained in [3, §2.10]. This chapter contains an overview of all project settings which are relevant for simulators.

General processor settings (like nML settings), specified in CHESSDE, are automatically passed to the simulators built. In addition, it is possible to specify common simulator settings at the processor level, which are then inherited by every simulator project, created to build a specific simulator.

The toplevel for simulator related settings is the **Simulator generation** option level. There are three sublevels, one for each type of simulators that can be built: **Cycle accurate**, **Instruction accurate**, and **Debug client**. These types are abbreviated respectively as CA, IA, DB. All these settings are saved in the `<processor>.prx` file.

At the project level, when starting a new simulator project (**File** → **New** → **Simulator**), you first have to choose the simulator type, and the **Simulator** project settings then inherit from the selected simulator type.

The different simulator settings are subdivided in different option groups, which are discussed in the following sections. Dependent on the simulator type, the option groups may contain different options.

8.1 General settings

- **Simulator name** Name of ISS, on *Windows* `.exe` is added to this name.
- **Work directory** The directory in which all intermediate files are generated.
- **Generate primitives with PDG** Use the PDG tool to generate the primitive functions from the `<processor>.p` file. Refer to [4] for information on how to create the `<processor>.p` file.
- **Generate controller with PDG** Use the PDG tool to generate the controller from the `<processor>_pcu.p` file. Refer to [4] for information on how to create the `<processor>_pcu.p` file.
- **Extra options** Not every possible option of CHECKERS has been added to CHESSDE. Rarely used options can be specified here in text format (same format as used in the old CHECKERS configuration files). See §8.7 for an overview.

8.2 Model

8.2.1 Controller

Controller settings for cycle accurate mode

These settings are only available when not using PDG for the controller.

- **Controller header file** File containing the C++ class implementing the processor controller.
- **Controller class name** Name of the C++ class implementing the processor controller.
- **Control operations before user-issue** Forces all control operations to be executed before calling the user-issue function so this function can test all controller signals.
- **Allow register reads after user-next_pc/issue** When these controller function are not writing to registers, register reads are allowed after calling these function. This allows for more flexible sorting of the different ISG operations in an instruction for execution.
- **Transitories read by user-issue** List of all transitories read by the user-issue controller function. Specifying a transitory in this list will make sure it is written before the user-issue function is called.
- **Transitories written by user-issue** List of all transitories written by the user-issue controller function. Specifying a transitory in this list will make sure it is only read after the user-issue function is called.
- **Transitories read by user-next-pc** List of all transitories read by the user-next-pc controller function. Specifying a transitory in this list will make sure it is written before the user-next-pc function is called.
- **Transitories written by user-next-pc** List of all transitories written by the user-next-pc controller function. Specifying a transitory in this list will make sure it is only read after the user-next-pc function is called.
- **Registers written by user-issue** List of all registers written by the user-issue controller function. Specifying a register in this list will make sure it is only read after the user-issue function is called.
- **Registers written by user-next-pc** List of all registers written by the user-next-pc controller function. Specifying a register in this list will make sure it is only read after the user-next-pc function is called.

Controller settings for instruction accurate mode

- **Controller header file** File containing the C++ functions implementing the processor controller.
- **Transitories read by user-next-pc** List of all transitories read by the user-next-pc controller function. Specifying a transitory in this list will make sure it is written before the user-next-pc function is called.
- **Transitories written by user-next-pc** List of all transitories written by the user-next-pc controller function. Specifying a transitory in this list will make sure it is only read after the user-next-pc function is called.
- **Dedicated end-of-loop check** Lets the ISS use a dedicated end-of-loop check function which is only called for the last instruction in a hardware do-loop. This optimizes the end-of-loop checking which otherwise has to be done every instruction in the user-next-pc function.
- **Include always actions** Also simulate always actions. Often always actions are used to implement part of the controller functionality. In compiled code instruction accurate mode this is better done in the user-next-pc or dedicated end-of-loop functions.
- **Interpret as next incremental instruction** List of transitories which when read will not contain the value written to them but rather the address of the instruction following the instruction being executed and eventual delay slots. This can be used in call or hardware do-loop instructions to save the return address or the loop start address.

Controller settings for debug client

- **Debug client header file** File containing the C++ class implementing the debugger.
- **Debug client class name** Name of the C++ class implementing the debugger.

8.2.2 Conversions

- **Generate convert functions** Let CHECKERS generate record (alias) conversion functions.
- ↪ **Excluded types** Do not generate record (alias) conversion functions for types listed here.

  When having a record storage in nML of type t, containing storage elements of type a, b, and c, the two corresponding CHECKERS convert functions have following interface :

```
inline void convert(t src, a& dst0, b& dst1, c& dst2) { ... }
inline void convert(a src0, b src1, c src2, t& dst) { ... }
```

- **Generate primitive conversions** Let CHECKERS generate primitive conversion functions.
- ↪ **Excluded types** Do not generate primitive conversion functions for types listed here.

