RISC-V Debugger

TRACE32 Online Help

TRACE32 Directory

TRACE32 Index

TRACE32 Documents ....

ICD In-Circuit Debugger ....

Processor Architecture Manuals ....

RISC-V ....

RISC-V Debugger 1

History 3

Introduction 4

Brief Overview of Documents for New Users 4

Demo and Start-up Script 5

List of Abbreviations and Definitions 6

Warning 7

Quick Start of the JTAG Debugger 8

Quick Start for Multicore Debugging 10

SMP Debugging 10

SMP Debugging - Selective 11

Homogeneous SMP/AMP Debugging 12

Heterogeneous SMP/AMP Debugging 13

Troubleshooting 14

SYStem.Up Errors 14

FAQ 15

RISC-V Specific Implementations 17

Debug Specification for External Debug Support 17

Floating-Point Extensions 17

Breakpoints 19

Software Breakpoints 19

On-chip Breakpoint Resources 19

On-chip Breakpoints for Instruction Address 19

On-chip Breakpoints for Data Address 19

On-chip Data Value Breakpoints 20

Examples for Standard Breakpoints 21

Access Classes 22

CPU Specific SYStem Commands 23

©1989-2018 Lauterbach GmbH

SYStem.CONFIG.state Display target configuration 23
SYStem.CONFIG Configure debugger according to target topology 24
<parameters> describing the “DebugPort” 25
<parameters> describing the “JTAG” scan chain and signal behavior 27
SYStem.CONFIG.HARTINDEX Set hardware thread index 30
SYStem.CPU Select the used CPU 31
SYStem.CpuAccess Run-time memory access (intrusive) 31
SYStem.JtagClock Define JTAG frequency 32
SYStem.LOCK Tristate the JTAG port 33
SYStem.MemAccess Run-time memory access (non-intrusive) 33
SYStem.Option MMUSPACES Separate address spaces by space IDs 34
SYStem.Mode Establish the communication with the target 35
SYStem.Option Special setup 36
SYStem.Option Address32 Define address format display 36
SYStem.Option EnReset Allow the debugger to drive nRESET (nSRST) 36
SYStem.Option IMASKASM Disable interrupts while single stepping 37
SYStem.Option TRST Allow debugger to drive TRST 37
SYStem.state Display SYStem.state window 37

CPU Specific FPU Command **38**
FPU.Set Write to FPU register 38

CPU Specific Functions **39**
XLEN() Current width of XLEN 39

Target Adaption **40**
Connector Type and Pinout 40
RISC-V Debug Cable with 20 pin Connector 40
Converter ARM-20 to ALTERA-10/RISCV-10 41

Support **42**
Available Tools 42
Compilers 42

Products **43**
Product Information 43
Order Information 43

©1989-2018 Lauterbach GmbH

Version 16-Nov-2018

History

29-Jun-18	Added description and example for the command FPU.Set.
22-Jun-18	New chapter “Quick Start for Multicore Debugging”
11-Apr-18	Added description of the command SYStem.Option MMUSPACES.
04-Sep-17	New manual.

©1989-2018 Lauterbach GmbH

Introduction

This manual serves as a guideline for debugging one or multiple RISC-V cores via TRACE32.

Please keep in mind that only the **Processor Architecture Manual** (the document you are reading at the moment) is CPU specific, while all other parts of the online help are generic for all CPUs supported by Lauterbach. So if there are questions related to the CPU, the Processor Architecture Manual should be your first choice.

Brief Overview of Documents for New Users

Architecture-independent information:

- **"Debugger Basics - Training"** (training_debugger.pdf): Get familiar with the basic features of a TRACE32 debugger.
- **"T32Start"** (app_t32start.pdf): T32Start assists you in starting TRACE32 PowerView instances for different configurations of the debugger. T32Start is only available for Windows.
- **"General Commands"** (general_ref_<*x*>.pdf): Alphabetic list of debug commands.

Architecture-specific information:

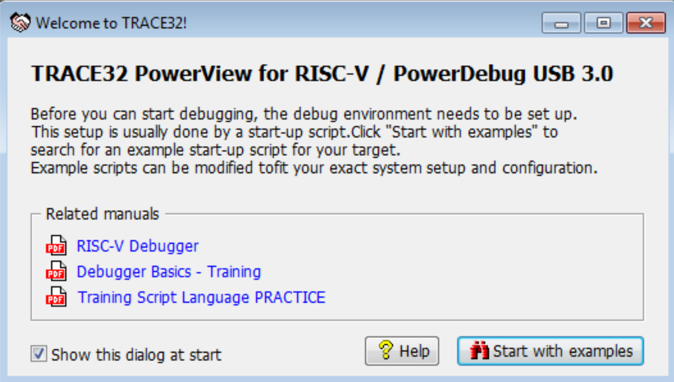
- **"Processor Architecture Manuals"**: These manuals describe commands that are specific for the processor architecture supported by your debug cable. To access the manual for your processor architecture, proceed as follows:
  - Choose **Help** menu > **Processor Architecture Manual**.
- **"OS Awareness Manuals"** (rtos_<*os*>.pdf): TRACE32 PowerView can be extended for operating system-aware debugging. The appropriate OS Awareness manual informs you how to enable the OS-aware debugging.

PRACTICE Script Language:

- **"Training Script Language PRACTICE"** (training_practice.pdf)
- **"PRACTICE Script Language Reference Guide"** (practice_ref.pdf)

©1989-2018 Lauterbach GmbH

To get started with the most important manuals, use the **WELCOME.view** dialog:



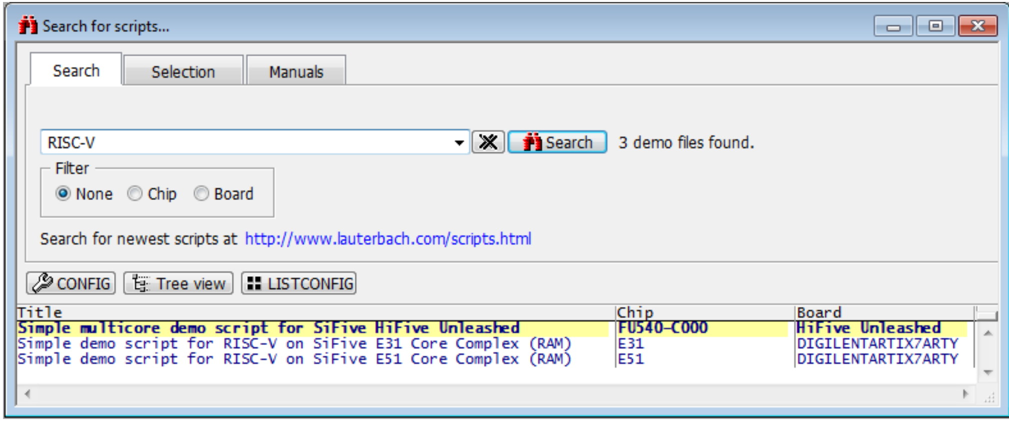
Demo and Start-up Script

Lauterbach provides ready-to-run start-up scripts for known hardware that is based on RISC-V.

To search for PRACTICE scripts, do one of the following in TRACE32 PowerView:

- Type at the command line: **WELCOME.SCRIPTS**
- or choose **File** menu > **Search for Script**.

