cadence®

Virtual Debug Interface User Guide

Product Version 48
April 2023

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

product documentation may however contain material that is no longer considered appropriate but still continues to reflect long-standing industry terminology. Such content will be addressed at a time when the related software can be updated without end-user impact.

Contents

1

Overview of Virtual Debug Interface

This document describes the Virtual Debug interface (version 48), which connects a software debugger to the CPU that is simulated or emulated by the Palladium system.

The Virtual Debug interface consists of the following parts:

- A shared library that is loaded by the software debugger
- A shared library that is loaded by the simulator or Palladium emulator
- A Bus Functional Model instantiated in the design

This product version supports the following protocols:

- JTAG (Joint Test Action Group)
- AMBA APB (Advanced Peripheral Bus)
- DAP (Debug Access Port)
- AMBA AHB (Advanced High-Performance Bus)
- AMBA AXI (Advanced eXtensible Interface)

This product version supports the following debuggers:

- Lauterbach Trace32
- Cadence Xtensa OCD
- ARM Development Studio
- Green Hills MULTI
- OpenOCD

The interface package includes the following files.

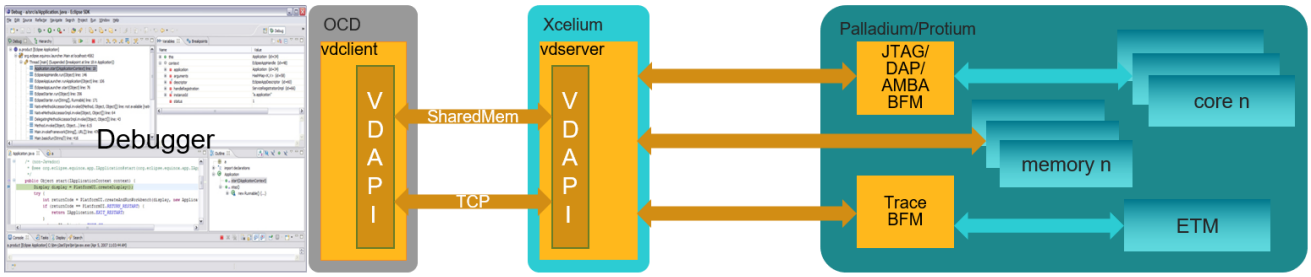
File	Description
`doc/*`	This documentation
`script/*.cmm, *.t32`	Trace32 configuration and practice scripts
`script/*.xml`	Xtensa OCD configuration and practice scripts
`v/vd_*_bfm.svp`	Encrypted BFM: JTAG, AMBA, SWDP, DAP6 and TPIU
`lib/32bit/libvdgtl.so`	Trace32 32-bit library Linux
`lib/32bit/libvdgtl.dll`	Trace32 32-bit library Windows
`lib/32bit/libvdserver.so`	Server 32-bit DPI library
`lib/32bit/libvdserver.nc.so`	Simulator 32-bit DPI library
`lib/32bit/libvdxocd.so`	Xtensa OCD 32-bit library Linux
`lib/32bit/libvdxocd.dll`	Xtensa OCD 32-bit library Windows
`lib/64bit/libvdgtl.so`	Trace32 64-bit library Linux
`lib/64bit/libvdgtl.dll`	Trace32 64-bit library Windows
`lib/64bit/libvdserver.so`	Server 64-bit DPI library
`lib/64bit/libvdserver.nc.so`	Simulator 64-bit DPI library
`lib/64bit/libvdxocd.so`	Xtensa OCD 64-bit library Linux
`lib/64bit/libvdxocd.dll`	Xtensa OCD 64-bit library Windows
`lib/64bit/libvdrddi.so.2`	Development Studio 64-bit library Linux
`lib/64bit/vdrddi_2.dll`	Development Studio 64-bit library Windows
`example/test_jtag`	JTAG connectivity test example
`example/ibex_pulp_j`	RISCV-based test example
`example/setup_example.csh`	Environment setup example

Virtual Debug Interface Environment

The Virtual Debug interface transfers the debugger requests in a transactional format to the simulated (or emulated) processor. The interface part loaded by the debugger is called a plug-in client. It communicates with the server loaded by the simulator through either shared memory or TCP sockets. The server library uses the SystemVerilog DPI and Direct Memory Access interfaces to communicate with the emulated processor core. It must run on a workstation with an Infiniband (IB) channel to either Palladium or Protium.

The debugger can run on a different host (either Linux or Windows) when TCP sockets are used. In such cases, the bit mode of the debugger does not need to match the compiled database.

The following diagram illustrates the Virtual Debug interface environment.



Related Sections:

- Software Compatibility for Virtual Debug Interface
- License Requirements for Virtual Debug Interface and Debuggers
- Integrating BFM Modules with the Design
- Compiling the Design
- Running and Bringing-Up the Virtual Debug Interface

2

Software Compatibility for Virtual Debug Interface

This section lists the software with which the Virtual Debug Interface tool is compatible.

Client Plug-In and Library	The OS platform is determined mostly by the software debugger. • 32-bit or 64-bit Linux distribution with 2.6 kernel or later • 32-bit or 64-bit Windows 7 or later
Palladium, Protium, and Xcelium	The Virtual Debug interface has been tested and is intended to run with the following software: • Palladium Z1: VXE22.04 and later • Palladium Z2: WXE22.04 and later • Protium X1 / X2: PTM22.05 and later • Xcelium: XCELIUM22.03.001 and later If a newer Cadence VXE, WXE, or PTM software is used, consult the VXE, WXE, or PTM documentation on the Xcelium release-version interoperability. For Palladium and Protium, the compatible versions of IXCOM and HDLICE releases, such as IXCOM22.04 and HDLICE22.04 should be used.
Lauterbach Trace32	• Trace32 2019.1. Cadence recommends to use the latest release. • Trace32 supports both Linux and Windows platforms and provides several binaries for different processor architectures.

Xtensa OCD	• Xtensa OCD 14.* and later versions • Xtensa tools RI 2020.* or later Virtual Debug has been tested with Xtensa OCD 14.0.5. It is also expected to work with other releases.
Arm Development Studio	Development Studio 2021.2. or later releases

Green Hills MULTI	• MULTI 7 and newer with compiler 2015.1 and later versions
	• Two options are available for virtual debug:
	  ◦ Using a virtualized probe (vprobe)
	  Contact GHS support for licensing and patch requirements.
	  ◦ Using an existing Green Hills Probe as a proxy
	  This requires Firmware 6.4 or newer and a patch.

OpenOCD	• OpenOCD 0.12, which includes vdebug client is required. • The Open On-Chip Debugger (OpenOCD) is an open source software that can be downloaded from `http://openocd.org/`. • To use OpenOCD with vdebug, the `--enable-vdebug=yes` option needs to be included in configuration step when building OpenOCD.

3

License Requirements for Virtual Debug Interface and Debuggers

Following are the license requirements for the Virtual Debug interface and debuggers.

Virtual Debug Interface License	When the debugger connects to the core, the Virtual Debug interface attempts to check out an `ACC_VDEBUG` license version 22.00 from the license server. If the license is not available, an error is reported by the debugger and the debugger fails to connect. The license is released when the debugger disconnects from the CPU.

Trace32 License	Lauterbach Trace32 needs a set of licenses to work. The minimum requirements are a front-end license and a GTL back-end license. The exact type of license that is required depends on the type of the core to connect.
	Specify the license using one of the following ways:
	• Using the environment variable:
	`export RLM_LICENSE=<port>@<host>`
	• Using the `client.lic` file in the current directory with content:
	`HOST <host> PORT <port>`
	For details, refer to the Trace32 Installation Guide.

Arm Development Studio License	ARM Development Studio needs a set of licenses, depending on the edition: Bronze, Silver, Gold, or Platinum. The license files can be set in the *Help/Arm License Manager* menu. No additional license for Virtual Debug connection is required.
MULTI License	MULTI requires a set of licenses for the corresponding target architecture and a `virtual_probe` license for the vprobe software, if applicable. Contact GHS support for licensing and patch requirements. No additional license for Virtual Debug connection is required.

Xtensa OCD and OpenOCD do not need a separate license to work.

4

Migrating from Physical JTAG

If there is an existing design with the JTAG signals physically connected to the workstation with the software debugger, several changes are needed to use the Virtual Debug Interface.

The software debugger reads out data from the emulated design, the design clocks stop, and the data is transferred between the emulator and the software. This is called **synchronization**. The performance of the Virtual Debug Interface is determined and limited by the synchronizations between the hardware and the software.

Observe the following steps to get the best performance:

1. Use the fastest design clock to drive the vdebug JTAG BFM. The JTAG BFM generates the JTAG clock `tclk`, and is equipped with the internal divider to scale the JTAG clock, based on the settings from the software debugger. Since all the clocks in the transactor are controlled and stopped for every data exchange, there is no risk of data loss. If the design cannot accept the fast JTAG clock, the clock could be scaled down at run time.
2. Adjust the software debugger settings. For details, refer to the software debugger specific sections.
    a. Change the timing from real time to virtual (simulation) time, if such setting is available.
    b. Disable or adjust timeouts, based on real time.
    c. Reduce the amount of data—the software debugger reads and writes periodically. This operation includes refresh and polling cycles.
3. Use the parallel vdebug interfaces, instead of the serial JTAG interface. This drastically reduces the number of synchronizations and increases the interface performance.

ⓘ For information about the best case performance scenarios, refer to the following sections:

- Palladium Performance Numbers
- Protium Performance Numbers
- Improving Transactor Performance

Related Sections:

- Configuring and Running Lauterbach Trace 32
- Configuring and Running Development Studio
- Configuring and Running Green Hills MULTI
- Configuring and Running OpenOCD

5

Integrating BFM Modules with the Design

The BFM needs to be compiled with the design. It can be activated separately using the reset signal. The vdebug server is active only if at least one compiled BFM is activated (not held in reset state).

When the interface signals are easily available on the design interface (applies most likely to JTAG), the BFM instance can be connected directly. Another option to integrate the BFM is to use the *net cut* methodology, in which a Verilog `force` command is used to drive the signals from the BFM without a need to modify the RTL. This method, which allows a run-time activation, is suitable when multiple BFMs connect to the same data in the design. Following is a snippet of RTL code that allows run-time activation through a signal from the TCL script.

```
reg bfm_enable = 1'b0;
`ifdef IXCOM_UXE
initial $export_deposit( bfm_enable );
`endif
reg out1, out2;

`define DUTPATH top.mod1.mod2.mod3.mod4.mod5
vd_swdp_bfm u_vd_swdp_bfm(
.insig1(`DUTPATH.sig1),
.insig2(`DUTPATH.sig2),
.outsig1(out1),
.outsig2(out2)
);

always @( bfm_enable )
if( bfm_enable)
begin
force `DUTPATH.dutsig1 = out1;
force `DUTPATH.dutsig2 = out2;
end
else
begin
release `DUTPATH.dutsig1;
release `DUTPATH.dutsig2;
end
```

The details of BFM integration are described in the following sections:

- JTAG Integration
- APB Integration
- DAP5 Integration
- DAP6 Integration
- AHB Integration
- AXI Integration
- TPIU Integration

JTAG Integration

A module named `vd_jtag_bfm`, implemented in SystemVerilog, provides a signal-level interface to the design (with the TAP controller). The following table describes the signals of this module.

Signal	Direction (from BFM)	Description
`clk_i`	Input	Specifies the internal transactor clock.
`rst_i`	Input	Specifies the asynchronous reset as active low. This signal can be used to dynamically activate the BFM or connected to power up reset.
`rst_o`	Output	Specifies the active low reset, which is asserted by the debugger, and is intended to reset the core logic. It can be left unconnected when not needed.
`trstn`	Output	This optional signal, active low, asserted by the debugger is used for bringing the TAP controller to a reset state. Some, especially simulated designs, might need this signal asserted before starting the JTAG communication.
`tclk`	Output	Specifies the JTAG clock driven by the BFM only during an active shift or scan. All JTAG outputs are driven on the negative edge of the `tclk`. `TDO` is sampled on the positive edge of `tclk`.
`tms`	Output	Specifies the Test Mode Select signal driven by BFM to set up the TAP to a particular state depending on the requested operation.
`tdi`	Output	Specifies the Test Data Input. It is the serial data driven on the negative edge of `tclk`.
`tdo`	Input	Specifies the Test Data Output. It is the serial data sampled on the positive edge of `tclk`.

Related Section

Integrating BFM Modules with the Design

APB Integration

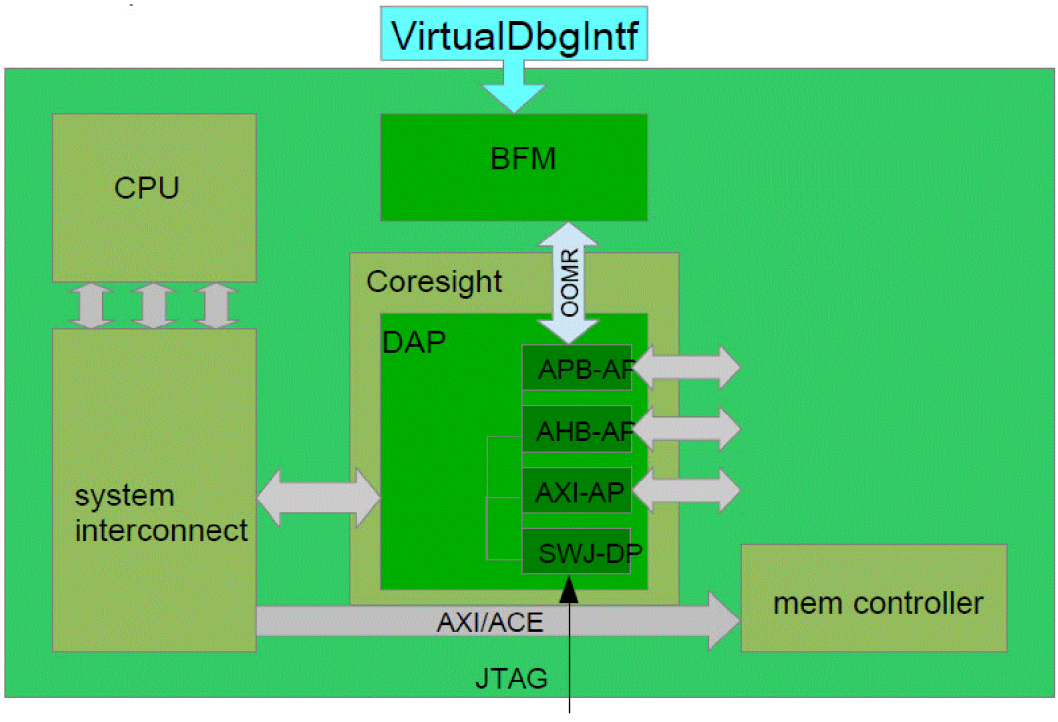
A module named `vd_apb_bfm`, implemented in SystemVerilog, provides a signal-level interface to the design APB bus. The APB signals are not present at the design interface. Therefore, the integration might need to include signal modifications inside the design.

