The ADL Command Line Interface (CLI)

A Description of the CLI

This document describes the ADL command line interface (CLI).

Author:	Michele Reese
Contact:	michele.reese@freescale.com
Author:	Brian Kahne
Contact:	brian.kahne@freescale.com

Table of Contents

1 The CLI

1.1 Overview

The CLI is a simple interactive and scripting interface to the ISS. This allows the user to control the execution of the model through such things as setting and clearing breakpoints, turning logging off and on for specific events, and slicing, where slicing is a methodology for identifying and extracting an interesting stream from a testcase. TCL is used as the scripting language. All of the usual TCL commands may be used, as well as the extra commands here, which are unique to this interface and all the user to control the ISS.

The interactive scripting mode simply uses TCL's main event loop and thus, by default, provides only very primitive command-line editing capabilities. These may be augmented, however, by using an external package such as `tclreadline`. For example, assuming that `tclreadline` is installed as a CDE/GAIN package, a user's `.tclshrc` file would contain the following code in order to use this package:

```
if {$tcl_interactive} {
        # If tcl is on an x86 32-bit or 64-bit system, then use the relevant
        # tclreadline package explicitly.  Otherwise, use plat.  This means that if
        # you're on a 64-bit system, but you invoke a 32-bit version of tcl, the
        # proper library will still be loaded.
        if { [regexp {(i686-[^/]+)|(x86_64-[^/]+)} [info library] match] } {
                lappend ::auto_path /_TOOLS_/dist/tclreadline-2.1.0/$match/lib
        } else {
                lappend ::auto_path /_TOOLS_/plat/tclreadline-2.1.0/lib
        }

        package require tclreadline

        # For now, disable custom completers, since installed commands might not
        # have a custom completer and thus you'll get weird behavior.
        ::tclreadline::readline customcompleter ""
        namespace eval tclreadline {
                # Optional, but the default prompt is kinda verbose, in my opinion.
        proc prompt1 {} {
                        return "% "
        }
        }
        ::tclreadline::Loop
}
```

The CLI also provides filtering which can be useful for decreasing the size of the text-based DAT files by turning off logging for events which are not of interest.

1.2 Invoking the Command-line Interface

To use the CLI, specify the `--cli` option on the command-line. For non-interactive scripting purposes, simply use the `--script=<file>` option. For example:

```
./model --cli foo.dat --script script.cli
```

This invokes `model`, invokes the TCL interpreter, and executes the script `script.cli`.

To invoke a script to be executed before any other actions occur, use the `--config=<file>`` option. This runs the specified file and then proceeds with normal operation, e.g. running a simulation, executing a script, etc.

You can also evaluate statements from the command-line using the `--eval=<cmd>` option. For example:

```
./model --script=foo.tcl --eval="set myvar 20"
```

You may also provide command-line arguments via the `--script-arg=<value>` option. These are inserted into Tcl's argv list:

```
./model --script=foo.tcl --script-arg="foo.txt"
```

This will assign `myvar` a value of 20 in the interpreter. All command-line `eval` commands, the `config` script, and the interactive interpreter or the script specified via `script` are all executed by the same interpreter.

1.3 Command-line Completion

Pressing the TAB key will cause the CLI to attempt to complete a word that the user is entering. If the word is at the beginning of the line, then the CLI will display all possible commands that match the existing word and will complete the word with the appropriate command if only one possibility exists. After the first word, the CLI will default to completing filenames, unless the current word begins with "$". If that is the case, then the CLI will attempt to complete an environment variable. Variable syntax of `$var` and `${var}` is supported. If a complete environment variable has been entered, followed by "/", then the CLI will substitute the value of the environment variable, if the variable exists.

1.4 Slicing

Slicing uses the `sliceon` and `sliceoff` or `traceon` and `traceoff` commands. The `sliceon` and `traceon` commands causes a UVP or DAT writer to be `created and registered with the logging manager and the current state of the model to be dumped to the UVP as the initial state.   The ``sliceoff` and `traceoff` commands causes the current state of the model to be dumped to the target file as the final state and the file to be closed. Concurrent slices are allowed: The `sliceon` or `traceon` command may be called while another slice is active. The `sliceoff` and `traceoff` commands take an optional filename argument. If supplied, then only that slice is disabled. Otherwise, all slices are disabled. The only difference between the trace and slice commands is that the trace command takes effect immediately, whereas the slice command is used to activate a trace when a specific instruction address is encountered. The address may be specified as a symbol name and an ignore-count may be specified.