8.2.3 Optimization

Optimization settings for cycle accurate mode

- **Cache decoded instructions** Speed up execution by caching decode instructions.
- **Instruction written at runtime** When this option is enable, the ISS will check if writes to program memory overwrite an already decoded (and cached) instrucion. If an instruction is overwritten, its cache is cleared and it will have to be decoded again the next time it is executed.
- **Merge ISG operations** Merge equal enabled operations to minimize the function call overhead when executing an instruction.
- ↪ **merge count** Maximum number of operations to merge.

Optimization settings for instruction accurate mode

- **Constant transitory types** List of data types of immediate constants. Instructions using immediate constants of types listed here can be reused for different constant values because the constant values are not compiled into the instruction but placed in a map based on the address of the instruction.

8.2.4 Pipeline

- **Enable killing of issued instructions** Prepare simulator for killing of instructions.
- **Wait state mode** Sets wait state mode.

  Possible values are:

  - **None** No wait states.
  - **Nop** In wait-state `nop` mode, the cycle in which the wait-signal is raised is finished, including writing results to static storage, and no actions are executed in the wait-cycle.
  - **Repeat** In `repeat` mode, all actions are repeated in the wait-cycle but no results are written to static storage as long as the wait-signal is high.
- **Shift fetch pipeline in wait-state** Configuration option used to enable the shifting of the fetch-pipeline in wait-states. When the fetch-pipeline is shifted in wait-states, started program memory fetched can continue.
- **Access of multi-word consts in fetch stage** Generate ISS where instruction fetch and `user_issue` is done before the multi-word constant generation. Enable this option for processors that do single-word instruction fetching where secondary instruction words containing immediates are already accessed in the instruction fetch stage.

- **Maximum decode index** Configuration option used to specify the maximum decode index value used when issuing instructions. Decode indexes are used together with instruction addresses to store decoded instruction in a cache. When multiple instructions can have the same address (e.g. after compaction), a decode index can be used to distinguish them.
- **Return address offset to stored value** Offset to be added to return address as stored in link register or on the stack to obtain the actual return address. This value is used when doing stack unwinding or reading variable info in stack frames of callers.

8.2.5 Memory interface

Memory interface settings for cycle accurate mode

- **Generate memory interface** Enables the generation of memory interfaces. More information about memory interfaces can be found in [2].
- ↪ **selected memories** Generate a memory interface for the specified memories.
- **Generate acknowledge signals** Add acknowledge signals to the memory interface.
- ↪ **selected memories** Add acknowledge signals to the memory interface for the specified memories.
- **Generate template functions** Generate templates for the required memory interface functions.
- **Vector pointers** Use vector pointers to address vector memories (one pointer per vector element iso one common pointer).

Memory interface for instruction accurate mode

- **Asynchronous get/put functions** List of memories for which to translate the load/store accesses into calls to get/put functions. More information on asynchronous get/put functions can be found in [2].
- **Vector pointers** Use vector pointers to address vector memories (one pointer per vector element iso one common pointer).

8.3 SystemC

- **Generate SystemC interface**. Generate a *SystemC* wrapper class for the ISS. Check [2] for more information about the generated *SystemC* wrapper.
- ↪ **with debug ports** Add debug interface to generated *SystemC* wrapper class.
- ↪ **with GUI** Add GUI support to generated *SystemC* wrapper class.
- **SystemC header file** Name of *SystemC* header file.
- **Number of instruction words in interface** The maximum number of instructions fetched simultaneously.
- **Generate ModelSim style testbench** Generate a ModelSim style testbench. When generating a plain testbench, an `sc_main` function will be generated. When generating a ModelSim style testbench, a `top` class containing all other *SystemC* objects will be generated.
- **Compile generated SystemC files** Compile all generated *SystemC* files. This option is disabled when the *SystemC* code has to be compiled with another compiler (e.g. when using ModelSim).
- **Active low reset** Generate a *SystemC* wrapper class with active low reset.
- **Clock port** The name of the clock port of the generated *SystemC* wrapper class.
- **Reset port** The name of the reset port of the generated *SystemC* wrapper class.

- **Break request port** The name of the break-request output port of the generated *SystemC* wrapper class.
- **Debug request port** Name of the debug request input-port of the generated *SystemC* wrapper class. If not specified, the name of this port is taken from the *nML* property `ocd_request`.

8.4 Functionality

8.4.1 User interface

- **Color storages when read or written** Let ISS color storage which are read (blue), written (red) or read and written (purple).
- ↪ **Selected storages** Color selected storages only.
- **PC history** Indication if ISS supports PC history.
- **Instruction history** Indication if ISS supports instruction history.
- **Cycle/instruction count** Indication if ISS supports cycle and instruction counting.
- **Source stepping** Indication if ISS supports source stepping.
- **Do not initialize PC and SP** Do not initialize the program counter and the stack pointer(s) after loading a program into the ISS.
- **Multi core debugger** Generate a debugger capable of connecting to multiple cores.