In particular, the BFM integration can be accomplished in two ways:

- Net Cut with Force (APB)
- Instantiation within Coresight (APB)

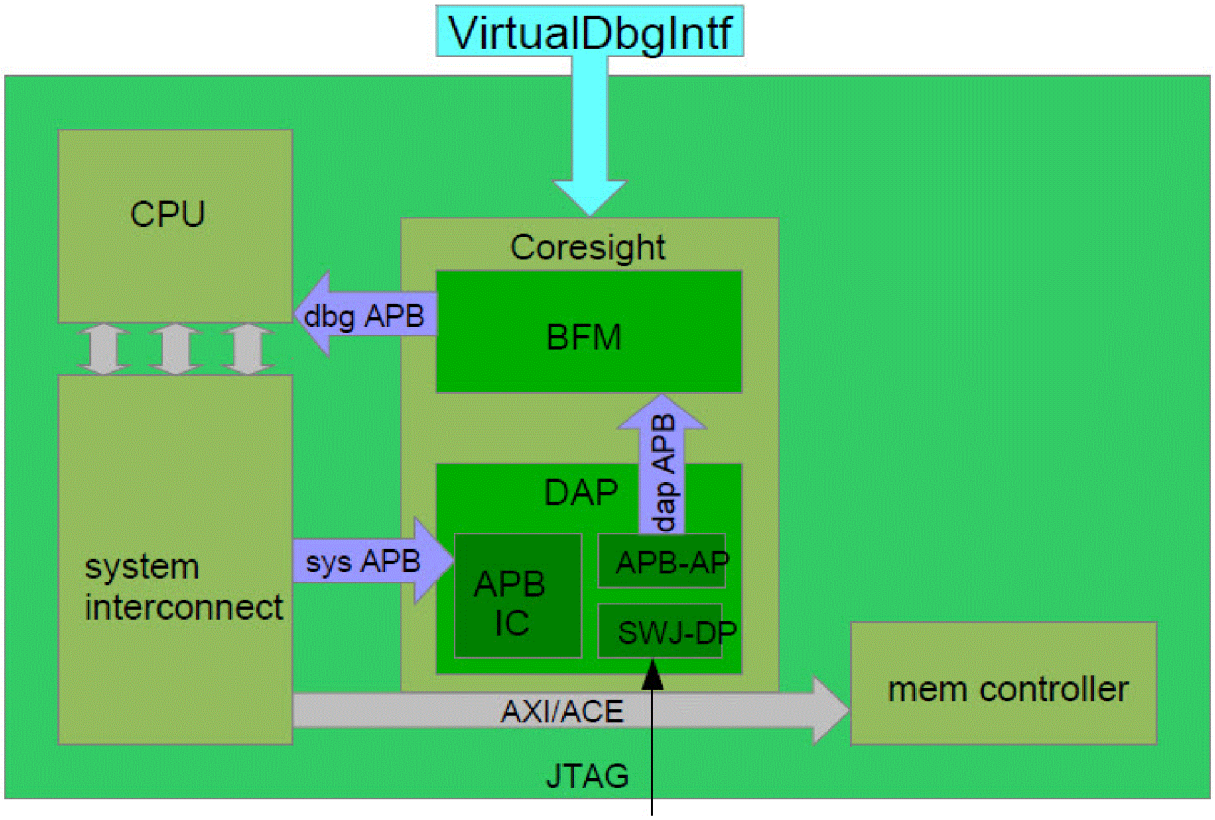
Net Cut with Force (APB)

In this case, no design modifications are necessary. The BFM gets instantiated in the hardware top-level wrapper and connects to the APB bus through the out-of-module references, dynamically cutting the existing connections by the Verilog `force` command. Following is a diagrammatic presentation of this concept.



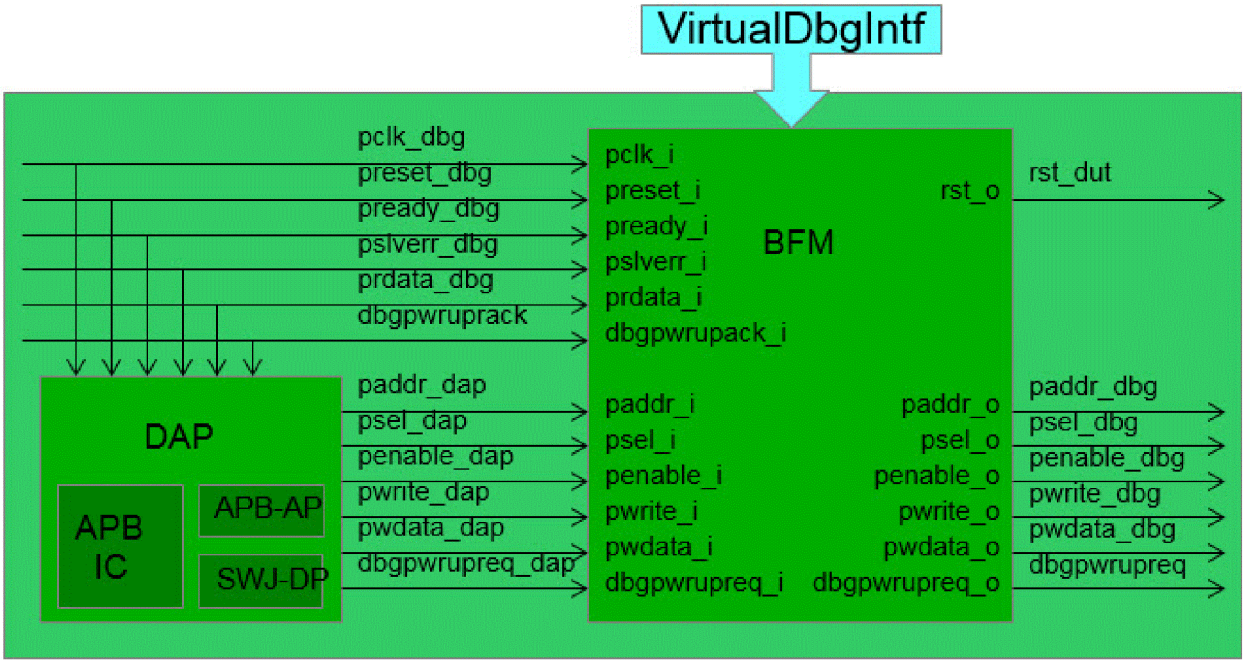
Instantiation within Coresight (APB)

The BFM can be instantiated inside the Coresight subsystem, parallel to the Debug Access Port (DAP) with modified signal interconnect. The BFM can take the DAP signals as inputs, and switch between its own APB master and the DAP. Any activities on the DAP deactivate the APB master. Following is a diagrammatic representation of the concept.



The BFM has an integrated APB multiplexer. All the APB outputs are driven by the debugger, only when the DAP is not active. To determine that, the BFM monitors the `dbgpwrupreq_i` signal from DAP and - when active - switches the internal multiplexer driving the APB outputs from the corresponding inputs. For instance, `paddr_o` is driven from `paddr_i`, not the debugger. This way, the DAP takes control of the APB bus. The control can be reclaimed when `dbgpwrupreq_i` is not active and the debugger issues a new command. If the multiplexer is not used, the `dbgpwrupreq_i` must be tied to `1'b0` and the `dbgpwruack_i` must be tied to `1'b1`.

The following diagram illustrates the signal connections.



The following table describes the BFM signals.

Signal	Direction	Description
`pclk_i`	Input	APB clock; its rising edge times all transfers.
`presetn_i`	Input	Asynchronous reset; active low. This signal can be used to dynamically activate the BFM or connected to power up reset.
`pready_i`	Input	Slave signal to optionally extend an APB transfer
`pslverr_i`	Input	A transfer failure coming from APB slave
`prdata_i[31:0]`	Input	APB data bus from the slave
`paddr_o[31:0]`	Output	APB address bus driven by debugger
`psel_o`	Output	Select signal for the slave
`penable_o`	Output	Indicates second and subsequent cycles of an APB transfer
`pwrite_o`	Output	Indicates the transfer type, write- high, read - low
`pwdata_o[31:0]`	Output	The write data driven by the debugger

`paddr_i[31:0]`	Input	APB address from the DAP
`psel_i`	Input	Select signal from the DAP
`penable_i`	Input	The enable signal from the DAP
`pwrite_i`	Input	The transfer type from the DAP
`pwdata_i[31:0]`	Input	The write data from the DAP
`dbgpwrupreq_i`	Input	The debug power-up request from the DAP. Tie to zero to deactivate the internal multiplexer.
`dbgpwrupreq_o`	Output	The BFM asserts this signal requesting the debug subsystem to power up after the state machine reset.
`dbgpwrupack_i`	Input	Power up acknowledgment coming from the debug subsystem; BFM monitors this signal, which must be asserted to `1`.
`rst_o`	Output	The optional core reset, active low, driven by debugger. Leave unconnected when not used.

Related Section

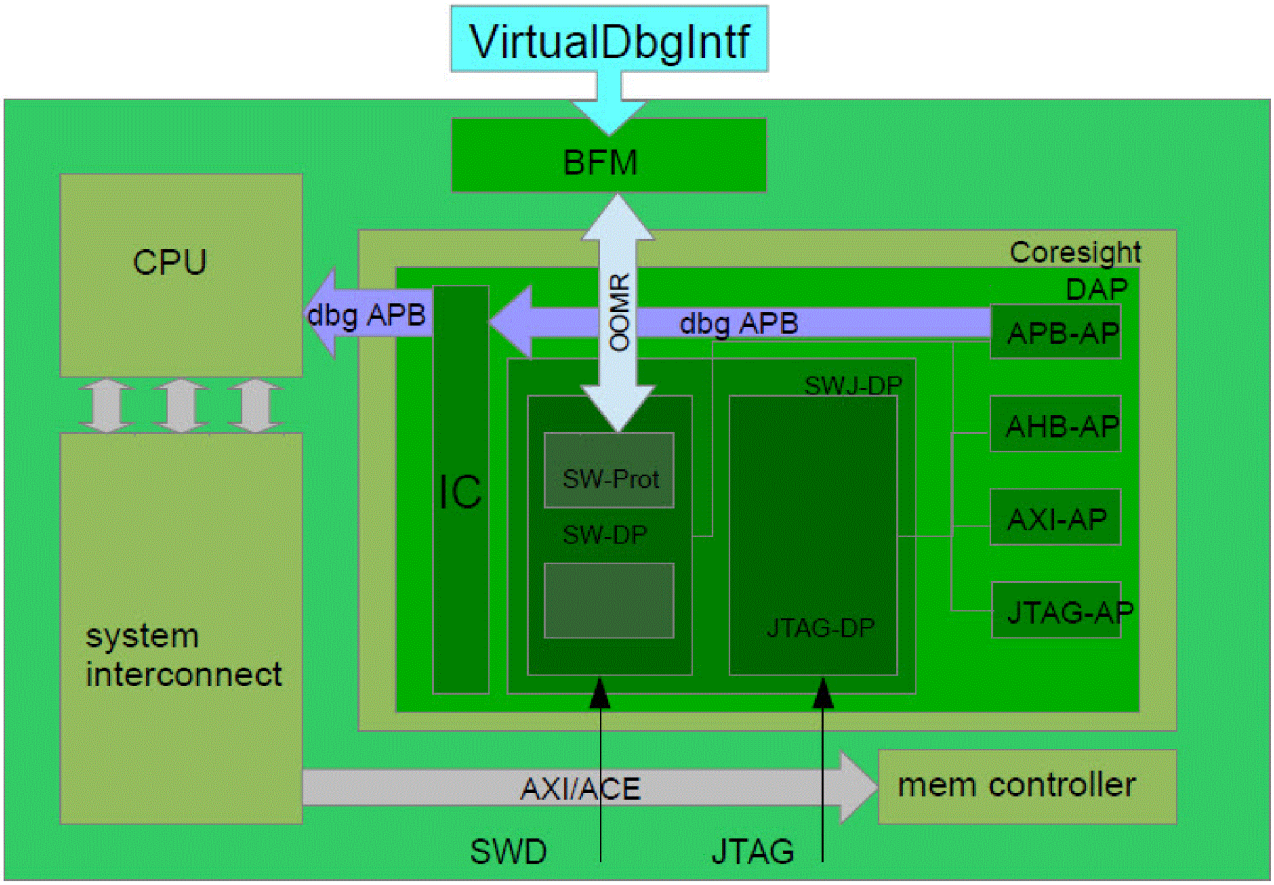
Integrating BFM Modules with the Design

DAP5 Integration

A module named `vd_swdp_bfm`, implemented in SystemVerilog, provides a signal-level interface to the internal Debug Access Port bus including power-control signals. There are different implementations of the DAP.

> Note
> This interface supports only some of these implementations. It works with Arm Debug Interface version 5.

The recommended way to integrate the BFM is to use force and run-time control. This integration is shown below.



The signals needed by the BFM can be found inside the DAP at the `sw_dp_protocol` level in the `swclk` clock domain. In addition to connecting the signals, some control signals also need to be forced, and these vary from one core to another.

Following are a few integration examples:

Cortex M3

```
`define VD_DAP_PATH tbench.uCORTEXM3INTEGRATION.uDAPSWJDP
  wire     vd_busreq;
  wire      vd_buswrite;
  wire     vd_busabort;
  wire [31:0]   vd_busaddr;
  wire [31:0]   vd_buswdata;
  wire [1:0]   vd_busmode;
  wire [3:0]   vd_busmask;
  wire [11:0]   vd_bustrncnt;
  wire    vd_cdbgpwrupreq;
  wire    vd_csyspwrupreq;
  wire     vd_cdbgrstreq;
  wire    vd_dresetn;

vd_swdp_bfm u_vd_swdp_bfm (
    .dbgrstn_i     (HRESETn),
    .dbgclk_i     (HCLK),
    .busack_i     (`VD_DAP_PATH.BusackT),
    .busrdata_i     (`VD_DAP_PATH.BusrdataC[31:0]),
    .buscmp_i     (`VD_DAP_PATH.BuscmpC),
    .buserror_i     (`VD_DAP_PATH.BuserrorC),
    .busendcnt_i    (`VD_DAP_PATH.BusendcntC[11:0]),
    .cdbgpwrupack_i    (`VD_DAP_PATH.CdbgpwrupackS),
    .csyspwrupack_i    (`VD_DAP_PATH.CsyspwrupackS),
    .cdbgrstack_i    (`VD_DAP_PATH.CdbgrstackS),
    .busreq_o     (vd_busreq),
    .buswrite_o     (vd_buswrite),
    .busabort_o     (vd_busabort),
    .busaddr_o     (vd_busaddr),
    .buswdata_o     (vd_buswdata),
    .busmode_o     (vd_busmode),
    .busmask_o     (vd_busmask),
    .bustrncnt_o    (vd_bustrncnt),
    .cdbgpwrupreq_o    (vd_cdbgpwrupreq),
    .csyspwrupreq_o    (vd_csyspwrupreq),
    .cdbgrstreq_o    (vd_cdbgrstreq),
    .rst_o      (vd_dresetn)
  );

always @(vd_swdp_enable)
    if( vd_swdp_enable == 1'b1 )
    begin
      force `VD_DAP_PATH.JTAGNSW    = 1'b0;
      force `VD_DAP_PATH.SWCLKTCK     = HCLK;
      force `VD_DAP_PATH.SBusreq     = vd_busreq;
      force `VD_DAP_PATH.SBuswrite     = vd_buswrite;
      force `VD_DAP_PATH.SBusabort     = vd_busabort;
      force `VD_DAP_PATH.SBusaddr[31:0]   = vd_busaddr;
      force `VD_DAP_PATH.SBuswdata[31:0]   = vd_buswdata;
```

```
    force `VD_DAP_PATH.SBusmode[1:0]    = vd_busmode;
    force `VD_DAP_PATH.SBusmask[3:0]   = vd_busmask;
    force `VD_DAP_PATH.SBustrncnt[11:0]  = vd_bustrncnt;
    force `VD_DAP_PATH.SCDbgPwrUpReq    = vd_cdbgpwrupreq;
    force `VD_DAP_PATH.SCSysPwrUpReq    = vd_csyspwrupreq;
    force `VD_DAP_PATH.SCDbgRstReq    = vd_cdbgrstreq;
  end
  else
  begin
    release `VD_DAP_PATH.JTAGNSW;
    release `VD_DAP_PATH.SWCLKTCK;
    release `VD_DAP_PATH.SBusreq;
    release `VD_DAP_PATH.SBuswrite;
    release `VD_DAP_PATH.SBusabort;
    release `VD_DAP_PATH.SBusaddr;
    release `VD_DAP_PATH.SBuswdata;
    release `VD_DAP_PATH.SBusmode;
    release `VD_DAP_PATH.SBusmask;
    release `VD_DAP_PATH.SBustrncnt;
    release `VD_DAP_PATH.SCDbgPwrUpReq;
    release `VD_DAP_PATH.SCSysPwrUpReq;
    release `VD_DAP_PATH.SCDbgRstReq;
  end
```