  You can now search the demo folder and its subdirectories for PRACTICE start-up scripts (*.cmm) and other demo software:



You can also inspect the demo folder manually in the root installation directory of TRACE32. The *~~/demo/riscv/* folder contains:

hardware/	Ready-to-run debugging and flash programming demos for evaluation boards. **Recommended for getting started!**
kernel/	Various OS Awareness examples.

©1989-2018 Lauterbach GmbH

List of Abbreviations and Definitions

CSR	Control and Status Register
DM	Debug Module, as defined by the standard RISC-V debug specification
DTM	Debug Transport Module, as defined by the standard RISC-V debug specification
HART	Hardware thread. A single RISC-V core contains one or multiple hardware threads.

Warning

<table>
<tr>
<td>WARNING:</td>
<td>To prevent debugger and target from damage it is recommended to connect or disconnect the debug cable only while the target power is OFF.

Recommendation for the software start:

1. Disconnect the debug cable from the target while the target power is off.
2. Connect the host system, the TRACE32 hardware and the debug cable.
3. Power ON the TRACE32 hardware.
4. Start the TRACE32 software to load the debugger firmware.
5. Connect the debug cable to the target.
6. Switch the target power ON.
7. Configure your debugger e.g. via a start-up script.

Power down:

1. Switch off the target power.
2. Disconnect the debug cable from the target.
3. Close the TRACE32 software.
4. Power OFF the TRACE32 hardware.</td>
</tr>
</table>

©1989-2018 Lauterbach GmbH

Quick Start of the JTAG Debugger

Starting up the debugger is done as follows:

1. Reset the debugger.

```
RESet
```

   The **RESet** command ensures that no debugger setting remains from a former debug session. All settings get set to their default value. **RESet** is not required if you start the debug session directly after starting the TRACE32 development tool. **RESet** does not reset the target.

2. Select the chip or core you intend to debug.

```
SYStem.CPU <cputype>
```

   Based on the selected chip the debugger sets the **SYStem.CONFIG** and **SYStem.Option** commands the way which should be most appropriate for debugging this chip. Ideally no further setup is required. Please note that the default configuration is not always the best configuration for your target.

3. Connect to target.

```
SYStem.Up
```

   This command establishes the JTAG communication to the target. It resets the processor and enters debug mode (halts the processor; ideally at the reset vector). After this command is executed, it is possible to access memory and registers.

   Some devices can not communicate via JTAG while in reset or you might want to connect to a running program without causing a target reset. In this case use

```
SYStem.Mode Attach
```

   instead. A **Break.direct** will halt the processor.

4. Load the program you want to debug.

```
Data.LOAD <file>
```

   This loads the executable to the target and the debug/symbol information to the debugger's host. If the program is already on the target and you just need the debug/symbol information then load with **/NoCODE** option.

   A detailed description of the **Data.LOAD** command and all available options is given in the **"General Commands Reference"**.

©1989-2018 Lauterbach GmbH

A simple start sequence example is shown below. This sequence can be written to a PRACTICE script (*.cmm, ASCII file format) and executed with the command **DO** *<filename>*.

```
WinCLEAR                              ; Clear all windows

SYStem.CPU FU540-C000                 ; Select the core type

MAP.BOnchip 0x10000++0xffff           ; Specify where FLASH/ROM is

SYStem.Up                             ; Reset the target and enter debug
                                      ; mode

Data.LOAD.Elf riscv_le.elf            ; Load the application

Register.Set PC main                  ; Set the PC to function main

Register.Set X2 0x63FFFFFC            ; Set the stack pointer to address
                                      ; 0x63FFFFFC

Data.List                             ; Open source code window *)

Register /SpotLight                   ; Open register window *)

Frame.view /Locals /Caller            ; Open the stack frame with
                                      ; local variables *)

Var.Watch %SpotLight flags ast        ; Open watch window for variables *)

Break.Set 0x1000 /Program             ; Set software breakpoint to address
                                      ; 0x1000(address 0x1000 outside of
                                      ; BOnchip range)

Break.Set 0x10100 /Program            ; Set on-chip breakpoint
                                      ; to address 0x10100 (address
                                      ; 0x10100 is within BOnchip range)
```

*) These commands open windows on the screen. The window position can be specified with the **WinPOS** command.

©1989-2018 Lauterbach GmbH

Quick Start for Multicore Debugging

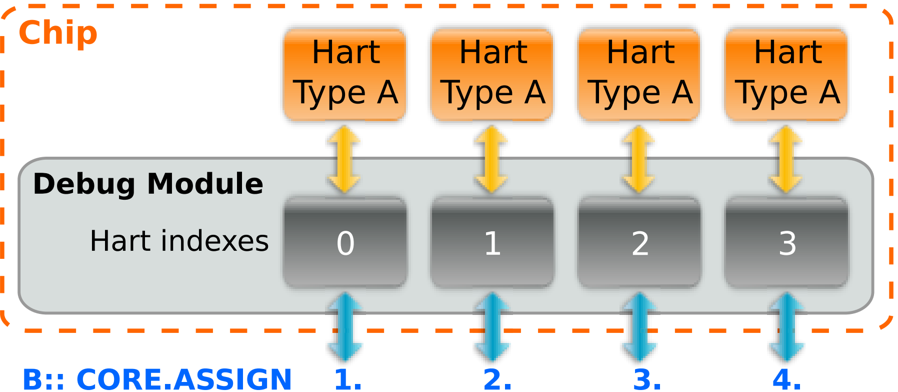
This chapter provides a quick start for multicore processing. The following example scenarios cover the most common use cases:

- **Example A**: SMP Debugging (**S**ymmetric **M**ulti**p**rocessing)
- **Example B**: SMP Debugging (**S**ymmetric **M**ulti**p**rocessing) - Selective
- **Example C**: Homogeneous SMP/AMP Debugging
- **Example D**: Heterogeneous SMP/AMP Debugging

SMP Debugging

This scenario for homogeneous symmetric multiprocessing (SMP) covers the following setup:

4 harts of the same type are connected to the same RISC-V Debug Module of the same chip, with the hart indexes of the RISC-V Debug Module ranging from 0 to 3. All 4 harts will be debugged simultaneously via SMP.



Example A:

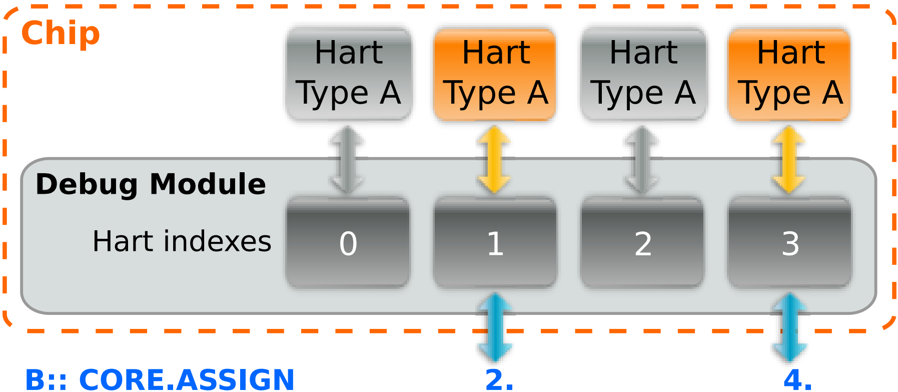
```
SYStem.CPU <type_a_cpu>
SYStem.CONFIG.CORE 1. 1.
SYStem.CONFIG.CoreNumber 4.                       ; 4 harts of type A in total
SYStem.CONFIG.HARTINDEX 0. 1. 2. 3.
CORE.ASSIGN 1. 2. 3. 4.                           ; Assign all 4 harts to the
                                                  ; SMP session
```

©1989-2018 Lauterbach GmbH

SMP Debugging - Selective

This scenario for homogeneous symmetric multiprocessing (SMP) covers the following setup:

4 harts of the same type are connected to the same RISC-V Debug Module of the same chip, with the hart indexes of the RISC-V Debug Module ranging from 0 to 3. The harts with hart indexes 1 and 3 will be debugged simultaneously via SMP.