An example of a slicing run:

```
setreg NIA0 0xd7c
setbrk 0xd32c
setreg L1CSR0 0x1
setreg L1CSR1 0x1
setreg L2CSR0 0x80000000
sliceon  0xd2c 1 p.slice1.uvp
sliceoff 0xd58 1
go
sliceon  0x2760 1 p.slice2.uvp
sliceoff 0x2e38 1
go
sliceon  0x27bc 5 p.slice3.uvp
sliceoff 0x25c 7
go
sliceon  0x27c 1 p.slice4.uvp
sliceoff 0x25c 13
```

```
go
sliceon  0x27c 5 p.slice5.uvp
sliceoff 0x25c 17
go
sliceon  0x27c 5 p.slice6.uvp
sliceoff 0x25c 17
go
quit
```

1.5 Annotation

ADL provides a mechanism for passing information through to the trace file. The user can instrument a model with annotation commands using the "info" function within an ADL description. You can also pass this information through the CLI using the annotate command:

```
annotate <address | symbol> "<msg>"
```

where `<msg>` can be any quoted string (escape characters are not currently supported).

Model state can be embedded in the message string using the following syntax options:

%reg:regname

%reg:regname[index]

%mem:address

All substituted values will display in hex notation in the annotation card, and all addresses embedded in the message must be in hex format.

Examples:

```
ADLISS> annotate 0x1000 "we just hit 0x1000 and NIA=%reg:NIA"
```

This would insert an annotation into the DAT stream showing the value of NIA each time address 0x1000 is encountered, for example:

```
"we just hit 0x1000 and NIA=0x1004"
```

```
ADLISS> annotate 0x1000 "we just hit 0x1000 and R1=%reg:GPR[1]"
```

This would insert an annotation into the DAT stream showing the value of R1 each time address 0x1000 is encountered:

```
"we just hit 0x1000 and R1=0xdeadbeef"
```

```
ADLISS> annotate 0x1000 "we just hit 0x1000 and mem[0x4000]=%mem:0x4000"
```

This would insert an annotation into the DAT stream showing the value stored in memory at 0x4000 each time address 0x1000 is encountered:

```
"we just hit 0x1000 and mem[0x4000]=0xfeedface"
```

1.6 Commands

There are two types of commands for the CLI.

1. On the command line to startup the CLI you have access to all of the ISS command line options. Type `--help` to display usage. A command script may be specified using input redirection:

   ```
   ./model in.dat < script.cli
   ```

   If a command file is specified, then the script is executed one command at a time and no interactive prompt is displayed.
2. Once you have started up the CLI and you are at a prompt, you have access to the CLI commands. You can type `help` at this prompt to see these.

You must type `quit` to exit the CLI.

The CLI commands common to ADL and uADL are:

- **active**:

  Return the active count for the current or specified path. Usage: `active [path]`

  For a core, the active count is 0 (if the core is halted) or 1 if active. For a system, this is the sum of the counts of the child nodes.
- **annotate**:

  Annotate the trace file. Usage: `annotate <address | symbol> "<msg>"`

  The annotation will execute when the specified address is reached.
- **cancelexcpt**:

  Cancel a pending, level-sensitive exception.

  Usage: `cancelexcpt [path:]<exception_name>`

  If the specified exception is not pending, then an error is returned.
- **cia**:
  Returns the current instruction address for the current or specified core. Usage: `cia [path]`.

- **clrbreakcmd**:
  Remove a breakpoint call-back function. Usage: `clrbreakcmd <func>`
- **clrbrk**:
  Clear a breakpoint. Usage: `clrbrk <address | symbol>`
- **clrfilter**:

  Turn off logging for specific events. Usage: `clrfilter [type ...]`

  The available options for filtering are:

Filter Types

Command	Description
annotation	Turn off annotation commands.
basic_block_end	Turn off end-of-basic-block logging.
breakpoint	Turn off breakpoints.
branchtaken	Turn off logging of branch-taken information.
cache_action	Turn off all cache events.
core_mem_read	Turn off logging of memory reads from the core's perspective (D cards in DAT files)
core_mem_write	Turn off logging of memory writes from the core's perspective (D cards in DAT files)
exception	Turn off logging of exceptions.
instr	Turn off logging of instructions (INSTR cards in DAT files).
instr_prefetch	Turn off instruction fetch-event logging (I cards in DAT files).
instr_read	Turn off instruction fetch logging (M cards between I and INSTR cards in DAT files).
instr_times	Turn off instruction initiation/completion time logging in time-tagged ISSs).
mem_read	Turn off logging of memory reads (M cards in DAT files).
mem_write	Turn off logging of memory writes (M cards in DAT files).
mmu_trans	Turn off logging of MMU activity.
reg_write	Turn off logging of register writes.
regfile_write	Turn off logging of register-file writes.
reg_read	Turn off logging of register reads.
regfile_read	Turn off logging of register-file reads.
watchpoint	Turn off logging of watchpoints.