Cortex A53

```
`define VD_DAP_PATH tbench.u_cssys.u_dap_upv8.cxdapswjdp_0.u_cxdapswjdp_swclktck
  wire        vd_busreq;
  wire        vd_buswrite;
  wire          vd_busabort;
  wire [31:0]   vd_busaddr;
  wire [31:0]  vd_buswdata;
  wire [1:0]    vd_busmode;
  wire [3:0]   vd_busmask;
  wire [11:0]   vd_bustrncnt;
  wire         vd_cdbgpwrupreq;
  wire         vd_csyspwrupreq;
  wire         vd_cdbgrstreq;
  wire         vd_dresetn;

  vd_swdp_bfm u_vd_swdp_bfm (
    .dbgrstn_i     (reset_n),
    .dbgclk_i     (clk),
    .busack_i     (`VD_DAP_PATH.busack_t),
    .busrdata_i     (`VD_DAP_PATH.busrdata_c[31:0]),
    .buscmp_i     (`VD_DAP_PATH.buscmp_c),
    .buserror_i     (`VD_DAP_PATH.buserr_c),
    .busendcnt_i    (`VD_DAP_PATH.busendcnt_c[11:0]),
    .cdbgpwrupack_i    (`VD_DAP_PATH.cdbgpwrupack_s),
    .csyspwrupack_i    (`VD_DAP_PATH.csyspwrupack_s),
    .cdbgrstack_i    (`VD_DAP_PATH.cdbgrstack_s),
    .busreq_o     (vd_busreq),
    .buswrite_o     (vd_buswrite),
    .busabort_o     (vd_busabort),
    .busaddr_o     (vd_busaddr),
    .buswdata_o     (vd_buswdata),
    .busmode_o     (vd_busmode),
    .busmask_o     (vd_busmask),
    .bustrncnt_o    (vd_bustrncnt),
    .cdbgpwrupreq_o    (vd_cdbgpwrupreq),
    .csyspwrupreq_o    (vd_csyspwrupreq),
    .cdbgrstreq_o    (vd_cdbgrstreq),
    .rst_o      (vd_dresetn)
  );

  always @(vd_swdp_enable)
    if( vd_swdp_enable == 1'b1 )
    begin
      force `VD_DAP_PATH.jtag_sel_int   = 1'b0;
      force `VD_DAP_PATH.swdsel     = 1'b1;
      force `VD_DAP_PATH.swclktck    = clk;
      force `VD_DAP_PATH.s_busreq     = vd_busreq;
      force `VD_DAP_PATH.s_buswrite    = vd_buswrite;
      force `VD_DAP_PATH.s_busabort    = vd_busabort;
      force `VD_DAP_PATH.sw_busaddr_31to24  = vd_busaddr[31:24];
      force `VD_DAP_PATH.sw_busaddr_7to2  = vd_busaddr[7:2];
      force `VD_DAP_PATH.s_buswdata[31:0]  = vd_buswdata;
```

```
    force `VD_DAP_PATH.s_busmode[1:0]   = vd_busmode;
    force `VD_DAP_PATH.s_busmask[3:0]   = vd_busmask;
    force `VD_DAP_PATH.s_bustrncnt[11:0]  = vd_bustrncnt;
    force `VD_DAP_PATH.s_cdbgpwrupreq   = vd_cdbgpwrupreq;
    force `VD_DAP_PATH.s_csyspwrupreq   = vd_csyspwrupreq;
    force `VD_DAP_PATH.s_cdbgrstreq    = vd_cdbgrstreq;
  end
  else
  begin
    release `VD_DAP_PATH.jtag_sel_int;
    release `VD_DAP_PATH.swdsel;
    release `VD_DAP_PATH.swclktck;
    release `VD_DAP_PATH.s_busreq;
    release `VD_DAP_PATH.s_buswrite;
    release `VD_DAP_PATH.s_busabort;
    release `VD_DAP_PATH.sw_busaddr_31to24;
    release `VD_DAP_PATH.sw_busaddr_7to2;
    release `VD_DAP_PATH.s_buswdata;
    release `VD_DAP_PATH.s_busmode;
    release `VD_DAP_PATH.s_busmask;
    release `VD_DAP_PATH.s_bustrncnt;
    release `VD_DAP_PATH.s_cdbgpwrupreq;
    release `VD_DAP_PATH.s_csyspwrupreq;
    release `VD_DAP_PATH.s_cdbgrstreq;
  end
```

The following table describes the BFM signals.

Signal	Direction	Description
dbgclk_i	Input	DAP5 clock; its rising edge times all transfers.
dbgrstn_i	Input	Asynchronous reset is set to active low. This signal can be used to dynamically activate the BFM or connected to power up reset.
busack_i	Input	DAP bus ack
busrdata_i[31;0]	Input	DAP bus read data
buscmp_i	Input	DAP bus push transaction
buserror_i	Input	DAP bus error
busendcnt_i	Input	DAP bus end counter
cdbgpwrupack_i	Input	Debug Power Up acknowledge, which is checked by debugger. It must be asserted for the interface to work.

csyspwrupack_i	Input	System Power Up acknowledge, which is checked by debugger.
cdbgrstack_i	Input	Debug Reset acknowledge, which is ignored.
busreq_o	Output	DAP bus request
buswrite_o	Output	DAP bus write
busabort_o	Output	DAP bus abort
busaddr_o[31:0]	Output	DAP bus address
buswdata_o[31:0]	Output	DAP bus write data
busmode_o[1:0]	Output	DAP bus mode
busmask_o[3:0]	Ouput	DAP bus mask.
bustrncnt_o[11:0]	Output	DAP bus transaction counter.
cdbgpwrupreq_o	Output	Debug power-up request, which is asserted by the debugger.
csyspwrupreq_o	Output	System power-up request, which is asserted by the debugger.
cdbgrstreq_o	Output	Debug reset request, not used, always 0.
rst_o	Output	The optional core reset, active low, driven by debugger. Leave unconnected when not used.

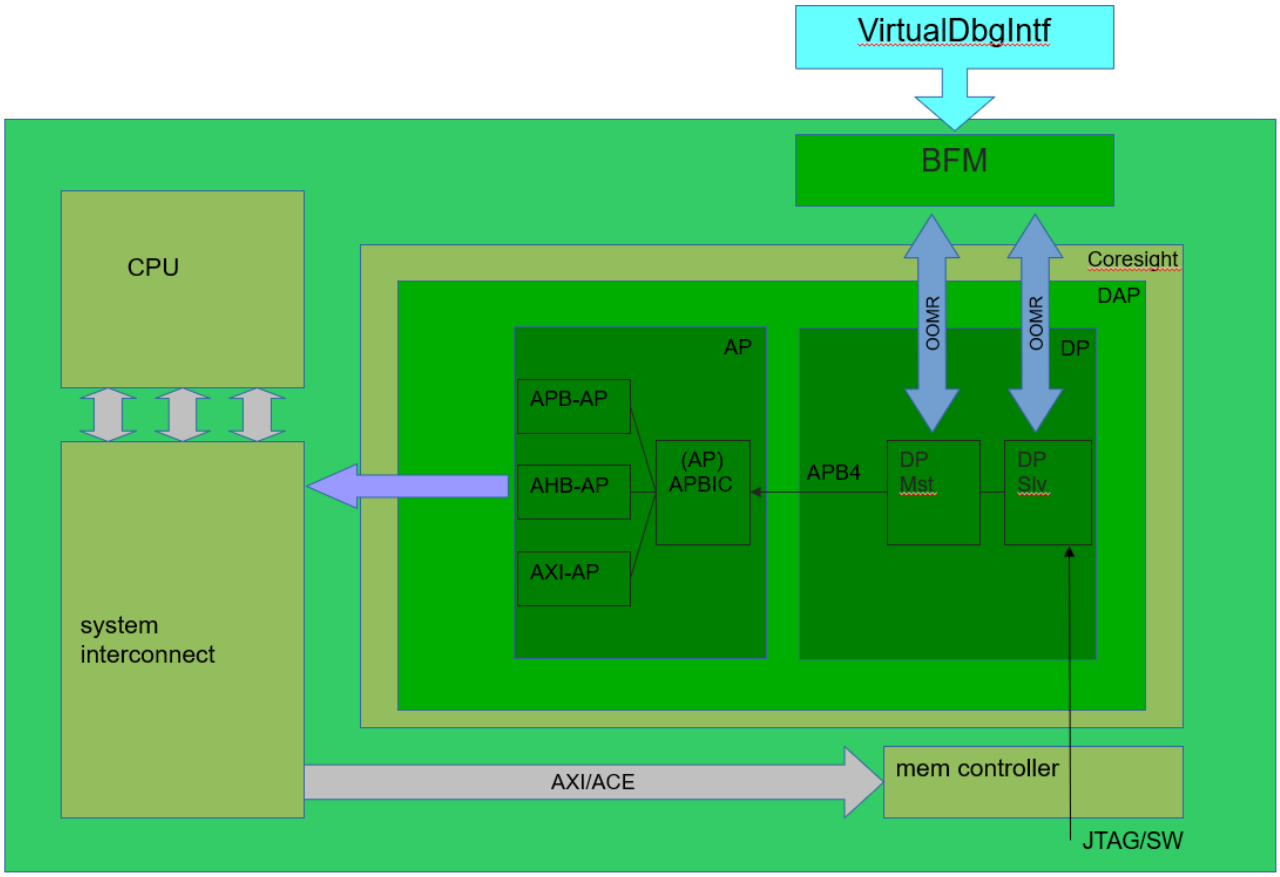
Related Section

Integrating BFM Modules with the Design

DAP6 Integration

A module named `vd_dap6_bfm`, implemented in SystemVerilog, provides a signal-level interface to the internal Debug Access Port bus including power-control signals. It supports the newest Arm Debug Interface version 6.

The recommended way to integrate the BFM is to use force and run-time control. This integration is shown below.



The signals needed by the BFM can be found inside the DP Master module and the DP Slave module of Arm Coresight SOC-600.

The input signals of DAP6 BFM contain:

- Power-control signals and DP register signals from the DP Slave module
- DP internal bus signals from the DP Master module

The output signals of DAP6 BFM contain:

- Power-control signals for the DP Slave module
- APB4 protocol signals to replace the DP Master module

Following is an integration example:

Example System of Arm Coresight SOC-600

```
`define VD_DAP6_DPSLV_PATH ess600_tb.u_dut.u_esa_aon
`define VD_DAP6_DPMST_PATH ess600_tb.u_dut.u_esa_aon.u_esa_dbgaonclkg
  wire              vd_enable;
  wire   [31: 0]    vd_addr;
  wire              vd_write;
  wire   [31: 0]    vd_wdata;
  wire              vd_sel;
  wire              vd_cdbgpwrupreq;
  wire              vd_csyspwrupreq;
  wire              vd_cdbgrstreq;
  wire              vd_dp_abort;
  reg               vd_dap6_enable = 1'b0;

  vd_dap6_bfm u_vd_dap6_bfm(
    .pclk_i                     (`VD_DAP6_DPMST_PATH.dbgaonclkg),
    .presetn_i                  (`VD_DAP6_DPMST_PATH.dbgaonclkg_dbgaon_resetn),
    .pready_i                   (`VD_DAP6_DPMST_PATH.pready_s0),
    .pslverr_i                  (`VD_DAP6_DPMST_PATH.pslverr_s0),
    .prdata_i                   (`VD_DAP6_DPMST_PATH.prdata_s0),
    .dp_eventstatus_i           (`VD_DAP6_DPSLV_PATH.ctitrigout14),
    .cdbgpwrupack_i             (`VD_DAP6_DPSLV_PATH.dp_cdbgpwrupreq),
    .csyspwrupack_i             (`VD_DAP6_DPSLV_PATH.dp_csyspwrupreq),
    .cdbgrstack_i               (`VD_DAP6_DPSLV_PATH.dp_cdbgrstack),
    .baseaddr_i                 (32'h00000000),
    .baseaddr_valid_i           (1'b1),
    .targetid_i                 (32'h07DA0477),
    .instanceid_i               (4'h0),

    .paddr_o                    (vd_addr[31:0]),
    .psel_o                     (vd_sel),
    .penable_o                  (vd_enable),
    .pwrite_o                   (vd_write),
    .pwdata_o                   (vd_wdata[31:0]),
    .rst_o                      (),
    .cdbgpwrupreq_o             (vd_cdbgpwrupreq),
    .csyspwrupreq_o             (vd_csyspwrupreq),
    .cdbgrstreq_o               (vd_cdbgrstreq),
    .dp_abort_o                 (vd_dp_abort)
  );
  always @(vd_dap6_enable) begin
    if( vd_dap6_enable == 1'b1)
    begin
      force `VD_DAP6_DPMST_PATH.psel_s0         = vd_sel;
      force `VD_DAP6_DPMST_PATH.penable_s0      = vd_enable;
      force `VD_DAP6_DPMST_PATH.pwrite_s0       = vd_write;
      force `VD_DAP6_DPMST_PATH.pwdata_s0[31:0] = vd_wdata;
      force `VD_DAP6_DPMST_PATH.paddr_s0        = vd_addr;
      force `VD_DAP6_DPMST_PATH.dpabort         = vd_dp_abort;
```

```
      force `VD_DAP6_DPSLV_PATH.dp_cdbgpwrupreq = vd_cdbgpwrupreq;
      force `VD_DAP6_DPSLV_PATH.dp_csyspwrupreq = vd_csyspwrupreq;
      force `VD_DAP6_DPSLV_PATH.dp_cdbgrstack   = vd_cdbgrstreq;
    end
    else
    begin
      release `VD_DAP6_DPMST_PATH.psel_s0;
      release `VD_DAP6_DPMST_PATH.penable_s0;
      release `VD_DAP6_DPMST_PATH.pwrite_s0;
      release `VD_DAP6_DPMST_PATH.pwdata_s0;
      release `VD_DAP6_DPMST_PATH.paddr_s0;
      release `VD_DAP6_DPMST_PATH.dpabort;

      release `VD_DAP6_DPSLV_PATH.dp_cdbgpwrupreq;
      release `VD_DAP6_DPSLV_PATH.dp_csyspwrupreq;
      release `VD_DAP6_DPSLV_PATH.dp_cdbgrstack;
    end
  end
```

In the above example, the power and debug "`ack`" input signals are connected to the power and debug "`req`" output signals. This is because the example system from Arm has always-on domain, so the "`req`" and "`ack`" signals are tied together.