User interface settings for cycle accurate mode

- **Pipeline information** Indication if ISS has pipeline information.
- ↪ **Pipeline depth** If not specified or if the number of specified stages is smaller than the pipeline depth as specified in the *nML*, the ISS will use the pipeline depth as specified in the *nML*.
- ↪ **Pipeline colors** Colors for the different pipeline stages. First color is for stage zero, second for stage one, ...
- **Focus stage** Number of the stage which will be used to position the micro code and source code window. The instruction which is in the stage specified here will always be visible.

8.4.2 I/O

- **Enable hosted I/O via <stdio.h>** Enables hosted I/O. Check chapter A for more information.
- **Load program arguments (argc, argv)** Make it possible to specify arguments for the application program being simulated. These arguments will be passe to the program's main function as `int argc` and `char ** argv`.
- **File output** Enable runtime file output in the ISS.
- ↪ **Selected storages** Only enable runtime file output in the ISS for the specified storages.
- **File input** Enable runtime file input.
- ↪ **Selected storages** Only enable runtime file input in the ISS for the specified storages.
- ↪ **Allow cycle-based file input** Provide possibility to specify cycle based runtime file input in the ISS.
- **Runtime graphical output** Enable runtime graphical output in the ISS.
- ↪ **Selected storages** Only enable runtime graphical output in the ISS for the specified storages.

- **VCD dump for comparison with RTL simulation** Make runtime output suitable for comparisons with results of RTL simulation.
- ↪ **Detect when value changes, not just when written** When doing file output, make it possible to only write the values when they actually change.

I/O settings for instruction accurate mode

- **Hosted Calls** Specify a list of functions which have to be executed on the host rather than being executed in the ISS. Check section A.2 for more information.

8.4.3 Break/Watch points

- **Breakpoints** Enable micro code and source code breakpoints.
- ↪ **Support hit-count** Indication if hit-count is supported in the ISS.
- ↪ **Support exporting of breakpoints** Indication if exporting of breakpoints (breakpoint on one core will also stop all other cores in a multi core simulation) is supported in the ISS.
- **Watchpoints** Enable watchpoints.
- ↪ **Selected storages** Only enable watchpoints in the ISS for the specified storages.

Break/Watch point settings for cycle accurate mode

- ↪ **Breakpoint stage (< 0 = decode-stage)** The address of the instruction in the stage specified here will be use to check for breakpoints. This option is now deprecated as the breakpoint and focus stage are now set with property `breakpoint_focus_stage` in the <`processor`>`_chess.h` file.

Break/Watch point settings for debug client

- ↪ **Support software breakpoints** Indication if software breakpoints are supported in the debug client.

8.4.4 Profiling

Profiling settings for cycle and instruction accurate mode

- **Profile instructions** Enable instruction profiling.
- **Trace execution point (function level profiling)** Enable execution tracing of call, return from subroutine, interrupt and return from interrupt instruction to generate execution point, execution trace and function level profiling reports.

Profiling settings for cycle accurate mode

- ↪ **at stage** Stage in which to update the profile information of an instruction. This is only relevant when issued instructions can be killed from the instruction pipeline. For example, when killing an instruction in the decode stage using the `kill_instr()` PDG intrinsic, the profile stage should be set to a stage after the decode stage.
- **Profile operations** Enable ISG operation profiling.
- **Profile hazards** Enable hazard (software stalls, hardware stalls, bypasses) profiling.
- **Profile storages** Enable storage profiling.
- ↪ **Selected storages** Enable storage profiling for specified storages.

8.4.5 Checks

The different checks can be reported as errors or as warnings.

Checks settings for cycle and instruction accurate mode

- **Check address ranges** Enable address range and address alignments checking.
- **Align addresses** Align addresses before using them. checking.
- **Check write conflicts** Configuration option used to enable runtime conflict checking in the ISS. Conflicts occur when any of the following is true:
  - The same instruction writes different values to the same storage field in the same cycle and the strength of the first write operation is larger than or equal to the strength of the second write operation.
  - Different instructions write different values to the same storage field in the same cycle.
- **Check stack range** Enable stack range check. The value of the stack pointer register should not exceed the stack range as specified during linking.
- **Check uninitialized transitories** Let ISS report operations reading from uninitialized transitories.

Checks settings for cycle accurate mode

- ↪ **Only report value conflicts** Only report a write conflict when the values being written are different.
- ↪ **Report strength resolved conflicts** Report write conflicts which were resolved by writes having a different strength.
- ↪ **Check conflicts on vector elements** Check write conflict on vector elements rather than on complete vector fields.
- **Check reserved instructions** Let ISS report instructions without enablings.