If no events are specified, then all events are turned off.

- **clrlogcmd**:

  Remove a logging call-back function. Usage: `clrlogcmd <func>`
- **clrwatchcmd**:

  Remove a watchpoint call-back function. Usage: `clrwatchcmd <func>`
- **excptlist**:

  Return a list of all valid exceptions for a core. Usage: `excptlist [path]`
- **filterlist**:

  Return a list of all logging events currently active, such as those set by `setfilter`. Usage: `filterlist`
- **getdynparm**:

  Retrieve the value of a dynamic configuration parameter. These are parameters which may be used to change aspects of a model at run-time such as cache configurations. Whether these parameters exist or not is dependent upon how the model was built.

  Usage: `getdynparm <parm-name>`

  Returns the integer value of the parameter, if the parameter is valid, or else returns an error message (and error code) if the parameter is invalid.
- **gettd**:

  Retrieve a temporal-decoupling parameter.

  Usage: `gettd [path:]<parm>`

  Refer to the settd command for more information about valid parameter values.
- **go**:

  Run until halt or break. If dynamic-binary-translation or the trace-cache is enabled (through the use of the `simmode` command), then execution will use the high-speed translation mechanism, if the model was built with this feature.

  Usage: `go`
- **help**, **?**:

  Display help information. If a command name is specified, then only help information about that command is listed. Usage: `help [command-name]`

  The `command-name` argument may be a sub-string, in which case all matching commands will have their help information printed.

- **icount**

  Display the instruction count for the current or specified node. Usage: `icount [path]`

  This displays the result in a human-readable fashion and returns the value to the interpreter so that it may be used by other TCL command.
- **ijam**:

  Directly execute an instruction. Usage: `ijam [path] <word> [word...]`

  This bypasses the normal fetch logic of the model, directly executing the specified instruction. Opcodes are specified as one or more 32-bit words. This is currently only implemented for ADL models.
- **inputlist**

  Display a list of files that have been read for this simulation. Usage: `inputlist`

  Returns a two-column list, where the first column is the filename and the second column is the format type, e.g. "dat", "elf32", etc. This return value may be directly read into a TCL dictionary.
- **listdynparm**

  List all of the model's dynamic parameters. These are parameters which may be used to change aspects of a model at run-time such as cache configurations. Whether these parameters exist or not is dependent upon how the model was built.

  Usage: `listdynparm [parm]`

  Returns a string listing parameter names and descriptions for all parameters in the system. If a `parm` argument is specified, then only those parameters containing `parm` will be listed.
- **logreg**:

  Log a specific register. Usage: `logreg [path:]<reg_name> [index]`

  Essentially, a **reg** command is performed, but the output is directed to the currently installed loggers. This can be useful when paired with the **setreg** command, as that command does not log a value update.
- **mem**:

  Display memory contents. Usage: `mem <start_address> <length> [read-size-in-bytes] [words-per-line]`

read-size is specified in bytes: 1, 2, 4, or 8. The default is 4 (a 32-bit word).

words-per-line specifies the number of elements to print per line. The default is 4.

- **memlist**:

  Return a list of 32-bit memory values. Usage: `memlist <start_address> <length> [read-size-in-bytes] [radix]`

  The result of this function may be used to directly initialize a TCL list.

  read-size is specified in bytes: 1, 2, 4, or 8. The default is 4 (a 32-bit word).

  radix may be either *dec*, *hex*, or *char*. The *char* option may only be used with a read-size of 1 byte and the result does not have white-space added between elements. Thus, the result may be used directly as string interpretation of memory results. If the radix is *hex*, then a *0x* prefix will be used. The default is *hex*.
- **modelreset**
  Reset the models in the simulation. Memory is not reset. Usage: `modelreset`
- **path**:
  Display the current modification path. Usage: `path`
- **pathlist**:

  Return a list of all valid paths. Usage: `pathlist [sys|cores|threads|]`

  If an optional argument flag is specified, then only paths of those types are returned. Multiple flags are allowed; specifying all flags is equivalent to not specifying any arguments.
- **pc**:
  Returns the current program counter for the current or specified core. Usage: `pc [path]`.
- **quit**:
  Quit the program. Usage: `quit`
- **reg**:
  Show a specific register. Usage: `reg [path:]<reg_name> [index]`
- **readreg**:
  Read a specific register using its in-model read method. Usage: `readreg [path:]<reg_name> [index]`
- **regs**:
  Show all registers. Usage: `regs [path]`

- **setactive**:

  Set the active count for the current or specified path. Usage: `setactive [path] <value>`

  This function allows the user to halt a core or set of cores by setting the active count to 0, or re-enable cores by setting the active count to 1. The prior active count is returned.
- **setbrk**:
  Set a breakpoint. Usage: `setbrk <address | symbol>`
- **setbreakcmd**:

  Add a call-back function to be invoked upon a breakpoint.

  Usage: `setbreakcmd <address | symbol> <cmd>`

  The call-back function's arguments are: `id effective-address`, where `id` is the index of the breakpoint and `effective-address` is the current effective address which caused the breakpoint. If the function returns the string `simstop`, then the simulation will halt and return control back to the interpreter.
- **setdynparm**:

  Set the value of a dynamic configuration parameter. These are parameters which may be used to change aspects of a model at run-time such as cache configurations. Whether these parameters exist or not is dependent upon how the model was built.

  Usage: `getdynparm <parm-name> <value (int)>`

  Returns 1 on success, or else returns an error message (and error code) if the parameter or value are invalid.

  This function should normally be called within a configuration script (--*config* command-line option) so that the model is fully configured before any data is read into the model or simulation begins. While resources may be modified during a simulation, this is not recommended, since not all state is preserved. For example, caches are re-allocated and cleared when they are re-sized.
- **setfilter**:

  Turn on logging for specific events. Usage: `setfilter [type ...]`

  Refer to Filter Types for possible values. If no events are specified, then all events are enabled.

- **setlogcmd**:

  Add a call-back function to be invoked upon a specified trace event. If an event takes a filter value, then the function is only invoked if the specified element is being logged. For example, a function may be set up so that it will only be called on a write to a specific register, rather than on a write to any register.

  The function is only invoked on an event for the node specified by the current path. Note that unlike other functions, no default context is assumed if contexts exist for the design. Thus, for a path which does not specify a context, a registered function will be triggered if the event occurs for any context.

  Usage: `addbreakcmd [path]<event> <cmd> <filter (optional)> <filter-arg1 (optional)> <filter-arg2 (optional)>`

  The valid event types are:

Event Types

Event	**Function Arguments**	**Filter Type**	**Additional Filter**	**Description**
basic_block_end	effective-address	None	None	Called upon end of a basic block.
instr	effective-address, opcode, instr-name	None	None	Called on instruction execution.
instr_args	opcode,instr-name, sources,targets,	None		Called on instrucion execution.
reg_write	id, name, value	Register name	None	Called on a write to any register, or a write to a specific register, if a filter value is specified.
regfile_write	id, name, index, value	Register-file name	Reg-file Index	Called on a write to any register-file, or a write to a specific register-file, if a filter value is specified.
reg_read	id, name, value	Register name	None	Called on a read to any register, or a read to a specific register, if a filter value is specified.

Event	Function Arguments	Filter Type	Additional Filter	Description
regfile_read	id, name, index, value	Register-file name	Reg-file Index	Called on a read to any register-file, or a read to a specific register-file, if a filter value is specified.
mmu_translation	id, translation-type, set, way, effective-address, real-address, lookup-name	None	None	Called on an MMU translation.
cache_action	id, cache-name, cache-action, cache-access, level, set, way, linemask, real-address	None	None	Called on a cache access.
mem_write	id, name, pre-write-flag, sequence, effective-address, real-address, value	Memory name	Address *or* Start-Address End-Address	Called on a write to memory (or a specific memory).
mem_read	id, name, pre-write-flag, sequence, effective-address, real-address value	Memory name	Address *or* Start-Address End-Address	Called on a read from memory (or a specific memory).

Event	Function Arguments	Filter Type	Additional Filter	Description
exception	id, name	Exception name		Called on the occurrence of any exception or a specific exception.