The following table describes the BFM signals.

Signal	Direction	Description
`pclk_i`	Input	DAP6 clock; its rising edge times all transfers.
`presetn_i`	Input	Asynchronous reset, active low.
`pready_i`	Input	Ready signal used to extend an APB transfer.
`pslverr_i`	Input	Slave error indicating a transfer failure.
`prdata_i`	Input	APB data bus from the slave
`dp_eventstatus_i`	Input	Top-level input coming from Cross-Trigger Interface(CTI), used to signal an event to the DP
`cdbgpwrupack_i`	Input	Debug power domain-up acknowledgement
`csyspwrupack_i`	Input	System power domain-up acknowledgement

cdbgrstack_i	Input	Debug reset acknowledgement to reset controller
baseaddr_i	Input	DP debug ROM base address
baseaddrvalid_i	Input	1’b1 indicates base address valid
targetid_i	Input	Specifies the target when the host is connected to a single device. This value is used in the DP TARGETID register field.
instanceid_i	Input	4’h0, Instance ID for SW multidrop selection
paddr_o	Output	APB address bus driven by the debugger
psel_o	Output	Select signal for APB slave. Indicates that the slave device is selected and that a data transfer is required.
penable_o	Output	Indicates second and subsequent cycles of an APB transfer
pwrite_o	Output	Indicates the transfer type: • write-high • read-low
rst_o	Output	DUT reset active low asserted by debugger
cdbgpwrupreq_o	Output	Debug power-up request, asserted by the debugger
csyspwrupreq_o	Output	System power-up request, asserted by the debugger
cdbgrstreq_o	Output	Debug reset request, not used, always 0
dp_abort_o	Output	Active high abort signal to abort a transaction

Related Section

Integrating BFM Modules with the Design

AHB Integration

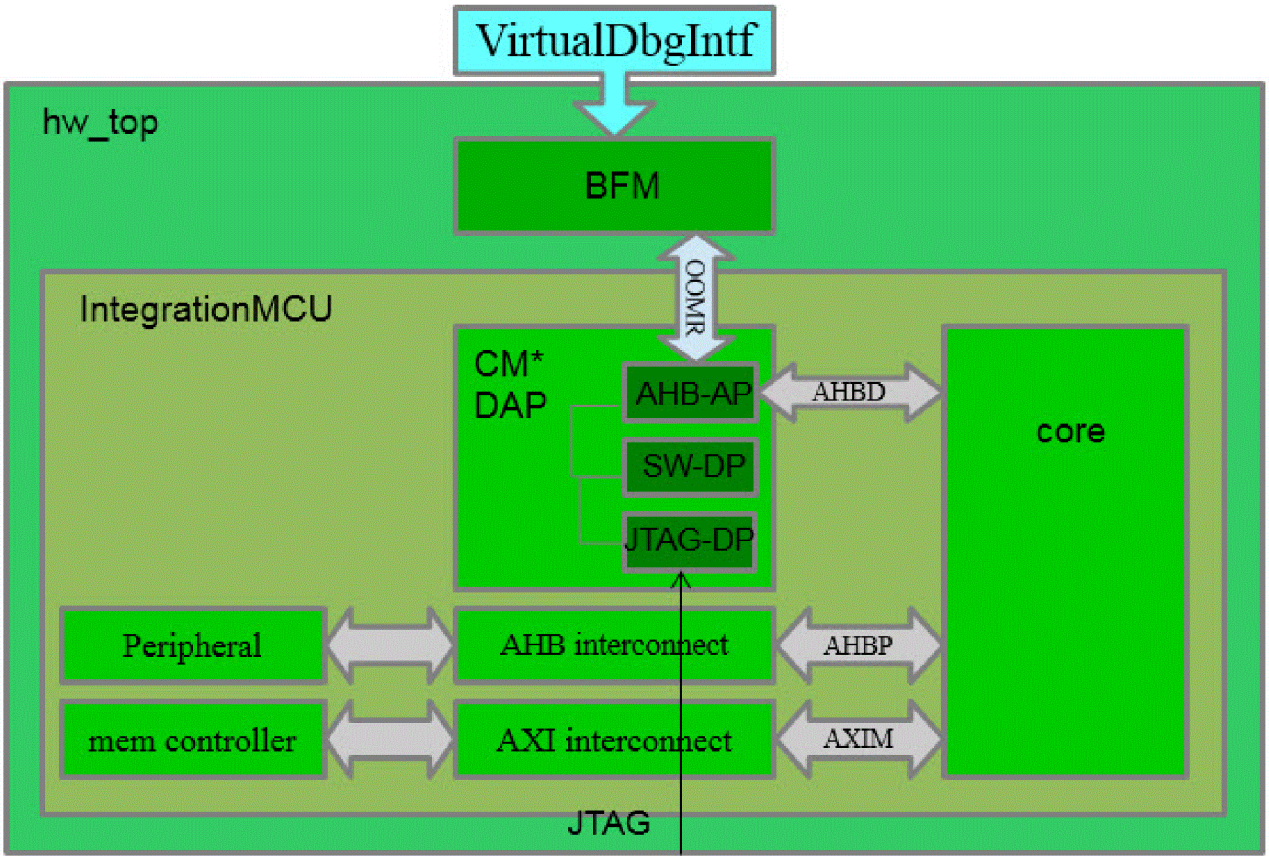
In some architectures, such as ARM Cortex M, the AMBA AHB bus connects the core with the peripherals. The Virtual Debug interface provides a suitable infrastructure to support it. A module named `vd_ahb_bfm`, implemented in SystemVerilog, provides a signal-level interface to the design AHB Lite bus. The AHB signals are not present at the design interface. Therefore, the integration might need to include signal modifications inside the design.

In particular, the BFM integration can be accomplished in two ways:

- Net cut with force (AHB)
- Instantiation within Coresight (AHB)

Net cut with force (AHB)

This is the recommended way in which no design modifications are necessary. The BFM gets instantiated in the hardware top-level wrapper and connects to the AHB bus through the out-of-module references, dynamically cutting the existing connections by the Verilog `force` command. Following is a diagrammatic presentation of this concept.



Instantiation within Coresight (AHB)

The BFM can be instantiated inside the debug subsystem, on the level where the AHB-AP is instantiated. The signal interconnect needs to be modified to take the control from the BFM instead of the AHB-AP. This modification deactivates the AHB-AP master in the design.

The table describes the BFM signals.

Signal	Direction	Description
`hclk_i`	Input	AHB clock; its rising edge times all transfers
`hresetn_i`	Input	Asynchronous reset; active low. This signal can be used to dynamically activate the BFM or connected to power-up reset.
`hready_i`	Input	Slave signal to optionally extend the data transfer phase

`hresp_i`	Input	Slave response: • Success 1'b0 • Error 1'b1
`hrdata_i[31:0]`	Input	Data bus from the slave
`haddr_o[31:0]`	Output	Address bus driven by debugger
`htrans_o[1:0]`	Output	Transfer type: • Idle • Busy • Non-sequential • Sequential
`hwrite_o`	Output	Indicates the transfer type: write- high or read - low
`hsize_o[2:0]`	Output	Transfer size, byte, halfword, word
`hprot_o[3:0]`	Output	Bus access protection
`hwdata_o[31:0]`	Output	Data bus from master to slave
`hburst_o[2:0]`	Output	Transaction burst mode, always 3'b000 single burst
`rst_o`	Output	The optional core reset, active low driven by debugger. Leave unconnected when not used.

Related Section

Integrating BFM Modules with the Design

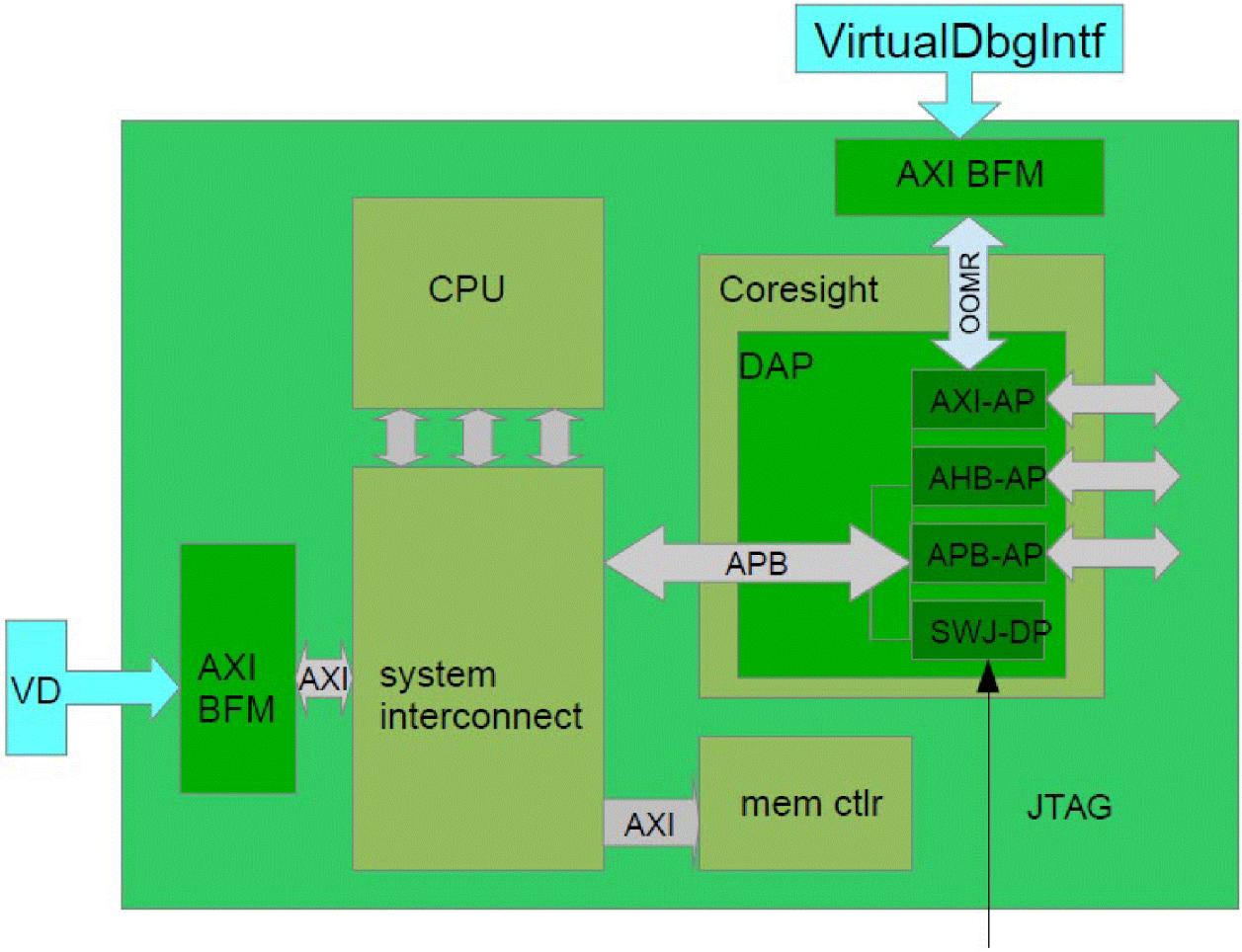
AXI Integration

In some rare designs, the AMBA AXI bus is used to connect the peripherals with the core. The Virtual Debug interface provides a suitable infrastructure for such a case. A module named `vd_axi_bfm`, implemented in SystemVerilog, provides a signal-level interface to the design using a subset of the AXI 4 standard. The AXI signals are not present at the design interface. Therefore, the integration might need to include signal modifications inside the design.

In particular, the BFM integration can be accomplished using Net Cut with Force method.

Net Cut with Force

This is the recommended way in which no design modifications are necessary. The BFM gets instantiated in the hardware top-level wrapper and connects to the AXI bus through the out-of-module references, dynamically cutting the existing connections by the Verilog `force` command. Following is a diagrammatic presentation of this concept.



The following table describes the BFM signals.

Signal	Direction	Description
`aclk_i`	Input	AXI clock; its rising edge times all transfers
`aresetn_i`	Input	Asynchronous reset; active low. This signal can be used to dynamically activate the BFM or connected to power-up reset.
`awready_i`	Input	Address write ready phase completed from slave
`awaddr_o[aw-1:0]`	Input	Write address

awburst_o[1:0]	Output	Write burst type, always 01 – INCR
awlen_o[7:0]	Output	Write burst length
awprot_o[2:0]	Output	Write protection
awsize_o[2:0]	Output	Write data bus width in bytes, 2^size
awvalid_o	Output	Write address valid
arready_i	Input	Read address ready phase completed from slave
araddr_o[aw-1:0]	Output	Read address
arburst_o[1:0]	Output	Read burst type, always 01 - INCR
arlen_o[7:0]	Output	Read burst length
arprot_o[2:0]	Output	Read protection
arsize_o[2:0]	Output	Read data bus width in bytes, 2^size
arvalid_o	Output	Read address valid
wready_i	Input	Write ready phase completed from slave
wdata_o[dw-1:0]	Output	Write data
wvalid_o	Output	Indicates that write data is valid
wstrb_o[dw/8-1:0]	Output	Write data strobe per byte
wlast_o	Output	Last data for thr write
rdata_i[dw-1:0]	Input	Read data
rvalid_i	Input	Read data valid
rlast_i	Input	Last read data
rready_o	Output	Specifies that read channel is ready

`bresp_I[1:0]`	Input	Write response
`bvalid_i`	Input	Write response valid
`bready_o`	Output	Write response ready
`rresp_i[1:0]`	Input	Read response
`rvalid_i`	Input	Read response valid
`rready_i`	Output	Read response ready
`rst_o`	Output	The optional core reset, active low driven by debugger. Leave unconnected when not used.