8.5 Host compilation

- **Generate simulator as** Specify what the output of compiling the generated ISS should be:
  - **Executable**
  - **Library**
  - **Dll**
- **C++ file extension** The file extension for generated C++ files.
- **Macro definitions (-D)** Enter pre-processor definitions here (do not include the `-D`).
- **Include path (-I)** Enter pre-processor include paths (do not include the `-I`).
- **Makefile name** Name of generated make-file.
- **Do not generate a main function** If enabled, no `main` function will be generated. This is useful when generating a library or a DLL for use with another main function.
- **Only use C++ API, no Tcl API** If enabled, the *Tcl/tk* API will not be added to the generated ISS.

8.5.1 Linux

- **Host compiler command** Path to the compiler to be used to compile the generated files.
- **Extra compilation options (optimized)** Extra options to be used to compile the generated files with optimization.
- **Extra compilation options (non critical files)** Extra options to be used to compile the non critical generated files without optimization.
- **Extra link options** Enter additional link options here.

8.5.2 Windows

- **Host compiler** Set compiler type to be used to compile the generated ISS.
- **Extra compilation options (optimized)** Extra options to be used to compile the generated files with optimization.
- **Extra compilation options (non critical files)** Extra options to be used to compile the non critical generated files without optimization.
- **Extra link options** Enter additional link options here.
- **Link as console application** Link as console application rather than as Windows application.

8.5.3 Extra files/dirs

- **File included in UI-core class** File to be included in the generated UI-core class. This file can be use to add extra member variables and/or member functions to the generated UI-core.
- **File included in processor class** File to be included in the generated processor class. This file can be use to add extra member variables and/or member functions to the generated processor model.
- **Include files** List of files to be included in each generated file.
- **C++ files** List of C++ files to be compiled along with the generated files.
- **Libraries** Libraries to be added to the link command. Libraries with an extension (e.g. `.a` or `.lib`), or starting with `-l`, are used as specified. Other libraries get `-l` prepended on Unix, and `.lib` appended on Windows.
- **Library directories** List of library search directories.
- **Tcl scripts to run in ISS** List of Tcl scripts to run after starting the ISS.

8.6 User functions

- **Print functions.** Additional I/O functions for primitive processor types, to print and read values in a user-defined format. The format of this option is $t = f$ with the t primitive type and f the function name. For example, when having implemented following two functions for a primitive processor type `A` :

  ```
  ostream& my_format(ostream& os, A  a);  // writes "a" to "os"
  istream& my_format(istream& is, A& a);  // reads "a" from "is"
  ```

  To be able to use this function in the **Register** window (§4.1.2), you have to specify following option :

  ```
  A=my_format
  ```

- **Processor init-function.** Function to be called when creating an object of the generated processor model class. This happens when starting or resetting the ISS. The specified function will take a pointer to the generated processor model class as argument.

- **Processor fini-function.** Function to be called when destroying an object of the generated processor model class. This happens when closing or resetting the ISS. The specified function will take a pointer to the generated processor model class as argument.
- **Processor post-step function.** Function to be called after simulating a cycle. The specified function will take a pointer to the generated processor model class as argument.
- **Processor pre-step function.** Function to be called before simulating a cycle. The specified function will take a pointer to the generated processor model class as argument.
- **Processor pre-update function.** Function to be called before calling the controller update function when simulating a cycle. The specified function will take a pointer to the generated processor model class as argument.
- **UI-core init-function.** Function to be called when creating a UI-core object. The specified function will take a pointer to the generated UI-core processor model class, and integer and an array of `char` pointers as argument.
- **UI-core post-simulate function.** Function to be called after simulating a series of cycles. The specified function will take a pointer to the generated UI-core processor model class as argument.
- **UI-core pre-simulate function.** Function to be called before simulating a series of cycles. The specified function will take a pointer to the generated UI-core processor model class as argument
- **Hook in Tcl processor function.** Function which is called when creating an ISS. This function will be called from the *Tcl/tk* processor function. User command line arguments (specified with `-u` are passed to this function. The function is typically used to perform some *Tcl/tk* related actions like registering C functions as *Tcl/tk* commands or binding C and *Tcl/tk* variables. The specified function will take a pointer to the generated UI-core processor model class, an integer containing the number of user arguments and an array of character pointers with the users arguments as arguments. The function must return an integer indicating success or failure of the function.

8.7 Extra options

The syntax for the extra options is:

Extra_options:
    List_of_configuration_options

List_of_configuration_options: one of
    Configuration_option
    List_of_configuration_options Configuration_option

Configuration_option: one of
    List_option
    List_of_mappings_option
    List_of_lists_option
    String_option
    Integer_option
    Boolean_option

List_option:
    Option_name = *List_of_values*

List_of_mappings_option:
    Option_name = *List_of_mappings*

List_of_lists_option:
Option_name = *List_of_lists*

String_option:
Option_name = *Identifier*

Integer_option:
Option_name = *Integer*

Boolean_option:
Option_name

Port_map_option:
Option_name : *List_of_Port_mappings*

List_of_values: one of
Identifier
List_of_values Identifier

List_of_mappings: one of
Identifier = *Identifier*
List_of_mappings Identifier = *Identifier*

List_of_lists: one of
(*List_of_values*)
List_of_lists (*List_of_values*)

List_of_Port_mappings: one of
Identifier -> *Identifier*
List_of_Port_mappings Identifier -> *Identifier*

Identifier: one of
*[a-zA-Z.][/a-zA-Z0-9_.+-]**
"*[^]*"
{*[{]*}

Integer:
[-+][0-9]+*

Extra options are:

`ClassName = <identifier>` Configuration option used to specify the name of the generated processor class.