Example B:

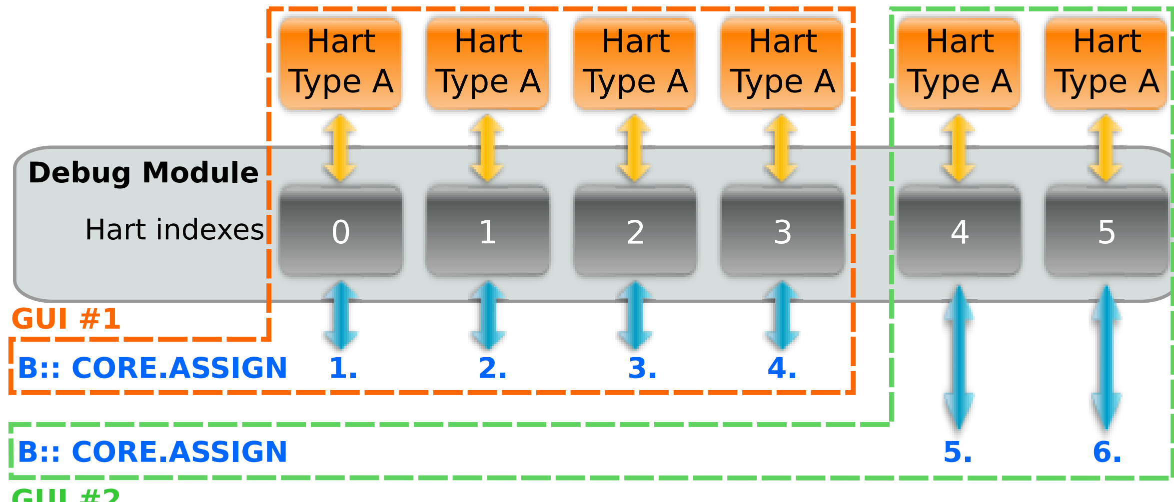
```
SYStem.CPU <type_a_cpu>
SYStem.CONFIG.CORE 1. 1.
SYStem.CONFIG.CoreNumber 4.                 ; 4 harts of type A in total
SYStem.CONFIG.HARTINDEX 0. 1. 2. 3.
CORE.ASSIGN 2. 4.                           ; Assign harts with the
                                            ; logical indexes 2 and 4
```

©1989-2018 Lauterbach GmbH

Homogeneous SMP/AMP Debugging

This scenario covers both homogeneous symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP).

6 harts of the same type are connected to the same RISC-V Debug Module of the same chip, with the hart indexes of RISC-V Debug Module ranging from 0 to 5. The first 4 harts will be debugged in an SMP session, and the remaining 2 harts in another SMP session.



Example C:

```
; ---- TRACE32 PowerView GUI #1 ----------------------------------------

SYStem.CPU <type_a_cpu>
SYStem.CONFIG.CORE 1. 1.
SYStem.CONFIG.CoreNumber 6.                   ; 6 harts of type A in total
SYStem.CONFIG.HARTINDEX 0. 1. 2. 3. 4. 5.
CORE.ASSIGN 1. 2. 3. 4.                       ; Assign the first 4 harts
```

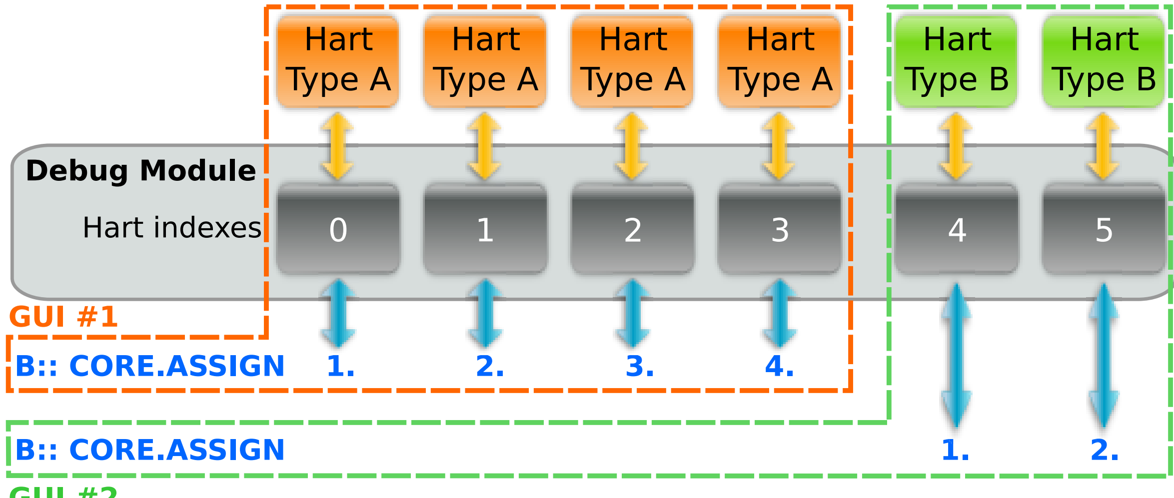
```
; ---- TRACE32 PowerView GUI #2 ----------------------------------------

SYStem.CPU <type_a_cpu>
SYStem.CONFIG.CORE 2. 1.
SYStem.CONFIG.CoreNumber 6.                   ; 6 harts of type A in total
SYStem.CONFIG.HARTINDEX 0. 1. 2. 3. 4. 5.
CORE.ASSIGN 5. 6.                             ; Assign the last 2 harts
```

©1989-2018 Lauterbach GmbH

This scenario covers both heterogeneous symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP).

6 harts are connected to the same RISC-V Debug Module of the same chip, with the hart indexes of the RISC-V Debug Module ranging from 0 to 5. The first 4 harts are of type A and will be debugged in an SMP session, and the remaining 2 harts are of type B and will be debugged in another SMP session.