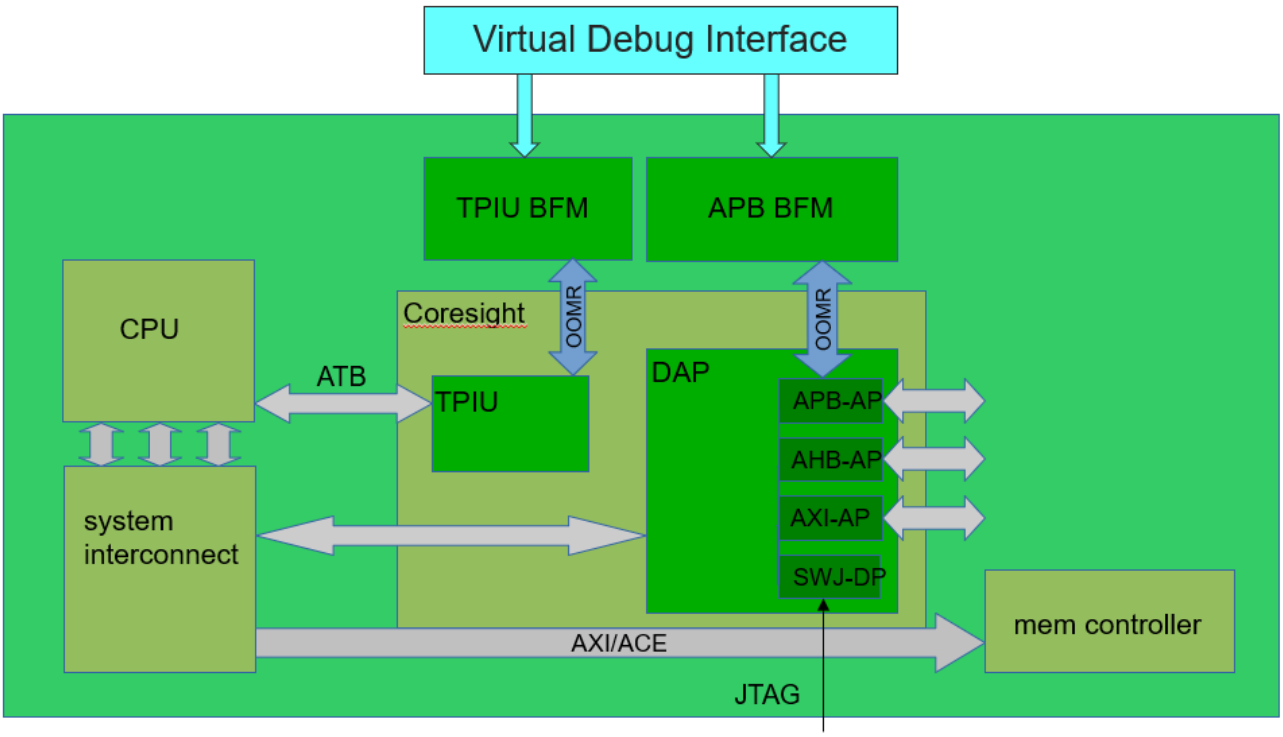
Related Section

Integrating BFM Modules with the Design

TPIU Integration

The TPIU BFM is used for Trace Port Interface Unit. This component acts as a bridge between on-chip trace data to a data stream that could be used by analyzer or debugger. A module named `vd_tpiu_bfm`, implemented in SystemVerilog, provides a signal-level interface to the Trace Port Interface Unit (TPIU) inside the design.

The TPIU BFM should be used simultaneously with other debug interfaces, such as, JTAG, APB, and so on. Following is a diagrammatic presentation of APB and TPIU.



The following table describes the signals of this module.

Signal	Direction (From BFM)	Description
`tclkin_i`	Input	Trace clock in, 2X `traceclk`
`trstn_i`	Input	Trace reset, active low
`tdata_i [data_width-1:0]`	Input	Trace data

`tctl_i`	Input	Trace control bit. Use this signal in Trace mode.
`tmode_i`	Input	Trace mode: • 1= Normal; Should be used with `tctl_i` • 0= Continuous; Should be used without `tctl_i`

Example of TPIU Integration

Cortex M4

```
`ifdef VD_DEBUG_TPIU
`define VD_TPIU_PATH tbench.u_cm4_tpiu
// ----------------------------------------------------------------
// TPIU BFM instance
// ----------------------------------------------------------------
vd_tpiu_bfm #(.data_width(4)) u_vd_tpiu_bfm (
// Inputs
.tclkin_i (`VD_TPIU_PATH.TRACECLKIN),
.trstn_i (`VD_TPIU_PATH.TRESETn),
.tdata_i (`VD_TPIU_PATH.TRACEDATA),
.tctl_i (1'b0),
.tmode_i (1'b0));
`undef VD_TPIU_PATH
`endif //VD_DEBUG_TPIU
```

6

Compiling the Design

Following are the commands required for Palladium Z1 and Palladium Z2:

Compile the BFM with the rest of the DUT using the `vlan` and `ixcom` commands.

```
vlan $BITS -sv v/vd_*_bfm.svp
ixcom -$BITS <design_sources> [ixcom_options] -ua +tb_import_systf+
[vd_*_bfm.]stop
```

The `tb_import_systf` option activates the system task `$stop` in the optionally provided module (like v`d_jtag_bfm`), allowing emulation to stop on software errors.

Following are the commands required for Protium X1 and Protium X2:

Compile the BFM with the rest of the DUT using the `vlan` and `ixcom` commands.

```
vlan $BITS -sv v/vd_*_bfm.svp
ixcom -$BITS <design_sources> [ixcom_options] -ptm +tb_import_systf+
[vd_*_bfm.]stop
```

The `tb_import_systf` option activates the system task `$stop` in the optionally provided module (like `vd_jtag_bfm`), allowing emulation to stop on software errors.

Following are the commands required for Xcelium:

Compile the BFM with the rest of the DUT using your preferred flow. The BFMs need the read and write access. These need to be elaborated with the `-access rw` option.

```
xmvlog $BITS -sv v/vd_*_bfm.svp
xmelab -$BITS [your_design_flags] [xlm_options] -access rw
```

7

Running and Bringing-Up the Virtual Debug Interface

This topic describes the environment variables used to run the Virtual Debug Interface and levels of debugging possible with the Virtual Debug Interface.

Controlling Debugger Communication with Simulator and Emulator

The environment variable can be used to control the ways the debugger process communicates with the simulator and emulator, either through a shared memory or TCP sockets. By default, it uses shared memory called `/vdipc` and uses TCP port 8192.

Setting the `VD_SESSION` environment variable is required if multiple debugger or simulator sessions are running on one workstation. The `VD_SESSION` variable is accepted by both the client and the server, but it does not need to be identical for both.

- For the client, the `VD_SESSION` value overrides the configuration parameters. It either specifies shared memory in the form of `/name` or specifies the server host and port in the form of <*host*>:<*port*>. When the **:** (colon) is present and either host or port are empty, these are passed from the debugger configuration parameters.
- For the server, the first portion of the value is interpreted as the shared memory name, if it starts with the "/". Otherwise the first portion is ignored. The second part specifies the TCP

port number. The `VD_SESSION` can be used to disable either shared memory or TCP by specifying empty strings for the appropriate portion.

Examples:

The vdebug server uses default values: shared memory `/vdipc` and TCP port 8192:

```
> unset VD_SESSION (not set)
```

The vdebug server listens on shared memory named `/myipc` and the default TCP port 8192:

```
> export VD_SESSION=/myipc
```

The vdebug server listens on shared memory named `/myipc`, does not listen on TCP port:

```
> export VD_SESSION=/myipc:
```

Following command instructs the server to create a shared memory named `/localhost` and listen on TCP 8193 port:

```
> export VD_SESSION=/localhost:8193
```

The vdebug server listens only on TCP 8193 port, and does not create shared memory:

```
> export VD_SESSION=:8193
```

Bringing-Up the vdebug Server

In one terminal, start the simulator or emulator session, either with xrun or xeDebug, specifying the vdebug server library through the `-sv_lib` option.

For the emulated cores, load `libvdserver.so` and swap into hardware specifying tbrun mode in the `xc_run.tcl`:

```
> xrun -R -sv_lib <VD_PATH>lib/64bit/libvdserver.so -input xc_run.tcl -exit
```

For the simulated cores, load `libvdserver.nc.so`:

```
> xrun -R -sv_lib <VD_PATH>lib/64bit/libvdserver.nc.so -exit
```

When one of the BFMs gets activated or gets out of reset, vdserver prints a banner on the terminal and waits for a connection from the debugger:

```
0ns: vd_server 48 pdapp 23.03 bfm JTAG start; polling 1000 shared mem /localhost
listening on TCP port 8192
```

Once started in a terminal, the server can be stopped by Ctrl-C. Pressing Ctrl-C multiple times within one second can be used to control the server. If a debugger is connected, multiple Ctrl-C will force to disconnect the client. If there is no client connected, multiple Ctrl-C will shut down the server.

Note
The vdebug server works the best in the `tbrun` mode. It can also work in the `nbrun` mode. However, the performance depends on the interval and other activities between the testbench and the design.

The `run` mode is functional, but usually too slow. The vdebug server does not work in the `autorun` mode.

If the debugger has been started before the simulator or emulator session, it might appear hung and would need to be restarted.

If the previous session has not been shut down properly, it might leave the shared memory file in the system. It should not impact the run, but it may affect your run scripts. Delete it with the Unix command:

```
> rm /dev/shm/<session>
```

Changing Frequency of Debugger Connection Checks

The virtual debug interface operates in a polling mode when not connected and blocking when connected. Previous versions could be configured to always work in blocking mode, but the introduction of the TCP connection made it impractical and this setting has been removed.

- The Virtual Debug server starts in polling mode, periodically checking for debugger connections while the simulator (or emulator) session runs. The frequency of the checks can be controlled by setting the `VD_POLL` environment variable:

  ```
  > export/setenv VD_POLL=checks_per_second
  ```

  Recommended values are in the range of 100 to 1 (default). This value is adjusted dynamically, depending on the emulation/simulation performance.
  Once the debugger connects, the software switches to the blocking mode, in which the run is

fully controlled by the debugger. This continues until the debugger disconnects, and the transactor resumes operation in the polling mode and the simulator (or emulator) continues to run. The debugger can be connected and disconnected multiple times during the same Palladium session.

- The virtual bus interface switches to blocking mode when connected. The simulator (or emulator) session runs as long as needed to execute debugger commands including pauses. Closing the debugger reverts the server to the polling mode.

After the vdebug server is up and running it is time to connect the debugger.

Enabling Generation of Debug Information from Client

The first step is to enable generation of debug information from the client library by setting the environment variable `VD_DEBUG` to `1` in the debugger terminal. Higher values produce more diagnostic messages. This environment variable prints additional debug messages in the log file. Adding `0x100` to the value additionally echoes the messages to the console. The same environment variable can be set in the simulator (or emulator) terminal.

Changing Name and Location of Debug Logs

When `VD_DEBUG` is set, the `vd_client.log` is created in the current run directory. The name and the location of the debug log can be changed by setting the `VD_LOG` environment variable.

The BFM can be compiled with the optional `-define DEBUG` option, which when combined with the `+gfifoDisp+`<*`bfm_module_name`*> option of the `ixcom` command, enables the BFM to generate the debug messages in the log file. This requires the DUT to be recompiled and introduces additional synchronizations, thereby affecting the run-time performance. By matching the corresponding client and BFM entries, the communication between the debugger and BFM can be debugged.

Controlling the JTAG Frequency

The frequency of JTAG can be controlled by overriding the internal BFM divider value using the `VD_DIVIDER` environment variable. The syntax is:

```
export VD_DIVIDER=value
```

If `VD_DIVIDER` is set, the specified value is used for the BFM internal divider instead of the calculated value from the BFM clock and `jtagclock` setting.

For information about the formulas, refer to *Customizing the Entries in the Scripts* section in Configuring and Running Lauterbach Trace 32.

8

Configuring and Running the Debuggers

Following sections describe the processes of configuring, running, and debugging the debuggers.

Configuring and Running Lauterbach Trace 32

This topic describes the processes of configuring, running, and debugging the Lauterbach Trace32 debugger.

Configuring Lauterbach Trace32

The vdebug release includes the Lauterbach Trace32 client library, `libvdgtl`. The library matching your platform may need to be copied to directory where you run your debugger.

There are two ways to configure the debugger. Trace32 can load the client library either directly or through the proxy called `t32mciserver`, provided by Lauterbach.

> The use of `t32mciserver` is necessary to debug multiple cores at the same time from separate Trace32 windows. This configuration is recommended when Trace32 needs to run on a workstation on a different network than the Palladium host, which is the case, typically, when Trace32 runs on Windows. Such configuration provides better performance by reducing the number of low-level packets sent across slower inter-net connections. It is recommended to run `t32mciserver` on the Palladium host and use shared memory to communicate between the client (loaded by `t32mciserver`) and vdebug server (loaded by Xcelium). The communication between Trace32 and `t32mciserver` is over TCP sockets.

- The `config.t32` configuration file must contain the following line to enable the direct loading of the client library by the Trace32:

```
PBI=MCILIB
```

- For the connection through the `t32mciserver`, the PBI line must read:

```
PBI=MCISERVER
PORT=5001            ; port on which the t32mciserver listens
NODE=cva-demo4 ; host on which t32mciserver is running
```

To enable multiple debugger instances connecting to the same multi-core module, additional settings are required. Refer to the Creating Simultaneous Connections to Multiple Cores section for details.

> Note
> It is possible to run multiple, independent debug sessions on one workstation. In this case, separate `config.t32` files are needed, each with the TMP variable pointing to different directories. Also, different IPC must be selected using the `VD_SESSION` variable. For more information, refer to Running and Bringing-Up the Virtual Debug Interface.

The connection through the shared library requires that the bit mode of the Trace32 matches not only the library, but also the simulation or emulation (which is most likely 64-bit). The TCP socket connection requires only the Trace32 debugger to match the client-library bit mode. But the Palladium or simulation can be of any mode.

The following practice script illustrates initializing the Trace32 debugger to use the JTAG protocol.

```
system.config.debugport gtl0 ; select GTL plugin
system.gtl.modelname "/vdipc" ; either share memory /name or host:port for TCP connection
system.gtl.modelconfig "bfmclockperiod=10ns"
system.gtl.libname "./libvdgtl.so" ; vdebug client library
system.gtl.jtagprobename "jtag_gtl"
system.gtl.dmaname "dma_gtl"
system.gtl.transactorconfig "jtag_gtl" "tbench.u_vd_jtag_bfm"
system.gtl.transactorconfig "dma_gtl" "tbench.u_memory.mem_array"
system.gtl.prebundle off
system.jtagclock 50MHz

; GTL logging - optional
;system.gtl.log.open "t32log.txt" /unbuffered
;system.gtl.log.addoption calls smartcalldata info
;diag 0x30e0 1
;diag 0x3208 1
;system.gtl.connect

; model-time based to avoid timeouts when emulation is stopped
system.vt.timeintargettime on
system.vt.pauseintargettime on
system.vt.maxpause 10us           ; limit the max pause duration

system.vt.pollingpause 1ms        ; Trace32 54083 and later;
```

```
system.cpu cortexa75
system.config.debugaccessport 0
system.config.coredebug.base 0x94110000
system.config.cti.base 0x94120000
system.config.etm.base 0x94140000

; for slow systems you may want to uncomment these settings
system.option.memstatuscheck on
map.updateonce                    ; update window content once;
system.polling slow               ; minimize target state inquiries
;setup.urate 1s                   ; reduce the screen update rate
;system.vt.timescale 0.1          ; scale the pauses down
;system.vt.maxtimeout 1ms        ; timeout in target time
;system.vt.hardwaretimeout off    ; disable hardware timeouts

;to.set swi on                    ; needed for the terminal output on 32b Cortex A
;actions
system.detect.daisychain          ; JTAG scan chain
;system.detect.dap                ; DAP scan, JTAG and DAP only
;system.mode.prepare              ; prepare mode allows bus access
system.up                         ; bring up to system ready state
area                              ; open a message window

; software load, memory set
;data.load.elf ../tests/init/init.elf /gtldmaload
;tdiag data.set <
      RAM_address
    >++0xfff %long 0xaaaaaaaa
;break.set main

; terminal setting for multihosting
;term.heapinfo 0x00007000 0x000070ff 0x000070ff 0x00007000
;term.method armswi
;term.mode string
```