`DefaultGuiType` Specify how ISS when used as stand-alone application should start by default: `x` to start with GUI, `T` to start with textual interface or `c` to start with console window.

`DisableHazards = <list of values>` Disable the hazard rules of the specified classes.

`DisableInstructions = <list of values>` Disable the instructions of the specified classes.

`InitAllStorages = <integer>` Configuration option used to instruct CHECKERS to generate an ISS which initializes all storage. By default, only storage initialized with a `hw_init` statement are initialized. All storages will be initialized with zero.

`LinkAddressOffsetInSystemArea` **= <integer>** Used to set the offset of the return address in system area.

`load : <list of port mappings>` Configuration option used to specify alternative port names for port used in load oprations for the generated *SystemC* wrapper class. Check [2] for more information about the generated *SystemC* wrapper.

When generating a *SystemC* wrapper class with CHECKERS, CHECKERS will create a file `Mdl_<processor>_systemc_interface.txt` containing the port mapping configuration statements as used in that run of CHECKERS. These configuration statements can be pasted in the extra options and updated as required. Make sure not to change the CHECKERS name of the first port mapping. It identifies the load or store operations for which the name mappings must be applied.

`OffsetToStoredReturnAddress = <integer>` Configuration option used to specify the offset of the effective return address to be added to the address as stored in the link register or hardware stack.

`store : <list of port mappings>` Configuration option used to specify alternative port names for port used in store oprations for the generated *SystemC* wrapper class. Check [2] for more information about the generated *SystemC* wrapper.

When generating a *SystemC* wrapper class with CHECKERS, CHECKERS will create a file `Mdl_<processor>_systemc_interface.txt` containing the port mapping configuration statements as used in that run of CHECKERS. These configuration statements can be pasted in the CHECKERS configuration file and updated as required. Make sure not to change the CHECKERS name of the first port mapping. It identifies the load or store operations for which the name mappings must be applied.

`UserPdcPutFunction = <identifier>` Function called for each memory location written when loading the program. Check [2] for more information.

`UserInitTcl = <identifier>` Function called after creating the embedded *Tcl/tk* interpreter but before loading any ISS specific code in it. Check [2] for more information.

Chapter 9

Processor modeling for simulation and debugging

This chapter summarizes the required processor modeling for simulation and debug client generation.

9.1 Cycle accurate mode

In cycle-accurate mode, no ISS-specific processor modeling is required. CHECKERS uses following central processor description files :

- The primitive processor header file (`<processor>.h`, [5]).
- The nML description (`<processor>.n`, [6]).
- The PDG description of the primitive functions used in the nML actions (`<processor>.p`, [4]).
- The PDG description of the processor controller (`<processor>_pcu.p`, [4]).

The PDG descriptions are also used by the RTL generator GO [1].

9.2 Instruction accurate mode

The instruction-accurate simulation mode uses compiled-code simulation. To run a simulation in instruction-accurate mode, first, you need an instruction-accurate ISS, which is built via CHESSDE, and secondly, a DLL is needed, which contains the processed application program. This DLL is automatically built by CHESSDE, after enabling the option **Project→Add simulator DLL** in a compilation project.

In the instruction-accurate mode, an ISS-specific controller model must be provided by the user. Indeed, the cycle-accurate PDG controller description cannot be reused, as this description relies on the pipelined execution of the consecutive instructions.

9.2.1 Instruction-accurate controller model

The controller model is specified in a C++ header file, the name of which is specified in the CHESSDE controller settings (§8.2.1).

Main controller function. The main controller function looks like this (with <PC> the name of the program counter in nML) :

```
void <processor>_user_next_pc(<processor>*          mdl,
                              unsigned              next_pc,
                              unsigned              number_of_words,
                              Checkers_next_pc_type next_pc_type)
{
   mdl-><PC> = next_pc;
}
```

The `mdl` argument is a pointer to the processor model. `next_pc` contains the address of the next instruction to be simulated, as calculated by the current instruction. Typically, it is sufficient to simply assign `next_pc` to the program counter.

The two extra arguments can be needed when modeling additional controller behavior like interrupts. The `number_of_words` argument contains the number of words of the current instruction. `next_pc_type` gives an indication of how the `next_pc` value was calculated by the current instruction. Possible values are:

- `npt_increment`. The `next_pc` value is the next address on the increment path.
- `npt_jump`. The `next_pc` value is a jump target (of a jump, call, or return instruction).
- `npt_loop_end`. The `next_pc` value is the address of the hardware loop start, and was obtained as result of a zero-overhead end-of-loop check (see further).
- `npt_cntrl`. The `next_pc` value was obtained as result of a operation marked with the controller property, but not a jump, call, return from subroutine, return from interrupt, nor hardware loop-end.
- `npt_delay_slot`. The `next_pc` value is the address of an instruction in the delay slots of a previously executed controller instruction.