- **setmem**:

  Set a memory value. Usage: `setmem <address> <value> [length]`

  Length is specified in bytes and may range from 1 to 4. The maximum write size is a 32-bit word. The default is 4.
- **setpath**:
  Specify a new modification path. Usage: `setpath path`
- **setpc**:

  Set the program counter of the current or specified core.

  Usage: `setpc [path] <value>`.
- **setreg**:
  Set a register. This does not trigger any side-effects due to write hooks but will trigger side effects caused by various watch predicates, e.g. setting of active contexts, in order to keep the model consistent. Usage: `setreg [path:]<reg_name> [index] <value>`
- **setregext**:
  Set a register. This will execute an external-write hook, if present for the register. Otherwise, its behavior is identical to **setreg**. Usage: `setregext [path:]<reg_name> [index] <value>`
- **setregendianness**:

  Set endianness to use when communicating register values with a debugger.

  Usage: `setregendianness <path> <big|little>`

  If <`path`> is a system, then all of the individual cores receive the same value. If <`path`> is set to the special value **root**, then the root node is used for configuration purposes.
- **setregmap**:

  Set a debug register map. This creates a mapping between indices used by a debugger, such as `gdb` and the registers within an ADL model. This function must be called by a configuration script before a model may be connected with a debugger. The first element in the list is assigned an index of 0, etc.

  Usage: `setregmap <path> <reg-item>...`

If `<path>` is a system, then all of the individual cores receive the same register mapping information. If `<path>` is set to the special value **root**, then the root node is used for configuration purposes.

Where `<reg-item>` is:

- `<register-name>:<width (int)>`: Specify a register and a specific width to apply to the register (in bits).
- `<register-name>`: Specify a register. The register's natural width is used.
- `<int>`: Specify a dummy placeholder. The integer is the placeholder width in bits.

To map elements of a register file, specify the name as `<register-name><index>`, e.g. element 5 of register file **GPR** would be expressed as **GPR5**. To then force this to be a 32-bit value: `GPR5:32`.

- **settd**:

Set a temporal-decoupling parameter. For a standalone simulation, this is only really useful for an MP JIT simulation, as the temporal-decoupling (TD) API is used for scheduling between cores. However, it may also be used as a means for returning control back to the scripting interface after N number of instructions have been executed.

Usage: `settd [path:]<parm> <value>`

The parameter name may be one o the following:

settd Parm Types

Parm	Description
threshold	Number of instructions to execute before returning control
count	Current instruction count
incr	Amount of increment for each instruction executed.

- **setwatchcmd**:

Add a call-back function to be invoked upon a watchpoint.

Usage: `setwatchcmd <address | symbol> <type> <cmd>`

The call-back function's arguments are: `id type effective-address pre-flag num-bytes value`. If the function returns the string `simstop`, then the simulation will halt and return control back to the interpreter. The `value` argument is only valid if the watchpoint is triggered by a write.

The watchpoint type may be one of the following:;;

Watchpoint Types

Type	Description
read	Trigger an event upon a read of this memory location.
write	Trigger an event upon a write to this memory location.
access	Trigger an event upon a read or a write to this memory location.

- **simload**:

  Load a file. Usage: `simload <filename> [format] [setpc] [--] [extra-flags]`

  The `format` option allows the user to override the default format, which is obtained from the filename's extension. If the `setpc` flag is present, then the load command will set the program counter of the core (or cores). This option is currently only supported for ELF files; for DAT files, the program counter is set if a register initialization command for the program-counter register is contained within the file.

  The `--` argument acts as a separator; everything afterward is regarded as a reader-specific flag. The separator is optional and not required if a format or `setpc` are specified.

  Currently supported reader-specific flags are:

Reader-Specific Flags

Reader Type	Flag	Description
ELF	elf-ea	Use effective addresses when loading the ELF file.
ELF	no-elf-ea	Turn off the use of effective addresses when loading an ELF file.
ELF	elf-pa	Translate logical addresses to physical using translations from the ELF headers.
ELF	no-elf-pa	Do not use logical-to-physical translations.
ELF	elf-log-bss	Log the usage of memory in the BSS section.
ELF	no-elf-log-bss	Turn off logging of the BSS section.
ELF	elf-use-fd	Use function descriptors for symbol lookup.

Reader Type	Flag	Description
ELF	no-elf-use-fd	Do not use function descriptors for symbol lookup.