Example D:

```
; ---- TRACE32 PowerView GUI #1 ----------------------------------------

SYStem.CPU <type_a_cpu>
SYStem.CONFIG.CORE 1. 1.
SYStem.CONFIG.CoreNumber 4.                 ; 4 harts of type A in total
SYStem.CONFIG.HARTINDEX 0. 1. 2. 3.         ; Hart indexes of type A
CORE.ASSIGN 1. 2. 3. 4.                     ; Assign all 4 harts of type A
```

```
; ---- TRACE32 PowerView GUI #2 ----------------------------------------

SYStem.CPU <type_b_cpu>
SYStem.CONFIG.CORE 2. 1.
SYStem.CONFIG.CoreNumber 2.                 ; 2 harts of type B in total
SYStem.CONFIG.HARTINDEX 4. 5.               ; Hart indexes of type B
CORE.ASSIGN 1. 2.                           ; Assign all 2 harts of type B
```

©1989-2018 Lauterbach GmbH

SYStem.Up Errors

The **SYStem.Up** command is the first command of a debug session where communication with the target is required. If you receive error messages while executing this command, this may have the following reasons:

- The target has no power.
- The target is in reset.
- The core is not enabled.
- There is logic added to the JTAG state machine.
- There are additional loads or capacities or serial resistors on the JTAG lines.
- There is a short circuit on at least one of the output lines of the core.
- There are stubs on the signal line.

©1989-2018 Lauterbach GmbH

Debugging via VPN Ref: 0307	**The debugger is accessed via Internet/VPN and the performance is very slow. What can be done to improve debug performance?** The main cause for bad debug performance via Internet or VPN are low data throughput and high latency. The ways to improve performance by the debugger are limited: In PRACTICE scripts, use "SCREEN.OFF" at the beginning of the script and "SCREEN.ON" at the end. "SCREEN.OFF" will turn off screen updates. Please note that if your program stops (e.g. on error) without executing "SCREEN.OFF", some windows will not be updated. "SYStem.POLLING SLOW" will set a lower frequency for target state checks (e.g. power, reset, jtag state). It will take longer for the debugger to recognize that the core stopped on a breakpoint. "SETUP.URATE 1.s" will set the default update frequency of Data.List/Data.dump/Variable windows to 1 second (the slowest possible setting). prevent unneeded memory accesses using "MAP.UPDATEONCE [address-range]" for RAM and "MAP.CONST [address--range]" for ROM/FLASH. Address ranged with "MAP.UPDATEONCE" will read the specified address range only once after the core stopped at a breakpoint or manual break. "MAP.CONST" will read the specified address range only once per SYStem.Mode command (e.g. SYStem.Up).

©1989-2018 Lauterbach GmbH

Setting a Software Breakpoint fails Ref: 0276	**What can be the reasons why setting a software breakpoint fails?** Setting a software breakpoint can fail when the target HW is not able to implement the wanted breakpoint. Possible reasons: The wanted breakpoint needs special features that are only possible to realize by the trigger unit inside the controller. Example: Read, write and access (Read/Write) breakpoints ("type" in Break.Set window). Breakpoints with checking in real-time for data-values ("Data"). Breakpoints with special features ("action") like TriggerTrace, TraceEnable, TraceOn/TraceOFF. TRACE32 can not change the memory. Example: ROM and Flash when no preparation with FLASH.Create, FLASH.TARGET and FLASH.AUTO was made. All type of memory if the memory device is missing the necessary control signals like WriteEnable or settings of registers and SpecialFunctionRegisters (SFR). Contrary settings in TRACE32. Like: MAP.BOnchip for this memory range. Break.SELect.<breakpoint-type> Onchip (HARD is only available for ICE and FIRE). RTOS and MMU: If the memory can be changed by Data.Set but the breakpoint doesn't work it might be a problem of using an MMU on target when setting the breakpoint to a symbolic address that is different than the writable and intended memory location.

©1989-2018 Lauterbach GmbH

Debug Specification for External Debug Support

The Lauterbach debug driver for RISC-V is developed according to the RISC-V debug specification for external debug support. The latest version can be found at
https://github.com/riscv/riscv-debug-spec/

Floating-Point Extensions

The Lauterbach debugger for RISC-V provides support for floating-point extensions of the RISC-V ISA. This covers both the single-precision floating-point extension (“F” extension) and the double-precision floating-point extension (“D” extension).

The floating-point features are provided by the **FPU** (Floating-Point Unit) command group.

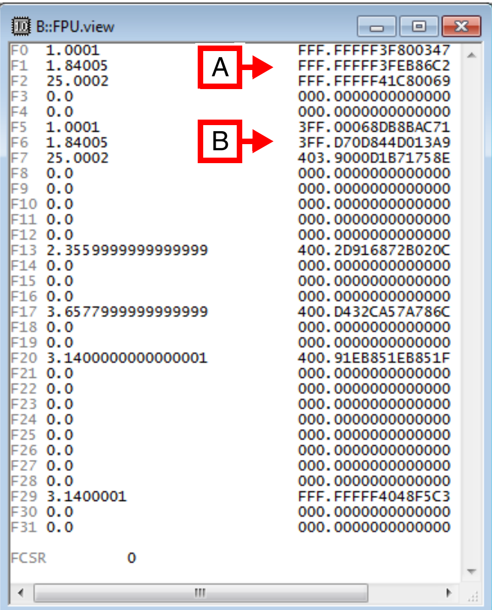
The **FPU.view** window does display the floating-point registers. Depending on whether the core under debug supports single-precision or double-precision, the **FPU.view** window automatically adjusts its register width.

RISC-V floating-point extensions are compliant with the IEEE 754-2008 arithmetic standard. Cores that support the double-precision extension do automatically support the single-precision extension as well. The RISC-V ISA specification defines that a 32 bit single-precision value is stored in a 64 bit double-precision floating-point register by filling up the upper 32 bits of the register with all 1s (Not a Number (NaN) boxing).

When modifying values with **FPU.Set**, the user can decide in which floating-point precision notation the value is written.

The **FPU.view** window does automatically display register values with NaN boxing in single-precision representation, and register values without NaN boxing in double-precision representation. The following example shows 64 bit floating-point registers that contain the same value in both single-precision and double-precision representation:

©1989-2018 Lauterbach GmbH



A Single-precision representation

B Double-precision representation

Breakpoints

For general information about setting breakpoints, refer to the **Break.Set** command.

Software Breakpoints

If a software breakpoint is used, the original code at the breakpoint location is temporarily patched by a breakpoint code (RISC-V *EBREAK* instruction). There is no restriction in the number of software breakpoints.

On-chip Breakpoint Resources

If on-chip breakpoints are used, the resources to set the breakpoints are provided by the hardware of the core itself.

For this purpose, a RISC-V core can have generic on-chip triggers that can either be used for on-chip instruction breakpoints or on-chip data breakpoints. These generic triggers are called “address/data match triggers”. The availability of such triggers is optional, and the number of triggers that are available depends on the respective hardware of the core.

This means that on-chip instruction and on-chip data breakpoints share the number of available trigger resources among each other.

One breakpoint can require one or multiple hardware resources, depending on the complexity of the breakpoint.

Example: We have a core with five address/data match trigger resources, and each breakpoint requires exactly one hardware resource. We can either set five on-chip instruction breakpoints, or we could set three instruction breakpoints and two data breakpoints.

On-chip Breakpoints for Instruction Address

On-chip breakpoints for instruction addresses are used to stop the core when an instruction at a certain address is executed.

The resources to set instruction breakpoints are provided by the hardware of the core. For details about the implementation and number of these breakpoints, see chapter **On-chip Breakpoint Resources**.