Reading the program counter. When an instruction is reading the program counter in its action attribute, due to instruction pipelining, typically, the address of a next instruction will be read out. In instruction-accurate mode, CHECKERS can only rely on the `chess_pc_offset()` instruction property [5], to determine which PC value must be read. So, in instruction-accurate mode, CHECKERS requires that every instruction reading out the program counter has an explicit `chess_pc_offset()` property.

Possibly, the computation of the function return address happens in the controller. For example, the action of a call instruction can look as follows in nML :

```
trn pc_next<addr>;
hw_init pc_incr = 0;  // transitory set in PDG controller description

opn call(tgt : c_16)
{
   action {
     stage ID:
       call(pc_w = tgt);
       LR = lr_w = pc_incr;  // instruction after the delay slot
   }
   image : tgt, delay_slots(1);
}
```

To simulate this in instruction-accurate mode, you have to tell CHECKERS to interpret the `pc_incr` transitory as the address of the next instruction on the increment path, after the current instruction and its delay slots. This can be done via the **Interpret as next incremental instruction** option in the CHESSDE controller settings (§8.2.1).

Zero-overhead loop check. In case the processor contains zero-overhead or hardware loop instructions, you must explicitly check for end-of-loop instructions in the instruction-accurate controller model.

To speed up the end-of-loop check of zero-overhead or hardware loops, it is possible to provide a dedicated end-of-loop check function. When you code the end-of-loop check directly in the main controller function, then this check is performed every cycle. On the other hand, in case of a dedicated end-of-loop check function, CHECKERS will only call this function for the last instruction of a zero-overhead loop. So, you are sure that the end-of-loop condition is satisfied, and you only have to check whether the loop is finished or not. For this purpose, the end-of-loop instructions are marked by the compiler (cf. the `.eol` statements in assembly code).

When providing this function, you have to enable the **Dedicated end-of-loop check** option in the CHESSDE controller settings (§8.2.1). This function looks as follows (the function body is copied from the Base example processor) :

```
inline unsigned <processor>_loop_end(<processor>* mdl, unsigned nw)
// "nw" is number of words of current instruction
{
   unsigned next_pc;
   int lf = mdl->LF.value();          // pointer to loop stack
   int lc = mdl->LC[lf].value();      // loop count
   if (lc == 1) {
      mdl->LF = lf - 1;               // pop loop stack
      next_pc = mdl->PC + nw;         // exit the loop
   }
   else {
      mdl->LC[lf] = lc - 1;
      next_pc = mdl->LS[lf].value();  // jump back
   }
   return next_pc;
}
```

In instruction-accurate mode, the program counter in the generated processor class has type `unsigned`. When referring to other registers declared in nML, you have to use the `value()` member to obtain the integer value.

9.2.2 Reusing constants

In instruction-accurate compiled-code simulation, every different instruction of the application program is translated into a separate C++ function. To reduce the number of generated C++ functions, and to reduce compile time, it is possible to reuse the same C++ function for instructions that only differ in the value of immediate parameters. This is mainly useful for longer immediates. To enable this reuse, you have to specify the data type of the corresponding constant parameters in nML, in the CHESSDE optimization settings (§8.2.3).

9.3 Debug client

CHECKERS generates a processor-specific debug client in terms of a processor-independent abstract debug interface. This abstract interface, called `Checkers_debugger`, is declared in the central CHECKERS include file, called `checkers_debugger.h`.

For every different processor, the user must provide an implementation of this abstract `Checkers_debugger` class. An example implementation is available for the Base example processor, interfacing with the Processor Debug Controller (PDC) unit generated by GO, via a JTAG link. It consists of following three classes:

- `pdc_commands`. This class is an interface around the `jtag_socket` library, driving the parallel or Amontec USB cable. It provides the basic PDC commands: register/memory access, reset/step/resume/request commands, and breakpoint commands.
- `Checkers_pdc_interface`. This class derived from `Checkers_debugger`, implements the `Checkers_debugger` interface in terms of the basic PDC commands provided in `pdc_commands`. This class implements the more complex but generic functionality like the register and memory read caches, and is considered to be processor-independent.
- `<processor>_pdc_interface`. In this (optional) class, which is derived from the above `Checkers_pdc_interface` class, processor-specific specializations can be done. Typically, some few virtual functions are redefined here.

For a new processor, when using the PDC unit generated by GO, only small changes are required to the `pdc_commands` class, typically related to register and memory access (e.g., the way data is moved between memories/registers and the PDC data register is processor dependent). Further processor-specific specializations can be implemented in the `<processor>_pdc_interface` class, while the generic code in `Checkers_pdc_interface` should be left unaffected.