The above script is available in the package in the `script` directory as `jtag_gtl.cmm`. The JTAG-specific settings in the script are as follows:

```
system.gtl.jtagprobename "jtag_gtl"
system.gtl.transactorconfig "jtag_gtl" "tbench.u_vd_jtag_bfm"
system.jtagclock 50MHz
system.detect.daisychain ; JTAG scan chain
```

A practice script, illustrating initializing the Trace32 to use the APB protocol, is available in the package in the `script` directory as `apb_gtl.cmm`. The APB-specific settings are listed in blue:

```
system.config.debugbusname "apb_gtl"
;system.gtl.gpioname "gpio_gtl" ; optional to drive core reset
system.gtl.transactorconfig "apb_gtl" "tbench.u_asic.u_vd_apb_bfm"
```

```
;system.gtl.transactorconfig "gpio_gtl" "tbench.u_asic.u_vd_apb_bfm"
```

Two practice scripts, illustrating the initialization of Trace32 to use the DAP protocol, are available in the package in the script directory, as `dap_gtl.cmm` and `dap6_gtl.cmm`. The DAP-specific settings are shown below in magenta.

```
system.config.dapname "dap_gtl"
;system.gtl.gpioname "gpio_gtl" ; optional to drive core reset
system.gtl.transactorconfig "dap_gtl" "tbench.u_asic.u_vd_swdp_bfm"
;system.gtl.transactorconfig "dap_gtl" "tbench.vd_dap6_bfm"
;system.gtl.transactorconfig "gpio_gtl" "tbench.u_asic.u_vd_swdp_bfm"
```

A practice script, illustrating initializing the Trace32 to use the AHB protocol is available in the package in the script directory as `ahb_gtl.cmm`. The AHB-specific settings are shown below in orange.

```
system.config.memorybusname "ahb_gtl"
;system.gtl.gpioname "gpio_gtl" ; optional to drive core reset
system.gtl.transactorconfig "ahb_gtl" "tbench.u_vd_ahb_bfm"
;system.gtl.transactorconfig "gpio_gtl" "tbench.u_vd_ahb_bfm"
```

A practice script, illustrating initializing the Trace32 to use the AXI protocol is available in the package in the script directory as `axi_gtl.cmm`. The AXI-specific settings are shown below in red.

```
system.config.memorybusname "axi_gtl"
;system.gtl.gpioname "gpio_gtl" ; optional to drive core reset
system.gtl.transactorconfig "axi_gtl" "tbench.u_vd_axi_bfm"
;system.gtl.transactorconfig "gpio_gtl" "tbench.u_vd_axi_bfm"
```

Customizing the Entries in the Scripts

Customize the following entries according to your design.

- `system.gtl.modelname "/vdipc" ; /name SHM; host:port for TCP`
  Specifies the connection mode in the format: `/name` — shared memory on the same workstation or `host:port` for TCP-socket connection. This value is overridden by the `VD_SESSION` environment variable.

- `system.gtl.transactorconfig "jtag_gtl" "tbench.u_vd_jtag_bfm"`
  Specifies the full hierarchical path to the BFM instance inside the emulated design.

- `system.gtl.modelconfig "bfmclockperiod=10ns"`
  Specifies a list of configuration parameters, separated by `|`.
  One mandatory parameter `bfmclockperiod` specifies the BFM-input clock period, (`clk_i`,

`pclk_i`, `dbgclk_i` — depending on the BFM type) as declared in the RTL sources. It is used to calculate the maximum JTAG clock frequency, as half of the frequency specified by the clock period. Combined with the `system.jtagclock` setting mentioned below, it determines the internal BFM divider. It is necessary to specify a correct value to calculate the divider properly. It also applies the correct timing from the Trace32.

- `system.gtl.prebundle off`
  Instructs Trace32 to **not** bundle the commands together to minimize the traffic. By default, Trace32 bundles the transactions together, which improves performance. This value should be used only during the bring-up phase.
- system.jtagclock 50MHz
  Specifies the desired JTAG clock frequency. It should not exceed the maximum frequency as determined from the `bfmclockperiod` specified in the transactor configuration.

The JTAG clock `tclk` is generated from the BFM clock (with `bfmclockperiod` frequency) and an integer divider. Its value is derived from the ratio of the BFM clock and `jtagclock` setting with the following formulas:

```
divider = INT((1/bfmclockperiod)/(2*jtagclock))
tclk    = (1/bfmclockperiod)/(2*divider)
```

The `bfmclockperiod` of 10ns and `jtagclock` 50MHz generate the fastest `tclk` of 50MHz:

```
divider = INT((1/10ns)/(2*50MHz)) = INT(100MHz/100MHz) = 1
tclk    = (1/10ns)/(2*1) = 100MHz/2 = 50MHz
```

The `bfmclockperiod` of 10ns and `jtagclock` of 20MHz generate 2x slower `tclk`:

```
divider = INT((1/10ns)/(2*20MHz)) = INT(100MHz/40MHz) = 2
tclk    = (1/10ns)/(2*2) = 100MHz/4 = 25MHz
```

> Note
> The integer divider can also be controlled using the `VD_DIVIDER` environment variable. For more information about this variable, refer to *Controlling the JTAG Frequency* section in Running and Bringing-Up the Virtual Debug Interface.

- The following setting specifies the core type followed by optional access port number, mandatory for ARM 8-cores-based debug and CTI addresses, followed by optional ETM-based address.

  ```
  system.cpu.cortexa75
  ```

```
system.config.debugaccessport 0
system.config.coredebug.base 0x94110000
system.config.cti.base 0x94120000
system.config.etm.base 0x94140000
```

- The `system.vt.*` settings control the timing of the debugger. The first two settings instructs Trace32 to use the simulator time for timeouts.

```
system.vt.timeintargettime on
system.vt.pauseintargettime on
```

- The following setting limits the pause duration issued by the Trac32 to 10us, which suits most emulated systems. Without this setting, the debugger issues 100ms pause after reset, which might take too long in emulation.

```
system.vt.maxpause 10us
```

- The next recommended setting is available from the Trace32 build 54083 onwards and lets the emulator execute run cycles, while debugger is polling for run status. It significantly speeds up the code execution. It can be set to milliseconds, which would further reduce the number of synchronizations at the cost of increasing the GUI response time.

```
system.vt.pollingpause 1ms
system.option.memstatuscheck on
```

This setting, applicable to ARM Architecture 8 cores, prevents the data override in the transfer register and might be mandatory for these cores.

- The following setting instructs Trace32 not to update the content of the show windows and minimizes the traffic.

```
map.updateonce   ; update window content once
```

For details, refer to the Trace32 Practice Script and GTL Debug User Guide.

Running Lauterbach Trace32

1. With the simulator or emulator session running, start the debugger in another terminal. For example, start the 64-bit Trace32 for ARMv7 with JTAG:
   ```
   > <install>/bin/pc_linux64/t32marm -c config.t32 -s jtag_gtl.cmm
   ```

   > Using the provided `jtag_gtl.cmm` script, the debugger tries to detect the scan chain in the DUT. When the device ID is listed in the message area window, the transactor works properly. If that does not work, either an error message is displayed, or the GUI appears to hang. Refer to the Creating Simultaneous Connections to Multiple Cores section to resolve such issues.

2. Uncomment the following line, and re-run the script:
   ```
   system.mode.prepare
   ```
3. Modify the script to uncomment the following line:
   ```
   system.up
   ```
   On successful completion, the transactor is installed, and the software can be debugged.
4. To assess the performance, execute the commented-out `tdiag` command in the `jtag_gtl.cmm` script.

   > You need to adjust the base address to point to the available RAM address that is at least 4KB long in your system. Result values that are significantly below one second are considered good.

5. Load the code with `data.load.elf` commented out in the sample `cmm` script.
   The `/gtldmaload` loads the image directly into the design memory, instead of using the bus. It improves the speed of loading the image by at least a 1000 times.

Debugging Lauterbach Trace32

The example `.cmm` file provided in the `scripts` subdirectory also includes commented-out lines for the Trace32 logger. Uncommenting these lines creates a trace log file named `t32log.txt`, which can be used to trace the GTL commands and data as issued and received by the Trace32 debugger.

```
system.gtl.log.open "t32log.txt" /unbuffered
system.gtl.log.addoption calls smartcalldata info
```

If the messages do not provide enough information, check the waveforms, especially immediately after integration when even the `system.detect.daisychain` does not work.

The JTAG signals, especially `TDO` connectivity, are worth checking. Once the JTAG signals connections are verified, the TAP state machine can be traced and compared with the state in the Trace32 debugger. The internal state can be written in the `t32log.txt` by uncommenting the following line:

```
;diag 3201 1 ; turn on extended dap diag for JTAG
```

Dynamic Switching of Trace32

Trace32 supports dynamic switching from one interface to another interface. It can use scripts to activate a JTAG interface and switch to APB interface.

See Dynamic Activation of Virtual Debug Interfaces section for details.

Using TPIU in Lauterbach Trace32

> For information about integrating the TPIU BFM in testbench, refer to the TPIU Integration section. For information about compiling the design with TPIU BFM, refer to the Compiling the Design section.

Customizing the TPIU Script

Customize the following entries according to your design.

- `system.gtl.tracename "tpiu_gtl"`

  `system.gtl.transactorconfig "tpiu_gtl" "tbench.u_vd_tpiu_bfm"`

  Specify the full hierarchical path to the TPIU BFM instance inside the design.

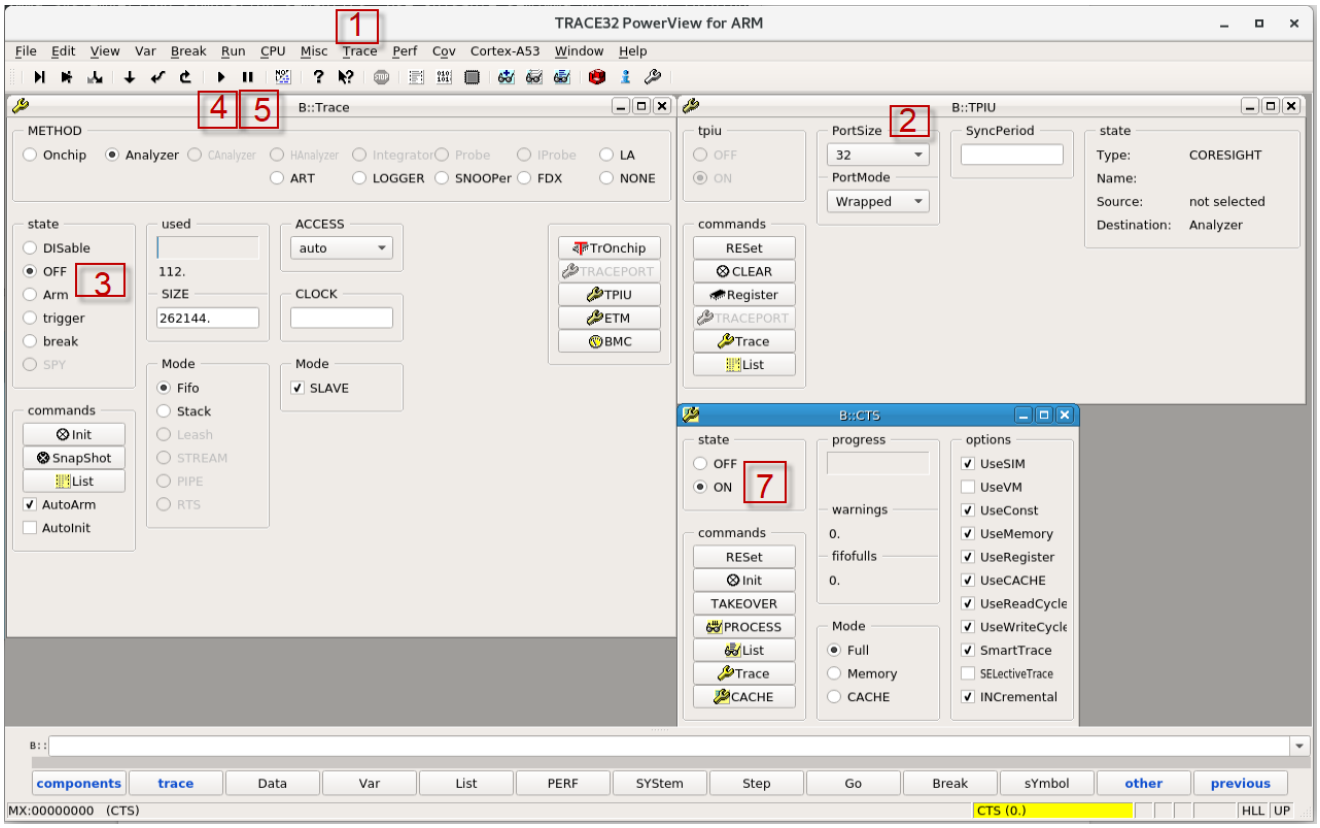
- `system.config.coredebug.base`
  `system.config.cti.base`
  `system.config.etm.base`
  `system.config.tpiu.base`
  `system.config.etb.base`
  Specify the necessary base address for Trace32 debugger.

Note

- For multi-core system, the selected test should be corresponding to the `coredebug.base`.
- `etm` : Embedded Trace Moule
- `tpiu`: Trace Port Interface Unit
- etb: Embedded Trace Buffer

Setting the Trace32 Debugger to Use TPIU

1. In Trace32 GUI, click the *Trace* button in top menu bar and select *Configuration*.
   The METHOD is set to ‘Analyzer’ automatically.
2. Click the *TPIU* button on right section of the Configuration window. In TPIU window, modify the data width to 32.
3. Change the state in Configuration window to *Arm*.
   In the TPIU window, the state is shown as *On* automatically.
4. Click *Run* to start debugging.
   The ‘used’ section in *Configuration* window increases, which means saving data into trace buffer.
5. After finishing the test, click the *Stop* button.
6. Click Trace and select *CTS Setting*.
7. Change the state of CTS to be *ON*, after finishing the ‘progress’, the Trace functionalities could be used.



Using DAP6 in Lauterbach Trace32

> For information about integrating the DAP6 BFM in testbench, refer to the DAP6 Integration section. For information about compiling the design with DAP6 BFM, refer to the Compiling the Design section.

Customizing the DAP6 Script

Customize the following entries according to your design.

```
system.cpu cortexa75
system.config.apbap1.base     DP:0x00040000
system.config.coredebug.base APB:0x01110000
system.config.etm.base        APB:0x01140000
system.config.cti.base        APB:0x01120000
```

The script `'APB:'` indicates that the address is accessible through the APB bus.

`APBAPn.Base <address>` is SoC-600 command. This command informs the debugger about the start address of the register block of the "`APBAPn:`" access port. It notifies the existence of the access port. An access port typically provides a control register block which needs to be accessed by the debugger to read/write from/to the bus connected to the access port.

Related Section:

Creating Simultaneous Connections to Multiple Cores

Configuring and Running Xtensa OCD

This topic describes the processes of configuring, running, and debugging the Xtensa OCD debugger.

Configuring Xtensa OCD

The vdebug release ships with the Xtensa OCD client library, `libvdxocd`. The library matching your environment needs to be copied into the `xocd` installation tree, under the `modules` subdirectory, possibly overwriting an existing, older version of the library.

For `xocd 14.0.*` and later releases, client library is `lib/64bit/libvdxocd.so` for Linux and `lib/64bit/libvdxocd.dll` for Windows.