On-chip instruction breakpoints are particularly useful in scenarios where the program code lies in read-only memory regions such as ROM or flash, as software breakpoints cannot be used in such scenarios. Furthermore breakpoints for instruction address ranges can only be realized with on-chip breakpoints.

On-chip Breakpoints for Data Address

On-chip breakpoints for data addresses are used to stop the core after a read or write access to a memory address.

©1989-2018 Lauterbach GmbH

The resources to set data address breakpoints are provided by the core. For details about the implementation and number of these breakpoints, see chapter **On-chip Breakpoint Resources**.

On-chip data address breakpoints with address range

Some RISC-V on-chip data address breakpoint triggers allow to set triggers for address ranges. Address ranges for on-chip breakpoint of RISC-V can be implemented in two different ways:

- **Address range via address mask**:
  An address range can be expressed with an address mask, if the range matches the following criteria:

  Let the address range be from address A to address B, with A < B.
  Let X = A XOR B (infix operator XOR: “exclusive or”).
  Let Y = A AND X (infix operator AND: “logical and”).
  Then all bits in X that equal to one have to be in consecutive order, starting from the least significant bit.
  Then Y has to equal zero.

- **Address range via two addresses**:
  An address range can be expressed with a start address and an end address.

An address range via address mask requires less hardware resources than an address range via two addresses. If the criteria for the address mask are met then the debugger will always choose the mask method, in order to save hardware resources.

Examples:

```
Break.Set 0x0000--0x0FFF /Read        ; Address range suitable for
                                      ; address mask

Break.Set 0x0100--0x01FF /Read        ; Address range suitable for
                                      ; address mask

Break.Set 0x3040--0x307F /Write       ; Address range suitable for
                                      ; address mask

Break.Set 0xA000--0xB0FF /Write       ; Address range suitable for
                                      ; two addresses

Break.Set 0xA000--0xA0FD /Write       ; Address range suitable for
                                      ; two addresses
```

On-chip Data Value Breakpoints

The hardware resources of the core can be used to stop the core when a specific value is read or written:

- **Data Value Breakpoint (Read)**:
  Stop the core when a specific data value is read from a memory address.

- **Data Value Breakpoint (Write)**:
  Stop the core when a specific data value is written to a memory address.

For more information about data value breakpoints, see the **Break.Set** command.

©1989-2018 Lauterbach GmbH

Assume you have a target with

- FLASH from `0x0--0xffff`
- RAM from `0x10000--0x3FFF`

The command to set up TRACE32 correctly for this configuration is:

```
MAP.BOnchip 0x0--0xffff
```

The following shows examples for setting standard software breakpoints:

```
Break.Set P:0x20100 /Program            ; Software breakpoint on
                                        ; instruction address

Break.Set main /Program                 ; Software breakpoint on symbol
```

The following shows examples for setting standard on-chip breakpoints:

```
Break.Set P:0x40 /Program               ; On-chip breakpoint on
                                        ; instruction address

Break.Set P:0x40--0x48 /Program         ; On-chip breakpoint on
                                        ; instruction address range

Break.Set D:0x1010 /Read                ; On-chip read breakpoint on
                                        ; data address

Break.Set D:0x1020 /Write               ; On-chip write breakpoint on
                                        ; data address

Break.Set D:0x1030 /ReadWrite           ; On-chip read and write breakpoint
                                        ; on data address

Break.Set D:0x1010--0x101F /Read        ; On-chip read breakpoint on
                                        ; data address range

Break.Set D:0x10 /Read                  ; On-chip read breakpoint on
                 /DATA.Long 0x123       ; data address, combined with
                                        ; condition for read data value
```

©1989-2018 Lauterbach GmbH

Access Classes

For background information about the term *access class*, see **"TRACE32 Glossary"** (glossary.pdf).

The following RISC-V specific access classes are available.

Access Class	Description
P	Program memory access
D	Data memory access
E	Run-time memory access (see **SYStem.CpuAccess** and **SYStem.MemAccess**)
CSR	Control and Status Register (CSR) access. The CSR address of this access class does always address data of maximum CSR register width *XLEN* (see **XLEN()** for details). If the CSR register is smaller than the maximum size, the unused segment gets filled up with zero.
M	Machine privilege level
S	Supervisor privilege level not yet implemented, machine privilege level will be used
U	User privilege level not yet implemented, machine privilege level will be used

To perform an access with a certain access class, write the class in front of the address.

Example:

```
Data.dump D:0x0--0x3
Data.dump SD:0x0--0x3
PRINT Data.Long(CSR:0x300)
```

©1989-2018 Lauterbach GmbH

SYStem.CONFIG.state Display target configuration

Format:	**SYStem.CONFIG.state** [**/**<*tab*>]
<*tab*>:	**DebugPort** | **Jtag**

Opens the **SYStem.CONFIG.state** window, where you can view and modify most of the target configuration settings. The configuration settings tell the debugger how to communicate with the chip on the target board and how to access the on-chip debug and trace facilities in order to accomplish the debugger's operations.

Alternatively, you can modify the target configuration settings via the TRACE32 command line with the **SYStem.CONFIG** commands. Note that the command line provides *additional* **SYStem.CONFIG** commands for settings that are *not* included in the **SYStem.CONFIG.state** window.

<*tab*>	Opens the **SYStem.CONFIG.state** window on the specified tab. For tab descriptions, refer to the tab descriptions below.
DebugPort	The **DebugPort** tab (default) informs the debugger about the debug connector type and the communication protocol it shall use. For descriptions of the commands on the **DebugPort** tab, see **DebugPort**.
Jtag	The **Jtag** tab informs the debugger about the position of the Test Access Ports (TAP) in the JTAG chain which the debugger needs to talk to in order to access the debug and trace facilities on the chip. For descriptions of the commands on the **Jtag** tab, see **Jtag**.

©1989-2018 Lauterbach GmbH

Format:	**SYStem.CONFIG** *<parameter>*
<parameter>: **(DebugPort)**	**CORE** *<core> <chip>* **CoreNumber** *<number>* **DEBUGPORT** [**DebugCable0**] **DEBUGPORTTYPE** [**JTAG**] **Slave** [**ON** | **OFF**] **TriState** [**ON** | **OFF**]
<parameter>: **(JTAG)**	**DRPOST** *<bits>* **DRPRE** *<bits>* **IRPOST** *<bits>* **IRPRE** *<bits>* **Slave** [**ON** | **OFF**] **TAPState** *<state>* **TCKLevel** *<level>* **TriState** [**ON** | **OFF**]

The **SYStem.CONFIG** commands inform the debugger about the available on-chip debug and trace components and how to access them.

The **SYStem.CONFIG** command information shall be provided after the **SYStem.CPU** command which might be a precondition to enter certain **SYStem.CONFIG** commands and before you start up the debug session, e.g. by **SYStem.Up**.