To enable debug client generation in CHESSDE, you specify the `<processor>_pdc_interface` class name and its header file under **Debug client→Model→Controller**.

When using our hardware link, the debug client generated by CHECKERS connects with the `jtalk` server program, driving the parallel or Amontec USB cable.

When using a third-party debugger DLL to interact with the target processor, instead of using our default hardware link, also a `pdc_commands` implementation is available showing how to interface with such a debugger DLL.

Appendix A

Implementing hosted I/O

When executing any <stdio.h> functions on the target processor, the resulting I/O calls are intercepted by the CHECKERS simulator or debugger, which executes these I/O actions on the host computer, and then resumes execution on the target. The interception works via a special hosted I/O break point. The characteristics of this type of file I/O, like interception at a high level, to reduce the amount of I/O code on the target processor, and the possibility to read or write a larger amount of data via the binary `fread()`/`fwrite()` functions, are discussed in [3, §3.4.1].

Section A.1 discusses the implementation of hosted file I/O via <stdio.h> in more detail. Section A.2 discusses a different way of doing hosted I/O, called hosted calls, only available in instruction-accurate (IA) simulation mode.

A.1 Hosted file I/O via <stdio.h>

We provide a restricted runtime C library, which supports the complete <stdio.h> header file, almost fully compliant with the ISO/IEC C99 standard. In case of IP PROGRAMMER, the ported runtime C library is included in the distribution, in the form of header files and compiled archive, and it is ready to use. In case of IP DESIGNER, the top directory of the distribution contains the `runtime.zip` archive, containing the header files and source code of our C library. The local `README` file explains how to retarget and install this library. This section further describes the <stdio.h> implementation.

A.1.1 Differences with C99 standard

Our hosted implementation of <stdio.h> does not support some special *length modifiers* and *conversion specifiers* in the format string of the different `printf()` and `scanf()` functions :

- The special length modifiers `j`, `z`, and `t` are not supported. All other length modifiers (like `h`, `l`, or `ll`) are supported.
- No length modifiers as supported for the conversion specifiers `c` and `s`.
- The conversion specifiers `p` and `n` are not supported.

On the other hand, for memories not supporting byte access, we provide additional word version (`fread_word()` and `fwrite_word()`) of the `fread()` and `fwrite()` functions [3, §3.4.1].

`fread()` and `fwrite()` functions can be provided per memory, to be able to read/write to the different processor root memories. This is done by adding the appropriate `chess_storage()` annotation to the `void*` argument, e.g. :

```
size_t fread(void chess_storage(DM)* ptr, ...);
size_t fread(void chess_storage(PM)* ptr, ...);
```

A.1.2 Interface between ISS and target processor

To reduce the amount of code that has to run on the target processor (which may have limited memory sizes), I/O calls are intercepted at a high level. For example a function like `printf()`, passes its parameters and the required I/O action to an internal I/O struct, after which it calls an internal hosted I/O function, onto which a break point is set. When the hosted I/O break point is hit, the Checkers simulator or debugger reads out the I/O struct, does the required I/O action (e.g. involving the interpretation of the `printf()` format string), and writes back any results to the I/O struct, after which execution resumes.

Note that it is essential that DWARF variable information is enabled in the CHESS compiler for the hosted I/O functionality to work. Otherwise, the ISS cannot find back the I/O interface struct in the memory of the target processor.

The hosted I/O interface struct defined in `src/stdio.c` looks as follows :

```
struct Hosted_clib_vars {
   int         call_type;
   int         stream_id;
   int         stream_rt;
   const char* path;
   const char* mode;
   const char* format;
   long        offset;
   int         whence;
   int         eof;
   int         c;
   const char* puts_s;
   char*       gets_s;
   int         size;
   int         nmemb;
   union {
      const void chess_storage(DM)* write_ptr_dm;
      const void chess_storage(PM)* write_ptr_pm;
   } write_ptr;
   union {
      void chess_storage(DM)* read_ptr_dm;
      void chess_storage(PM)* read_ptr_pm;
   } read_ptr;
   void*       ap; // va_list ap
};
```

The two unions with read and write pointers (here for the Base example processor) must be modified to contain pointers for all root memories on which `fread()` and `fwrite()` operations are needed.

The hosted I/O breakpoint is put on the `_hosted_clib_io_brkpt` label inside the `_hosted_clib_io()` function when loading a program into the ISS. For the Base example core this function is implemented as follows :

```
inline assembly void _ihosted_clib_io() property(loop_free) clobbers()
{
   asm_begin
      nop
      nop
.label _hosted_clib_io_brkpt
      nop
   asm_end
}
```

```
extern "C" void _hosted_clib_io(Hosted_clib_vars* p) property(loop_free)
{
    _ihosted_clib_io();
    p->call_type = clct_none;
}
```

Every hosted I/O function calls the `_hosted_clib_io()` function. The `_ihosted_clib_io()` inline assembly function (with additional NOPs) makes sure that the instruction pipeline is flushed before reaching the breakpoint `_hosted_clib_io_brkpt`.