Xtensa OCD server uses a configuration file in the XML format, commonly called a topology file. One controller section specifies the configuration of the `vdebug` library is shown below.

```
<configuration>
          <controller id='Controller0' module='libvdxocd' debug=''
          host='pdhost' port='8192'
          bfm-path='tbench.u_vd_jtag_bfm' bfm-clock='10000'
          poll-low='5000' poll-high='50000'
          dmi-inst='tbench.dram0.mem.dataArray:0x3ff00000:0x40000,
          tbench.iram0.mem.dataArray.0x40000000:0x100000'
          type='jtag' jtck-div='1'
          />
</configuration>
```

In the above code, the settings imply the following:

- `module='libvdxocd'`
  Specifies the client-module name.
- `host='pdhost' port='8192'`
  Specifies the host name and the port of the vdebug server. Overridden by setting the `VD_SESSION` environment variable. See the Running and Bringing-Up the Virtual Debug Interface section for details.
- `bfm-path='tbench.u_vd_jtag_bfm'`

Specifies the hierarchical path to the BFM Instance.

- `bfm-clock='10000'`
  Specifies the BFM input clock cycle in period in `ps` (`clk_i`, `pclk_i`, `dbgclk_i` — depending on the BFM used). The value `10000` and above indicates the clock frequency of 100MHz (`1/10^-8`)`ps`.
- `poll-low='5000' poll-high='50000'`
  Specifies the polling rate, when the debugger is idle or running. The rate is dynamically adjusted between the low and high values. Both values specify the number of BFM clock cycles that the hardware executes while checking for a new command from the debugger. Larger values result in faster emulation or simulation; lower values reduce the lag with which vdebug responds to debugger requests. The recommended value for the poll-low is the value that leads to an acceptable emulation performance and minimum lag for the debugger response. The poll-high value should have a value that results in the response lag to be around 1 second. The actual values depend on the design characteristics. But the pair of 5000 and 50000 is a good starting point.
- `dmi-inst='mem_path:base:size<:offset>, mem_path:base:size<:offset>...'`
  Specifies up to 20 instances of design memories for the back-door access through vdebug. Each memory instance needs to be identified by its full hierarchical path, its base address in the design memory map, and by its size in bytes. The <> denote an optional entry. The default value of offset is 0.

  > ⓘ The back-door memory access can only be used if the memory instances occupy unique and continuous address in the design address space. Memories with slices or bit splits cannot be used.

- `type='jtag' or type='apb'`
  Specifies the library access type: `jtag` for JTAG access and `apb` for APB access.
- `jtck-div='1'`
  Specifies the JTAG clock divider. The minimum value of `1` results in the JTAG clock being half the frequency of the clock driving the JTAG BFM, which is the fastest possible value. Higher numbers produce slower clocks.

Sample scripts, `xocd-vdebug-jtag.xml` and `xocd-vdebug-apb.xml`, are provided in the package in the `script` directory. These files contain the required components for use.

Follow the `topology-example.xml` file inside the `xocd-14.0*` folder or the Xtensa Debug Guide for information about adding more components needed by specific testcase.

Running Xtensa OCD

With the simulator or emulator session running, start the debugger in another terminal.

```
> <install>/xt-ocd -c xocd-vdebug-jtag.xml -srst -trst -l xocd.log
```

Debugging Xtensa OCD

The XOCD provides several run time options to enable command trace and debug at different levels.

```
> <install>/xt-ocd -c xocd-vdebug-jtag.xml -srst -trst -dTD -dC -l xocd.log
```

For details, refer to the Xtensa OCD section in the Xtensa Debug Guide.

Configuring and Running Development Studio

This topic describes the processes of configuring, running, and debugging the Development Studio.

Configuring the Development Studio

Use the Platform Configuration Editor in the Development Studio to add or create a new debug platform, and configure a new debug adapter.

Creating User Configuration Database:

1. Create a user Configuration Database, if it does not already exist. To create one, select *File* -> *New* -> *Other* option from the menu.
2. Select *Configuration Database* under *Configuration Database* group. Then, click *Next* and enter a new name, for example, `vdebug`. Click *Finish*.

Creating New Platform Configuration:

1. Create a new Platform Configuration by doing a right-click on *Configuration Database* and selecting *New/Platform Configuration*.
2. Select *Advanced Platform Detection* or *manual configuration* and click *Next*.
3. Enter *Platform Manufacturer* and *Name*, such as `vdebug_core_jtag|swdp`. Then, click *Finish*.
4. Enter (<*your_company*>) and *Name*, such as `vdebug_<core>_jtag` or `vdebug_<core>_SWDP`. Click *Finish*.

Updating the vdebug Libraries:

> Note
> Starting with PDAPP 22.11, the vdebug release includes client libraries for Development Studio. Cadence recommends to copy the client libraries into your configuration database under your workspace.

1. Right-click your *Configuration Database* visible in the Project Explorer and select *Properties*. The location of your Configuration Database on your hard drive is displayed.

2. Create subdirectories *Probes/Cadence* under your configuration database and copy *<DS_Install>/sw/debugger/configdb/Probes/Cadence/probes.xml.*
3. Open `probes.xml` from your configuration database in any text editor. Change the name tag under probe (in line 9) to distinguish the newest probe. For example, "`Cadence VDebug Latest`". Change the name under `debug_lib` tag in the line below to "`vdrddi`". Save the file and close the editor.
4. Copy the client library from the vdebug release *<vdebug_install>/lib/64bit/* to the directory where the modified `probes.xml` has been saved.

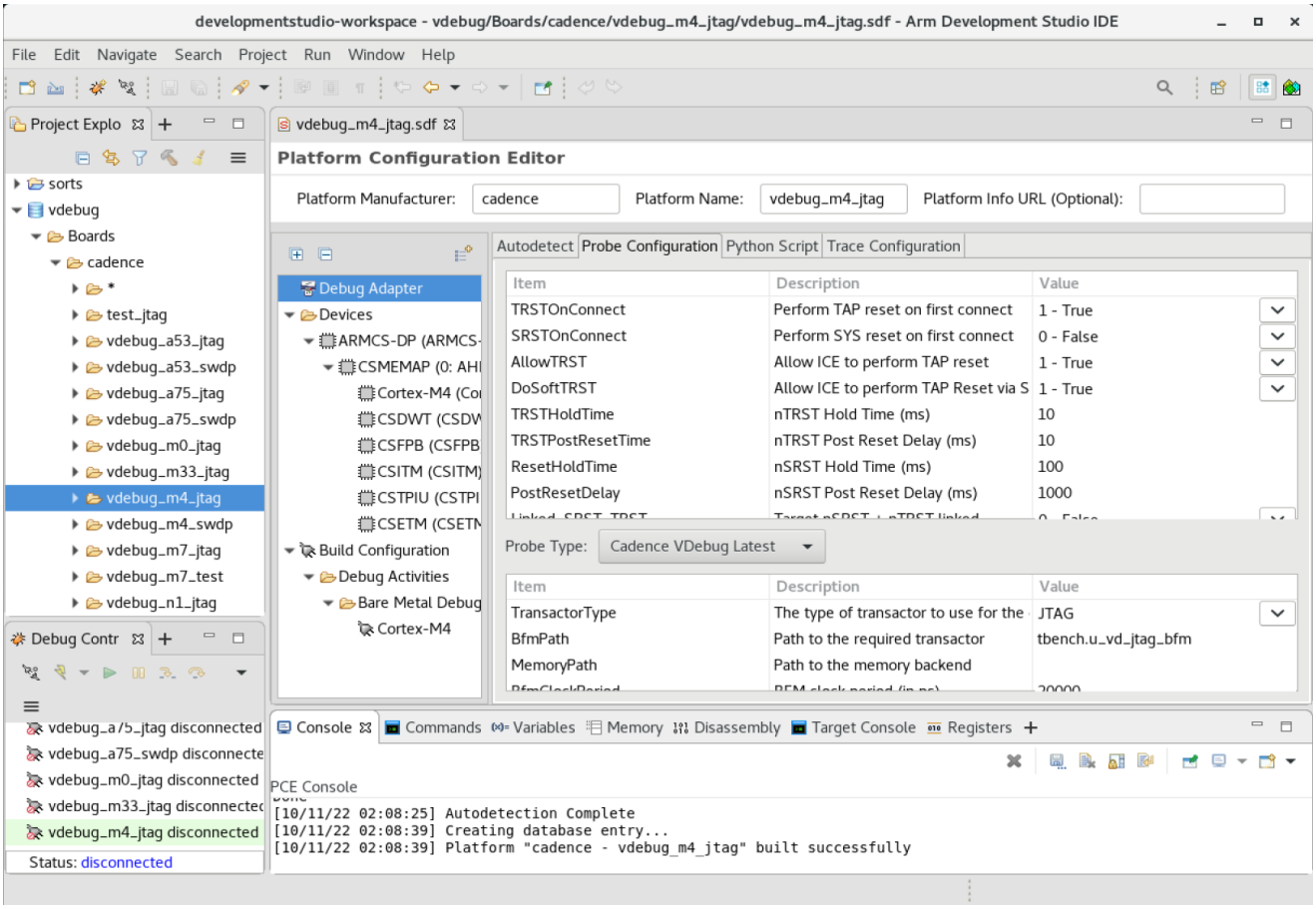
   > The library file to copy is called `libvdrddi.so.2` for Linux and `vdrddi_2.dll` for Windows.

5. Restart Development Studio for the updated library to be visible to Development Studio .

Specifying Debug Adapter Configuration:

1. On the *Autodetect Tab,* select your updated library *Cadence VDebug Latest* or *Cadence Virtual Debug* from the drop-down menu and specify the connection to the vdebug server.
   a. Either by specifying the hostname and TCP port of the workstation on which the vdebug server is running and listening on that TCP port. The format is <*host*>:<*port*>. For example, `pdhost04:8192`.
   b. Or by specifying the shared memory name, if the Development Studio runs on the same workstation as the vdebug server. The format is /<`name`>. For example, `/myshm`.
2. Select your updated library *Cadence VDebug Latest* or *Cadence Virtual Debug* on the *Probe Configuration* tab as the *Probe Type* from the drop-down menu.
3. Select *JTAG* for *JTAG BFM* and *SWDP* for *SWDP BFM* as *TransactorType*.
4. Specify the full hierarchical path to the BFM in the design as *BfmPath*.
5. Specify the clock period in `ps` of the design clock driving the BFM as *BfmClockPeriod*. For the JTAG interface, this value is used to calculate the generated JTAG clock frequency.
6. Leave *UseBatchAccell* as *True* unless debugging.
7. Specify the path to the memory instance for the back-door image load as *MemoryPath*. The option to directly load the image is not enabled by default, and requires an additional script from Arm. Contact Arm Support for details.
8. Specify where to create debug log file, when *DebugLevel* > `0` is set under *VdLogPath*.

9. Specify a numeric *DebugLevel*. Specify values greater than `0` when bringing up the design. Adding `0x100` to the value (like `0x101`) echoes the debug messages to the *Development Studio Console*, which might have an adverse effect on performance.



10. Go back to the *Autodetect* tab. For JTAG connection, specify the *Clock Speed* as half of the frequency specified as *BFMClockPeriod* on the *Probe Configuration* tab. For instance, for *BFMClockPeriod* of `10000ps`, specify *Clock Speed* of `50MHz`.
11. After specifying all these values, and making sure that vdserver and emulation or simulation with an Arm core is running, click on *Autodetect Platform*. PCE queries the design and fills up the detected *Devices and Debug Activities*.
12. Save the configuration through *File–Save* option.
13. Right-click the configured platform and build it by selecting *Build Platform*.

If any of the above steps fail, refer to the *Debugging Development Studio* section below, or generate debug log (see steps 8 and 9 above).

Running Development Studio

With the simulator or emulator session running, the configured debug platform can be connected to the core.

1. Switch to the *Development Studio* perspective from *Window/Perspective/Open Perspective/Development Studio*.
2. Open *Debug Configurations* from *Run/Debug Configurations* or right-click in *Debug Control*.
3. Select *Generic Arm C/C++ Application* and click on *New* icon on the toolbar on the left side.
4. On the *Connect* tab, filter platforms by specifying the name you have given to the platform, for example `vdebug` and select the core to be connected under *Bare metal Debug*. For multi-core systems, you can specify a single core, a cluster, and all cores.
5. Select your updated library *Cadence VDebug Latest* or *Cadence Virtual Debug* as a target connection and specify the host and port where the vdebug server is running as <*host*>:<*port*>. For example `pdhost04:8192`. If the Development Studio runs on the same workstation where the emulation or simulation is running, the shared memory can be used for communication. In this case, enter the shared memory name prefixed by forward slash. The format is /<*name*>, for example, `/localhost`.
6. Specify the configuration name reflecting the selected platform and core, such as `vdebug_m4_swdp`, and click Apply.
   The saved configuration appears in the *Debug Control* window.
7. Connect to the target core by double-clicking the selected configuration. You can check the messages in *Target Console*.

Debugging Development Studio

The diagnostic messages appear in the PCE Console during the platform configuration phase and in the Target Console during the connection to the core.

Configuring and Running Green Hills MULTI

This topic describes the processes of configuring, running, and debugging the Green Hills MULTI debugger.

Configuring MULTI Debugger

The interface consists the agent that implements the virtual-debug interface client and the probe. Both are supplied by Green Hills Software. Contact the Green Hills technical support for details.

Configuring and Running the Agent

The `vdebug_gnj_agent` program, which is installed in the GHS compiler directory, connects to the vdebug server and acts as an interface to the rest of the Green Hills tools. It must be manually invoked and must remain running for the duration of the debug session.

Run the program without arguments to see usage information and vdebug-specific options. The `-i` and `-j` options of the agent change the effective JTAG clock speed, which might affect debug performance.

For example, to connect to `tbench.u_vd_jtag_bfm` running on `pdhost` port `8192`, use a `/2` JTAG divisor, and listen for connections on port `5005`:

```
% ./vdebug_gnj_agent -j 2 5005 pdhost 8192 tbench.u_vd_jtag_bfm
```

vprobe or probe Configuration

The `.ghpcfg` files are the authoritative configuration format for both vprobe and physical Green Hills Probes. A `ghpcfg` file is strongly recommended for use when connecting MULTI to the debug server (`MPserv`), detailed in the Running and Bringing-Up the Virtual Debug Interface section.

If you do not have a `ghpcfg` file, you can use your probe or vprobe in standalone mode to configure

appropriate options and save the configuration.

- (vprobe only) Launch vprobe standalone, and open the Telnet interface. This provides a prompt equivalent to the telnet interface of a Green Hills Probe.

  ```
  $ghs_compiler_directory/vprobe/vProbe -T2323
  & telnet localhost 2323
  ```

- (Physical-probe only) Open the telnet interface or the ProbeAdministrator. Then, set the `debug_type` option:

  ```
  telnet $ghprobe_hostname
  set debug_type gnj
  ```

- Set the following options for IP and listening ports:
  This must be the IP of `vdebug_gnj_agent` host, such as `127.0.0.1`.

  ```
  set gnj_server_ip $agent_ipaddr ;;
  ```

  This must be the `vdebug_gnj_agent` listen port, such as `5005`.

  ```
  set gnj_server_host $agent_port ;;
  ```

- Set other probe-configuration options as required. Refer to the *Green Hills Debug Probes User Guide* for details of the commands and options.

- You can also test the connection to the agent and underlying vdebug server:
  `jr ;;` force tap reset
  `tl ;;` list core status

- When finished, exit telnet.