Syntax remarks:

The commands are not case sensitive. Capital letters show how the command can be shortened.
Example: “SYStem.CONFIG.TriState ON” -> “SYStem.CONFIG.TS ON”

The dots after “SYStem.CONFIG” can alternatively be a blank.
Example:
“SYStem.CONFIG.TriState ON” or “SYStem.CONFIG TriState ON”

©1989-2018 Lauterbach GmbH

CORE *<core>* *<chip>*	The command helps to identify debug and trace resources which are commonly used by different cores. The command might be required in a multicore environment if you use multiple debugger instances (multiple TRACE32 PowerView GUIs) to simultaneously debug different cores on the same target system. Because of the default setting of this command debugger#1: *<core>*=1 *<chip>*=1 debugger#2: *<core>*=1 *<chip>*=2 ... each debugger instance assumes that all notified debug and trace resources can exclusively be used. But some target systems have shared resources for different cores. For example a common trace port. The default setting causes that each debugger instance will control the (same) trace port. Sometimes it does not hurt if such a module will be controlled twice. So even then it might work. But the correct specification which might be a must is to tell the debugger that these cores sharing resources are on the same *<chip>*. Whereby the "chip" does not need to be identical with the device on your target board: debugger#1: *<core>*=1 *<chip>*=1 debugger#2: *<core>*=2 *<chip>*=1
CORE *<core>* *<chip>* (cont.)	For cores on the same *<chip>* the debugger assumes they share the same resource if the control registers of the resource has the same address. Default: *<core>* depends on CPU selection, usually 1. *<chip>* derives from the `CORE=` parameter in the configuration file (config.t32), usually 1. If you start multiple debugger instances with the help of t32start.exe you will get ascending values (1, 2, 3,...).
CoreNumber *<number>*	Number of cores to be considered in an SMP (symmetric multiprocessing) debug session. There are RISC-V core types which can be used as a single core processor or as a scalable multicore processor of the same type. If you intend to debug more than one such core in an SMP debug session you need to specify the number of cores you intend to debug. Default: 1.

©1989-2018 Lauterbach GmbH

DEBUGPORT [**DebugCable0**]	It specifies which probe cable shall be used e.g. "DebugCable0". At the moment only the CombiProbe allows to connect more than one probe cable. Default: depends on detection.
DEBUGPORTTYPE [**JTAG**]	It specifies the used debug port type "JTAG". It assumes the selected type is supported by the target. Default: JTAG.
Slave [**ON** | **OFF**]	If several debuggers share the same debug port, all except one must have this option active. JTAG: Only one debugger - the "master" - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave OFF**. Default: OFF. Default: ON if `CORE=`... >1 in the configuration file (e.g. config.t32).
TriState [**ON** | **OFF**]	TriState has to be used if several debug cables are connected to a common JTAG port. **TAPState** and **TCKLevel** define the TAP state and TCK level which is selected when the debugger switches to tristate mode. Please note: • nTRST must have a pull-up resistor on the target. • TCK can have a pull-up or pull-down resistor. • Other trigger inputs need to be kept in inactive state. Default: OFF.

©1989-2018 Lauterbach GmbH

<parameters> describing the "JTAG" scan chain and signal behavior

With the JTAG interface you can access a Test Access Port controller (TAP) which has implemented a state machine to provide a mechanism to read and write data to an Instruction Register (IR) and a Data Register (DR) in the TAP. The JTAG interface will be controlled by 5 signals: nTRST(reset), TCK (clock), TMS (state machine control), TDI (data input), TDO (data output). Multiple TAPs can be controlled by one JTAG interface by daisy-chaining the TAPs (serial connection). If you want to talk to one TAP in the chain you need to send a BYPASS pattern (all ones) to all other TAPs. For this case the debugger needs to know the position of the TAP it wants to talk to. The TAP position can be defined with the first four commands in the table below.

DRPOST *<bits>*	Defines the TAP position in a JTAG scan chain. Number of TAPs in the JTAG chain between the TDI signal and the TAP you are describing. In BYPASS mode each TAP contributes one data register bit. See example below. Default: 0.
DRPRE *<bits>*	Defines the TAP position in a JTAG scan chain. Number of TAPs in the JTAG chain between the TAP you are describing and the TDO signal. In BYPASS mode each TAP contributes one data register bit. See example below. Default: 0.
IRPOST *<bits>*	Defines the TAP position in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between TDI signal and the TAP you are describing. See example below. Default: 0.
IRPRE *<bits>*	Defines the TAP position in a JTAG scan chain. Number of Instruction Register (IR) bits of all TAPs in the JTAG chain between the TAP you are describing and the TDO signal. See example below. Default: 0.
Slave [**ON** | **OFF**]	If several debuggers share the same debug port, all except one must have this option active. JTAG: Only one debugger - the "master" - is allowed to control the signals nTRST and nSRST (nRESET). The other debuggers need to have the setting **Slave OFF**. Default: OFF. Default: ON if `CORE=...` >1 in the configuration file (e.g. config.t32).

TAPState *<state>*

This is the state of the TAP controller when the debugger switches to tristate mode. All states of the JTAG TAP controller are selectable.

0 Exit2-DR
1 Exit1-DR
2 Shift-DR
3 Pause-DR
4 Select-IR-Scan
5 Update-DR
6 Capture-DR
7 Select-DR-Scan
8 Exit2-IR
9 Exit1-IR
10 Shift-IR
11 Pause-IR
12 Run-Test/Idle
13 Update-IR
14 Capture-IR
15 Test-Logic-Reset

Default: 7 = Select-DR-Scan.

TCKLevel *<level>*

Level of TCK signal when all debuggers are tristated. Normally defined by a pull-up or pull-down resistor on the target.

Default: 0.

TriState [**ON** | **OFF**]

TriState has to be used if several debug cables are connected to a common JTAG port. **TAPState** and **TCKLevel** define the TAP state and TCK level which is selected when the debugger switches to tristate mode.
Please note:

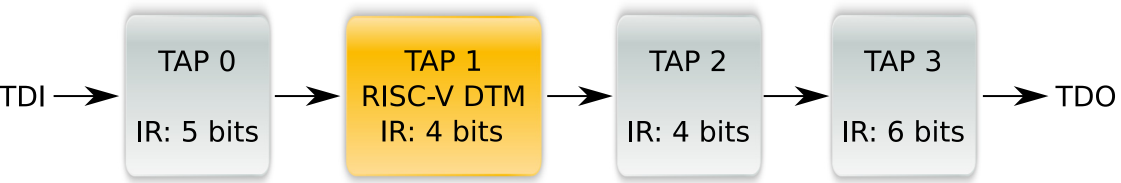
- nTRST must have a pull-up resistor on the target.
- TCK can have a pull-up or pull-down resistor.
- Other trigger inputs need to be kept in inactive state.

Default: OFF.

NOTE:	If you are not sure about your settings concerning **IRPRE**, **IRPOST**, **DRPRE**, and **DRPOST**, you can try to detect the settings automatically with the **SYStem.DETECT.DaisyChain** command.

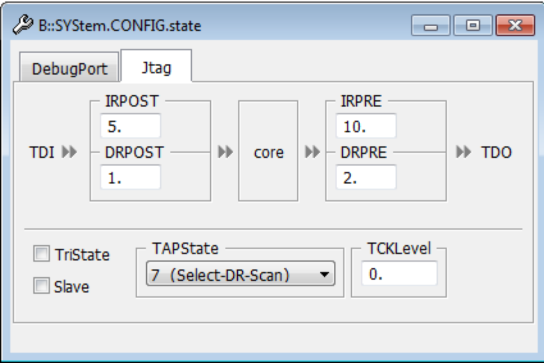
©1989-2018 Lauterbach GmbH

Example:



This example shows four TAPs in a JTAG daisy chain. The relevant TAP for RISC-V debugging is the Debug Transport Module (DTM) TAP. In order to address this TAP, the following settings are necessary:

```
SYStem.CONFIG IRPRE  10.
SYStem.CONFIG IRPOST  5.
SYStem.CONFIG DRPRE   2.
SYStem.CONFIG DRPOST  1.
```



Format:	**SYStem.CONFIG.HARTINDEX** *<index>*
<index>:	**0.** | **1.** … *n*

Default: 0.