When the breakpoint is hit, the ISS will execute the I/O call based on the information it reads in the interface struct. The interface struct should be allocated on the software stack, and should be passed as argument of the `_hosted_clib_io()` function. The example below shows the implementation of the `<stdio.h>` function `fputs()` :

```
int fputs(const char* s, FILE* stream) property(loop_free)
{
   Hosted_clib_vars _hosted_clib_vars;
// Pass arguments to host
   _hosted_clib_vars.stream_id = stream->stream;
   _hosted_clib_vars.puts_s = s;
   _hosted_clib_vars.call_type = clct_fputs;
// Initialize result to error (result >= 0 if OK and EOF in case of error)
   _hosted_clib_vars.stream_rt = EOF;
// Call fputs() on host
   _hosted_clib_io(&_hosted_clib_vars);
// Pass result from host
   return _hosted_clib_vars.stream_rt;
}
```

Note that by initializing the function result of `fputs()` to failure, a meaningful result is still returned when executing the main program while hosted I/O functionality would be disabled or not available (when running the program outside the CHECKERS debugger).

A.2 Hosted calls

Hosted calls are a fast way to do file I/O, only available in the instruction-accurate simulation mode.

During simulation, when entering a hosted function, instead of simulating the dummy function code on the target processor, a corresponding function on the host computer is called doing the actual file I/O, and simulation continues. The function called on the host takes a pointer to the processor model class as argument. An example is worked out below for the Base example processor.

First, you enter the function names `get_input` and `get_output` in the CHESSDE simulator option **Hosted calls** (§8.4.2). These functions can then be implemented as follows on the host computer :

```
#include <stdio.h>
#include <signal.h>
#include "Mdl_Tmicro.h"

static FILE* Tmicro_in;
static FILE* Tmicro_out;
static int init_done = 0;

inline void init_io  ()
{
   Tmicro_in = fopen("in.dat", "r");
```

```
   Tmicro_out = fopen("result.dat", "w");
   init_done = 1;
}

inline void get_input(Tmicro* b)
{
   if (!init_done)
      init_io();
   if (feof(Tmicro_in))
      raise(SIGINT); // Stop simulation
   int i;
   fscanf(Tmicro_in, "%d\n", &i);
   b->R[0] = i;
}

inline void put_output(Tmicro* b)
{
   if (!init_done)
      init_io();
   fprintf(Tmicro_out, "%d\n", b->R[0].value());
}
```

The application program on the target processor can provide dummy functions (only the function interface is relevant) :

```
int get_input() { return 0; }

void put_output(int i) { }
```

These functions can now be used to do basic I/O :

```
extern int fir(int);

int main() {
  while (1) put_output(fir(get_input()));
}
```

Note that in the host implementation, the function arguments must be accessed according to the processor argument call convention.

Appendix B
Mic format

The mic format is a simple format used to store executable code (text segments only) generated by CHESS.

A mic file has the following syntax:

mic file:
    ListOfLines

ListOfLines: one of
    Line
    ListOfLines Line

Line: one of
    LineNr Rts_Indicator Instruction Comment
    Comment

ListNr:
    integer
    Hexadecimal_integer

Rts_Indicator:
    —
    `.rts`

Instruction: one of
    ‘‘ *Binary_integer* ’’
    Hexadecimal_integer

Comment:
    —
    // any text

All integers should be decimal integers. A Hexadecimal_integer must start with `0x`. A *Binary_integer* must have the length of an instruction on the processor.

A return-from-subroutine indicator is used to mark return-from-subroutine instructions in case a plain jump is used both for jumping and for returning from a subroutine.

This is an example mic file :

```
0      "101111000000000110" //     JSR 0x6
1      "100100010100000000" //     AY = DM[0x100]
3      "101011110000000000" //     JUMP TO 0x0
4      "100110000100000001" //     DM[0x101] = AX
```

Bibliography

[1] *Go User manual, nML to HDL translation*. Target Compiler Technologies, Technologielaan 11-0002, B-3001 Leuven, Belgium, March 2011. Release 11R1.

[2] *Checkers ISS Interface manual*. Target Compiler Technologies, Technologielaan 11-0002, B-3001 Leuven, Belgium, March 2011. Release 11R1.

[3] *Chess Compiler User manual*. Target Compiler Technologies, Technologielaan 11-0002, B-3001 Leuven, Belgium, March 2011. Release 11R1.

[4] *Primitives Definition and Generation manual*. Target Compiler Technologies, Technologielaan 11-0002, B-3001 Leuven, Belgium, March 2011. Release 11R1.

[5] *Chess Compiler Processor Modeling manual*. Target Compiler Technologies, Technologielaan 11-0002, B-3001 Leuven, Belgium, March 2011. Release 11R1.

[6] *The nML Processor Description Language*. Target Compiler Technologies, Technologielaan 11-0002, B-3001 Leuven, Belgium, March 2011. Release 11R1.