- **simmode**

  Specify the type of interpreter to use (the normal interpreter, the high-speed dynamic-binary-translating ISS, or the trace-cache ISS). Returns the current type. May be called with no arguments, in which case it just returns the type. Subsequent calls to `go` will use the new type when executing.

  Usage: `simmode [normal|jit|dc]`
- **simreset**

  Reset the simulation. Usage: `simreset`

  This resets the simulation and clears the list of input files returned by `inputlist`.
- **simstat**

  Return statistics related to the simulator. Usage: `simstat <parm-name>`

  The exact parameters supported depend upon the model configuration.

  For models with JIT support:

  - **cold_runs**: Returns the number of basic blocks executed via the interpreter.
  - **dc_compiles**: Returns the number of decode-cache compiles performed.
  - **jit_compiles**: Returns the number of hot blocks compiled by the JIT.
  - **jit_dc_runs**: Returns the number of warm-but-not-hot basic blocks executed via the decode-cache system.
  - **jit_escapes**: Returns the number of returns to the simulation kernel, such as those caused by exceptions, calculated branches, page crossings, etc. Only valid if the model was compiled with `JIT_DEBUG` defined.
  - **jit_bb_count**: Returns a count of the basic blocks stored by the JIT translation has.
  - **jit_thread_count**: Returns the number of JIT worker threads which exist.
- **sliceoff**:

  Turn off slicing. Usage: `sliceoff <address | symbol> [count] [filename]`.

If no filename is specified, then all slices are disabled.

- **sliceon**:

  Turn on slicing. Usage: `sliceon <address | symbol> [count] <filename>`
- **step**:

  Step the currently active or specified path by one or `count` instructions. Usage: `step [path] [count]` In case of parallel architectures the whole instruction set is executed on one step.
- **straceon**:

  Turns on stream tracing. Usage: `straceon <fmt> [stdout|stderr]`. The `<fmt>` argument describes the type of tracing to use, currently `dat` or `uvp`. The second argument, if specified, directs output to the specified destination. If not specified, then tracing is directed to an in-memory buffer. The results may then be retrieved as a string by calling `straceget`.
- **straceget**:

  Return the current contents of the stream trace. Usage: `straceget`.

  After a call to `straceget`, the stream is reset so that a subsequent call will only return trace data for instructions executed after the last call to `straceget`. If the streaming interface is directed to stdout, then this call simply flushes stdout.
- **straceoff**:

  Turns off stream tracing. Usage: `straceoff`.
- **symbol**:

  Return the memory address of a symbol from the symbol table. Usage: `symbol <symbol>`.

  Note that this will only search the symbol table of a file loaded using the `simload` command. It will not work for any files loaded from the command-line.
- **traceoff**:

  Turn off tracing. Usage: `traceoff [filename]`.

  If no filename is specified then all traces are disabled. Otherwise, only the specified filename is disabled.
- **traceon**:

  Turn on tracing. Usage: `traceon <filename>`.

  Use this command, rather than `sliceon`, when you want to immediately enable tracing.

- **tracelist**

  Display a list of current tracing files. Usage: `tracelist`

  Returns a two-column list, where the first column is the filename and the second column is the format type, e.g. "dat", "uvp", etc. This return value may be directly read into a TCL dictionary.
- **tracemsg**

  Send a textual message to the logging system, if tracing of annotations is enabled. Usage: `tracemsg [path] <level> <msg>`

  Returns "1" if the message was sent (annotation tracing is on) or "0" if tracing is off.
- **triggerexcpt**:
  Trigger an external exception. Usage: `triggerexcpt [path:]<exception_name>`
- **writereg**:
  Write a register. This will trigger side effects caused by write hooks as well as side effects cause by any watch predicates. Usage: `writereg [path:]<reg_name> [index] <value>`

1.7 uADL Commands

The following commands are available only in uADL models.

- **ccount**
  Display the current cycle count. Usage: `ccount [path]`.
- **commitcount**
  Display the current instruction commit count. Usage: `commitcount [path]`.
- **cycle**:
  Cycle by one or `count` cycles. Usage: `cycle [count]`
- **issuecount**
  Display the current instruction issue count. Usage: `issuecount [path]`.
- **ptraceon**

  Turn on pipeline tracing. Usage: `ptraceon [filename|stdout|stderr]`.

  If no argument is specified, then logging is directed to standard out.
- **ptraceoff**
  Turn off pipeline tracing. Usage: `ptraceoff`.

Generated on: 2018/02/28 15:17:11 MST.