Configures the hardware thread (hart) index that is used by the Debug Module to address one or more harts.

The command requires an index for each hart that is covered by **SYStem.CONFIG.CoreNumber**.

Example:

```
SYStem.CONFIG.CoreNumber 5.
SYStem.CONFIG.HARTINDEX 3. 4. 5. 6. 7.
```

For further examples, see **“Quick Start for Multicore Debugging”**, page 10.

©1989-2018 Lauterbach GmbH

Format:	**SYStem.CPU** *<cpu>*
<cpu>:	**RV32** | **RV64** | …

RV32 and **RV64** are *default* entries for the 32-bit and 64-bit RISC-V cores respectively. These entries should only be selected if the respective debug IP of the target does conform to the official RISC-V debug specification, and if no dedicated *<cpu>* entry for the respective target CPU or board is available.

<cpu>	For a list of supported CPUs, use the command `SYStem.CPU *` or refer to the chip search on the Lauterbach website.

SYStem.CpuAccess   Run-time memory access (intrusive)

Format:	**SYStem.CpuAccess Enable** | **Denied** | **Nonstop**

Default: Denied.

This option declares if an *intrusive* memory access can take place while the CPU is executing code. To perform this access, the debugger stops the CPU shortly, performs the access and then restarts the CPU.

The run-time memory access has to be activated for each window by using the memory class E: (e.g. `Data.dump ED:0xA1000000`) or by using the format option `%E` (e.g. `Var.View %E var1`).

Enable	Allows intrusive run-time memory access. In order to perform a memory read or write while the CPU is executing the program, the debugger stops the program execution shortly. Each short stop takes 1 … 100 ms depending on the speed of the debug interface and on the number of the read/write accesses required. A red S in the state line of the TRACE32 main window indicates this intrusive behavior of the debugger.

©1989-2018 Lauterbach GmbH

Denied	Locks intrusive run-time memory access.
Nonstop	Locks all features of the debugger that affect the run-time behavior. Nonstop reduces the functionality of the debugger to: • Run-time access to memory and variables • Trace display The debugger inhibits the following: • To stop the program execution • All features of the debugger that are intrusive (e.g. action Spot for breakpoints, performance analysis via StopAndGo mode, conditional breakpoints, etc.)

SYStem.JtagClock — Define JTAG frequency

Format:	**SYStem.JtagClock** [*<frequency>*]
<frequency>:	**10000.** … **40000000.**

Default frequency: 10 MHz.

Selects the JTAG port frequency (TCK) used by the debugger to communicate with the processor. The frequency affects e.g. the download speed. It could be required to reduce the JTAG frequency if there are buffers, additional loads or high capacities on the JTAG lines or if VTREF is very low. A very high frequency will not work on all systems and will result in an erroneous data transfer.

<frequency>	The debugger cannot select all frequencies accurately. It chooses the next possible frequency and displays the real value in the **SYStem.state** window. Besides a decimal number like “100000.” also short forms like “10kHz” or “15MHz” can be used. The short forms imply a decimal value, although no “.” is used.

©1989-2018 Lauterbach GmbH

SYStem.LOCK
Tristate the JTAG port

Format:	**SYStem.LOCK** [**ON** | **OFF**]

Default: OFF.

If the system is locked no access to the JTAG port will be performed by the debugger. While locked the JTAG connector of the debugger is tristated. The intention of the lock command is for example to give JTAG access to another tool. The process can also be automated, see **SYStem.CONFIG TriState**.

It must be ensured that the state of the RISC-V DTM JTAG state machine remains unchanged while the system is locked. To ensure correct hand over the options **SYStem.CONFIG TAPState** and **SYStem.CONFIG TCKLevel** must be set properly. They define the TAP state and TCK level which is selected when the debugger switches to tristate mode.

SYStem.MemAccess
Run-time memory access (non-intrusive)

Format:	**SYStem.MemAccess** *<mode>*
<mode>:	**CPU** | **Denied**

Default: Denied.

This option declares if and how a *non-intrusive* memory access can take place while the CPU is executing code. Although the CPU does not get halted, run-time memory access creates an additional load on the processor's internal data bus. The currently selected run-time memory access mode is displayed in the state line.

The run-time memory access has to be activated for each window by using the memory class **E:** (e.g. **Data.dump** **ED:0xA1000000**) or by using the format option **%E** (e.g. **Var.View** **%E var1**).

CPU	The debugger performs non-intrusive memory accesses via the CPU.
Denied	Non-intrusive memory access is disabled while the CPU is executing code. Instead, intrusive accesses can be configured with **SYStem.CpuAccess**.

©1989-2018 Lauterbach GmbH

Format:	**SYStem.Option MMUSPACES** [**ON** | **OFF**] **SYStem.Option MMUspaces** [**ON** | **OFF**] (deprecated) **SYStem.Option MMU** [**ON** | **OFF**] (deprecated)

Default: OFF.

Enables the use of space IDs for logical addresses to support **multiple** address spaces.

For an explanation of the TRACE32 concept of address spaces (zone spaces, MMU spaces, and machine spaces), see **"TRACE32 Glossary"** (glossary.pdf).

NOTE:	**SYStem.Option MMUSPACES** should not be used if only one translation table is used on the target. If a debug session requires space IDs, you must observe the following sequence of steps: 1. Activate **SYStem.Option MMUSPACES**. 2. Load the symbols with **Data.LOAD**. Otherwise, the internal symbol database of TRACE32 may become inconsistent.

Examples:

```
;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x012A:
Data.dump D:0x012A:0xC00208A

;Dump logical address 0xC00208A belonging to memory space with
;space ID 0x0203:
Data.dump D:0x0203:0xC00208A
```

©1989-2018 Lauterbach GmbH

Format:	**SYStem.Mode** *<mode>*
<mode>:	**Down** **Prepare** **Go** **Attach** **Up**

Down (default) — Disables the debugger. The state of the CPU remains unchanged. The JTAG port is tristated.

Prepare — Initializes a debug connection.
The debugger does initialize the debug IP, but it does *not* perform any interaction with the CPU.

This debug mode is used if the CPU shall not be debugged or it shall be bypassed. In that case the debugger can still access the memory, e.g. via direct bus access. However, any operation that could alter the CPU state or would require CPU interaction (such as accessing GPR or CSR registers) is not possible in this debug mode.

Go — Initializes a debug connection, resets the target and lets the CPU run from its reset vector.

Attach — Initializes a debug connection. The target is *not* reset, i.e. the state of the target is *not* changed. Consequently the user program stays running if it was running, or stays stopped if it was stopped.

Up — Initializes a debug connection, resets the target, sets the CPU to debug mode and stops the CPU at its reset vector.

©1989-2018 Lauterbach GmbH

SYStem.Option Special setup

[SYStem.state window > Option]

The **SYStem.Option** commands are used to control special features of the debugger or emulator or to configure the target. It is recommended to execute the **SYStem.Option** commands **before** the emulation is activated by a **SYStem.Up** or **SYStem.Mode** command.

SYStem.Option Address32 Define address format display

Format:	**SYStem.Option Address32** [**ON** | **OFF** | **AUTO**]

Default: Depending on the CPU selected with the **SYStem.CPU** command.

Selects the number of displayed address digits in various windows, e.g. **List.auto** or **Data.dump**.

ON	Display all addresses as 32-bit values. 64-bit addresses are truncated.
OFF	Display all addresses as 64-bit values.
AUTO	Number of displayed digits depends on address size.

SYStem.Option EnReset Allow the debugger to drive nRESET (nSRST)

[SYStem.state window> EnReset]

Format:	**SYStem.Option EnReset** [**ON** | **OFF**]

Default: ON.

If this option is disabled the debugger will never drive the nRESET (nSRST) line on the JTAG connector. This is necessary if nRESET (nSRST) is no open collector or tristate signal.
Instead, during a **SYStem.Up**, the debugger will only assert a soft system reset via the “non-debug module reset” bit (*ndmreset*) of the *dmcontrol* register.

If this option is enabled the debugger will perform both reset options (assert nRESET line and set ndmreset bit) during a **SYStem.Up**.

©1989-2018 Lauterbach GmbH

SYStem.Option IMASKASM — Disable interrupts while single stepping

Format: **SYStem.Option IMASKASM** [**ON** | **OFF**]

Default: ON.

If enabled, the Step Interrupt Enable Bit will be cleared during assembler single-step operations. The interrupt routine is not executed during single-step operations.

SYStem.Option TRST — Allow debugger to drive TRST

[SYStem.state window > TRST]

Format: **SYStem.Option TRST** [**ON** | **OFF**]

Default: ON.

If this option is disabled, the nTRST line is never driven by the debugger (permanent high). Instead five consecutive TCK pulses with TMS high are asserted to reset the TAP controller which have the same effect.

SYStem.state — Display SYStem.state window

Format: **SYStem.state**

Displays the **SYStem.state** window for system settings that configure debugger and target behavior.

©1989-2018 Lauterbach GmbH

FPU.Set

Write to FPU register

Format:	**FPU.Set** *<register>*[**.***<precision>*] [*<expression>* | *<float>*]
<register>:	**F0** | **F1** … **F31**
<precision>:	**auto** **Single** **Double**

Writes to a floating-point register of the RISC-V core under debug.

auto	Automatic detection of the floating-point precision. The debugger automatically detects whether the current value of *<register>* is single-precision or double-precision, and uses the detected precision for the register write. • If single-precision is detected, **FPU.Set** *<register>***.auto** is equal to **FPU.Set** *<register>***.Single**. • If double-precision is detected, **FPU.Set** *<register>***.auto** is equal to **FPU.Set** *<register>***.Double**.
Single	Uses single-precision floating-point representation for the register write.
Double	Uses double-precision floating-point representation for the register write.
<float>	**Parameter Type**: Float.
<expression>	**Parameter Type**: Decimal or hex.

Example:

```
FPU.Set F4.auto   1.4      ; Write to register with
                           ; automatic detection of precision
FPU.Set F4.Single 2.7      ; Write to register with single-precision
FPU.Set F4.Double 3.2      ; Write to register with double-precision

FPU.Set F6.Single 0xABCD   ; Write to register with single-precision
                           ; in hexadecimal notation
FPU.Set F6.Double 12.      ; Write to register with double-precision
                           ; in decimal notation
```

©1989-2018 Lauterbach GmbH

XLEN() Current width of XLEN

[build 97333 - DVD 09/2018]

Syntax:	**XLEN()**

Returns the value of XLEN, i.e. the current width (in bits) of the general purpose registers of the core under debug. For further details about XLEN, please see the official RISC-V ISA specification.

Return Value Type: Decimal value.

Example:

```
PRINT XLEN()
```

©1989-2018 Lauterbach GmbH

Target Adaption

Connector Type and Pinout

The **Debugger for RISC-V** is shipped with a debug cable that supports the standard “RISC-V JTAG Debug Transport Module (DTM)”. This debug cable consists of two parts:

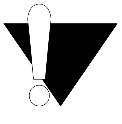
- **RISC-V debug cable with 20 pin connector**
- **Converter ARM-20 to ALTERA-10/RISCV-10**

The RISC-V debug cable can either be used alone or it can be extended with the ARM-20 to ALTERA-10/RISCV-10 converter. The target side of the converter conforms to the recommended JTAG interface of the RISC-V debug specification (For more details, see chapter **Debug Specification for External Debug Support**).

RISC-V Debug Cable with 20 pin Connector

Adaption for RISC-V Debug Cable: See **www.lauterbach.com/adriscv.html**

Signal	Pin	Pin	Signal
VREF-DEBUG	1	2	N/C
TRST-	3	4	GND
TDI	5	6	GND
TMS	7	8	GND
TCK	9	10	GND
RTCK	11	12	GND
TDO	13	14	GND
RESET-	15	16	GND
N/C	17	18	GND
N/C	19	20	GND



Pin 2, pin 17 and pin 19 must under no circumstances be connected on the target side. Otherwise the hardware of the debugger can get damaged.

©1989-2018 Lauterbach GmbH

Signal	Pin	Pin	Signal
TCK	1	2	GND
TDO	3	4	VREF-DEBUG
TMS	5	6	[RESET-]
N/C	7	8	[TRST-]
TDI	9	10	GND

NOTE: Pin 6 and pin 8 are optional and can be selected/deselected by jumpers.

©1989-2018 Lauterbach GmbH

Support

Lauterbach technical support is available via **www.lauterbach.com/tsupport.html**.

For support requests that concern RISC-V, contact **bdmriscv-support@lauterbach.com**

Available Tools

CPU	ICE	FIRE	ICD DEBUG	ICD MONITOR	ICD TRACE	POWER INTEGRATOR	INSTRUCTION SIMULATOR
BM-310			YES				
E31			YES				
E51			YES				
FU540-C000			YES				
N25			YES				
NX25			YES				

Compilers

Language	Compiler	Company	Option	Comment
C++	GCC	Free Software Foundation, Inc.	ELF/DWARF	

©1989-2018 Lauterbach GmbH

Products

Product Information

OrderNo Code	Text
LA-2717 JTAG-RISC-V	**Debugger for RISC-V (ICD)** supports RISC-V includes software for Windows, Linux and MacOSX requires Power Debug Module
LA-2717A JTAG-RISC-V-A	**JTAG Debugger for RISC-V Add.** supports RISC-V only suitable for debug cables newer than 07/2008 includes software for Windows, Linux and MacOSX

Order Information

Order No.	Code	Text
LA-2717	**JTAG-RISC-V**	**Debugger for RISC-V (ICD)**
LA-2717A	**JTAG-RISC-V-A**	**JTAG Debugger for RISC-V Add.**
Additional Options		
LA-7960X	MULTICORE-LICENSE	License for Multicore Debugging

©1989-2018 Lauterbach GmbH