- Save the current configuration:

  ```
  $ghs_compiler_directory/mpadmin -cfgsave localhost my_target.ghpcfg
  ```

- (vprobe only) Switch back to vprobe and end the process:

  ```
  fg <ctrl-c>
  ```

Following is an example of a minimal `ghpcfg` file for a PULPissimo RISC-V system:

```
{
  "version" : [3,0],
  "config" : [
    {
      "options" : {
        "global" : {
          "gnj_server_ip" : "not_set",
          "gnj_server_port" : "not_set",
          "target" : "rv32i",
          "checker" : "on",
          "verify_download" : "sparse"
        },
        "core 0" : {
          "alias" : "",
          "endianness" : "little",
          "hartsel" : "0x20",
          "rv_cache" : "none"
        }
      }
    }
  ]
}
```

In the `ghpcfg` file, setting the checker option to `off` can improve responsiveness while using MULTI. But it disables status updates while using the probe/vprobe in standalone mode.

Running MULTI

Ensure that the agent is running. See *Configuring and Running the Agent* section in the Configuring and Running Green Hills MULTI topic.

If you do not have a project, create one using the following steps:

1. Create a new workspace with a new project if it does not already exist. To create one, select *File -> Create Workspace -> New Project* from the menu. Then, click *OK*.
   The Project Wizard window opens, and the path for workspace should be specified in Directory.
2. Click *Next* and select *Stand-alone for the Operating System*. Then, click *Next* again and select *Generic-RISC-V for Target Board*. Click *Finish* to complete the setup of the Top Project.
3. Select an example to add.
   A good example to start with for a stand-alone project is *"Hello World (C)"*.
4. Click *Finish* to accept the default settings. The *MULTI Project Manager* window opens.

5. Select `src_hello\hello.gpj` in the *Project Manager*, click the *Build* button from menu bar or press `F7` to build the project.
6. You might want to modify the linkmap (`standalone_ram.ld`) or other build options to match the memory configuration of your target.

Connect and run:

1. Click the *Connect* button from menu bar or press F4 to open *Connection Chooser*. To connect to a hardware target, select the combination of target and debug server connection like `"Generic-RISC-V rv32im with Green Hills Debug Probe (mpserv)"`.
2. For Physical-probe, Choose *Ethernet/IP Connection* or *USB* connection as appropriate, and fill in the probe *hostname* as required.
3. On the "*Probe Config*" tab, select the `.ghpcfg` file that was saved previously as part of configuring the debugger.
4. For vprobe, on the "*Advanced*" tab, add `-vprobe $host:$port` arguments and replace with the host and listening port of the `vdebug_gnj_agent` process, such as `-vprobe localhost:5005`.

   > These parameters override the `gnj_server_ip` and `gnj_server_port` configuration options.

5. Click *OK* to close the connection editor. Then, click *Connect*.
6. Click the *Debug* button from menu bar or press F5 to open the MULTI Debugger. Click *Run* and select the specified connection.
7. Click *OK* in the '*Prepare Target*' box to download the program to RAM and run.
   In certain circumstances, a "*Terminate Connection*" dialog might appear if the download does not finish in a few seconds. It can be safely ignored; The dialog box disappears automatically when the download is complete.

Debugging MULTI

Before contacting GHS support, collect the following information:

- A list of MULTI or compiler versions, along with installed patches.
- A log from `vdebug_gnj_agent`, generated using the `VD_DEBUG` and `VD_LOG` environment variables. For information about the `VD_DEBUG` and `VD_LOG` environment variables, refer to the

Debugging the Virtual Debug Interface Issues section in the Running and Bringing-Up the Virtual Debug Interface topic.

- The full command-line invocation of `vdebug_gnj_agent`.
- The output from target support when connected (from either Telnet or MULTI).

Configuring and Running OpenOCD

This topic describes the processes of configuring, running, and debugging the OpenOCD debugger.

Compiling OpenOCD from Sources

You can download OpenOCD from the sourceforge site: https://sourceforge.net/projects/openocd/files/openocd/.

OpenOCD version 0.12 or later includes vdebug driver. It needs to be specifically enabled during the configuration step.

> Backdoor Memory Access and Polling
> To use backdoor memory access and polling, the sources need to be patched. Contact Cadence Customer Support for patch requests.

To compile OpenOCD:

1. Download and unpack the sources from the sourceforge site mentioned above.
2. Optionally, patch the sources by unpacking the patch into the OpenOCD directory: `tar xzf vdebug.diff.tgz`.
3. Patch the release by merging vdebug client into it: `patch -p1 < patch.diff`
4. Configure, compile, and install.

```
./configure --prefix=<install_path> --enable-vdebug=yes
make
```

```
make install
```

ⓘ If the release has been patched, the OpenOCD scripts from the vdebug release can be used. Otherwise, the scripts included in OpenOCD need to be used.

Configuring OpenOCD

The OpenOCD configuration is usually split across different script files for defining interface and target-specific settings. The interface script and target script are included from board-specific script, which alone can be referred from the command line.

Interface Settings

The interface settings are as follows. You can override the hostname and port by defining the `_VDEBUGHOST` and `_VDEBUGPORT` variables in other scripts.

```
if { [info exists VDEBUGHOST] } {
 set _VDEBUGHOST $VDEBUGHOST
} else {
 set _VDEBUGHOST localhost
}
if { [info exists VDEBUGPORT] } {
 set _VDEBUGPORT $VDEBUGPORT
} else {
 set _VDEBUGPORT 8192
}

adapter driver vdebug
vdebug server $_VDEBUGHOST:$_VDEBUGPORT

bindto 0.0.0.0

vdebug batching 1
vdebug polling 100 1000
```

Setting	Description
`adapter driver vdebug`	Mandatory. Activates the virtual debug interface.
`vdebug server localhost:8192`	Mandatory. Specifies the host and TCP, where the vdebug server is running.

bindto 0.0.0.0	Optional OpenOCD setting. Specifies hostname or Ipv4 address on which to listen for incoming TCP/IP connections. Specifying `0.0.0.0` enables connections from all hosts, by default OpenOCD accepts only the local connections.
vdebug batching 2	Optional. Specifies batching write requests `1` or '`w`', no batching `0` or '`off`' or read and write batching `2` or '`r`'. The default is `1`.
vdebug polling 100 1000	Mandatory. Specifying to polling frequency low, used when core is in the debug mode, and high, when the core is running. Both numbers represent the number of wait cycles used in the polling loop.

Target Settings

The target settings are as follows. The `_HARTID` and `_CHIPNAME` variables can be defined in the board settings.

```
if {![info exists _HARTID]} {
 set _HARTID 0x00
}
if {![info exists _CHIPNAME]} {
 set _CHIPNAME riscv
}

set _TARGETNAME $_CHIPNAME.cpu
target create $_TARGETNAME riscv -chain-position $_TARGETNAME -coreid $_HARTID

riscv set_reset_timeout_sec 120
riscv set_command_timeout_sec 120
riscv set_mem_access sysbus progbuf

proc vdebug_examine_end {} {
 vdebug register_target
}
$_TARGETNAME configure -event examine-end { vdebug_examine_end }
```

Setting	Description
riscv set_mem_access sysbus progbuf	Mandatory. Specifies the access to the design memory in the preference order, through a system bus, using buffer.

proc vdebug_examine_end {} { **vdebug register_target** **}** **$_TARGETNAME configure -event** **examine-end {** **vdebug_examine_end }**	Mandatory. Invokes the vdebug register target procedure at the end of target examine phase.

Board Settings

The board settings are as follows. In this example, the `_CHIPNAME _HARTID` and `_CPUTAPID` variables are defined and passed to the target script.

Settings for riscv ibex Core for JTAG Interface

```
source [find interface/vdebug.cfg]

set _CHIPNAME ibex
set _HARTID 0x20
set _CPUTAPID 0x249511c3

transport select jtag

reset_config trst_and_srst

adapter speed 12500
adapter srst delay 10

vdebug bfm_path tbench.u_vd_jtag_bfm 40ns

vdebug mem_path tbench.soc_domain_i.pulp_soc_i.gen_mem_l2_pri\[0\].sram_i.mem_array 0x1c000000
0x8000
vdebug mem_path tbench.soc_domain_i.pulp_soc_i.gen_mem_l2_pri\[1\].sram_i.mem_array 0x1c008000
0x8000
vdebug mem_path tbench.soc_domain_i.pulp_soc_i.gen_mem_l2\[0\].sram_i.mem_array 0x1c010000
0x80000

jtag newtap $_CHIPNAME cpu -irlen 5 -ircapture 0x05 -irmask 0x1f -expected-id $_CPUTAPID
jtag arp_init-reset

source [find target/vd_riscv.cfg]
```

Settings for Cortex m4 Core for DAP Interface

```
source [find interface/vdebug.cfg]

set _CHIPNAME m4
set _MEMSTART 0x00000000
set _MEMSIZE 0x10000

transport select dapdirect_swd

adapter speed 25000
adapter srst delay 5

vdebug bfm_path tbench.u_vd_swdp_bfm 20ns

vdebug mem_path tbench.u_mcu.u_sys.u_rom.rom $_MEMSTART $_MEMSIZE

source [find target/swj-dp.tcl]

swj_newdap $_CHIPNAME cpu -irlen 4 -ircapture 0x1 -irmask 0xf

source [find target/vd_cortex_m.cfg]
```

Setting	Description
transport select jtag	Optional. Specifies the JTAG transport, which is a default.
transport select dapdirect_swd	Optional. Specifies the DAP transport, if the testcase supports it.
reset_config trst_and_srst	Optional. Activates the `TRST` and `SRST` signals by reset.
adapter speed 12500	Mandatory. For JTAG, it specifies the JTAG clock frequency.
adapter srst delay 10	Optional. Specifies the reset signal pulse width in clock cycles.

vdebug bfm_path tbench.u_vd_jtag_bfm 40ns

Mandatory. Specifies the hierarchical path to the vdebug BFM and the frequency period of its input clock.

```
vdebug mem_path tbench.soc_domain_i.pulp_soc_i.gen_mem_l2_pri\[0\].sram_i.mem_array
0x1c000000 0x8000
vdebug mem_path tbench.soc_domain_i.pulp_soc_i.gen_mem_l2_pri\[1\].sram_i.mem_array
0x1c008000 0x8000
vdebug mem_path tbench.soc_domain_i.pulp_soc_i.gen_mem_l2\[0\].sram_i.mem_array
0x1c010000 0x80000
```

Optional. Specifies the memories for the backdoor access. Up to four memory instances can be specified. Each memory instance is identified by a hierarchical path, start address and size in bytes.

The backdoor memory access can only be used if the memory instances occupy a unique and continuous address in the design address space. Memories with slices or bit splits cannot be used.

ⓘ For more information, refer to the OpenOCD User Guide.

Running OpenOCD

With the simulator or emulator session running, start the debugger in another terminal. Usually, a board script including other scripts is referenced from the command line.

```
> <install>/bin/openocd -f share/openocd/scripts/board/vd_m7f_jtag.cfg
```

Debugging OpenOCD

The OpenOCD script file includes commented-out lines for the log output file.

```
debug_level 3
log_output vd_ocd.log
```

The `debug_level` can be set to `0` to 4. This setting affects the kind of messages sent to the server log.

- Level `0` is error messages only
- Level `1` adds warnings
- Level `2` adds informational messages
- Level `3` adds debugging messages
- Level `4` adds verbose low-level debug messages.

The default is level `2`. The `log_output` specifies the log name.

9

Dynamic Activation of Virtual Debug Interfaces

More than one Bus Functional Models (BFMs), for instance, the JTAG BFM and the APB BFM can be compiled into the design. The BFMs can be individually activated using the reset signal. When any of the BFMs gets out of the reset state, it notifies the server and prints the following on the screen:

```
Xns: vd_server 48 pdapp 23.03 bfm JTAG start; polling 1000 shared mem /localhost
listening on TCP port 8192
```

When another BFM gets active, the server prints out:

```
Xns: vd_server APB BFM activated, 2 active
```

The interfaces can also be deactivated, and the server prints out:

```
Xns: vd_server APB BFM down, 1 active
```

When the last active BFM is deactivated, the vdserver shuts itself down:

```
Xns: vd_server JTAG BFM down, 0 active
```

```
Xns: vd_server 48 stop; closing TCP socket 11 port 8192 deleting shared mem
/localhost size 8192
```

The vdserver is restarted when any of the compiled BFMs become active again. The reset signal used to activate the BFM cannot contain glitches.

> ⓘ The debugger can use only one interface at a time. The interface that will be activated depends on the script. You can choose to activate one interface, disconnect it, and activate another interface in a single debugger connection.

Trace32

Following is a sample Trace32 script for activating a JTAG interface and switching to the APB interface:

```
system.config.debugport gtl0
system.gtl.modelname "/vdipc"                          ; /name SHM, host:port TCP
system.gtl.modelconfig "bfmclockperiod=10ns"
system.gtl.transactorconfig "jtag_gtl" "tbench.u_vd_jtag_bfm"
system.gtl.transactorconfig "dma_gtl" "tbench.u_memory.mem_array"
system.gtl.transactorconfig "apb_gtl" "tbench.u_cssys.u_vd_apb_bfm"
system.gtl.libname "./libvdgtl.so"                     ; GTL shared library name
system.gtl.jtagprobename "jtag_gtl"                    ; configure the JTAG access
system.gtl.dmaname "dma_gtl"
system.jtagclock 50MHz
...
system.cpu cortexa53
...
system.up
data.load.elf ../tests/aarch64/init/init.elf /gtldmaload
print "JTAG connected, up and software loaded"

system.down
system.gtl.jtagprobename ""
system.config.debugbusname "apb_gtl"                   ; configure the APB access

system.up
print "APB connected"
```

ⓘ For more information, refer to the Trace32 Processor Architecture Manual.

Xtensa OCD

Restart the Xtensa OCD daemon with a different configuration script to switch between JTAG and APB interfaces.

Development Studio

The Platform Configuration Editor supports JTAG and DAP interfaces. However, a target platform can be configured only with one JTAG or SWDP interface. Consequently, a debug configuration uses one configured interface.

MULTI

The `vdebug_gnj_agent` program acts as an interface to the rest of the Green Hills tools, and the option of this program specifies the hierarchical path to vdebug BFM. So, the entire programs need to be restarted with a different hierarchical path to switch to another interface.

OpenOCD

Restart the OpenOCD with a different configuration script to switch between JTAG and DAP interfaces.

Related Sections:

- Integrating BFM Modules with the Design
- Creating Simultaneous Connections to Multiple Cores

10

Creating Simultaneous Connections to Multiple Cores

One instance of the transactor supports simultaneous connections to different cores on the same JTAG chain or bus, driven by the BFM. This topic provides information about configuring Trace 32 and Arm Development Studio for creating simultaneous connections to multiple cores.

Green Hills Multi also supports simultaneous connections to multiple cores. Contact Green Hills Support for help.

Configuring Trace32 for Simultaneous Connections to Multiple Cores

With Trace32, such a setup requires the `t32mciserver` to run on the Palladium Infiniband host, and needs a special configuration on the client side as described below, exemplified by two instances of Trace32 connecting to the same multi-core module.

Trace32, connecting to core 1, should have `-c configmp0.t32`:

```
PBI=MCISERVER
PORT=5001           ; port on which the t32mciserver listens
NODE=cva-demo4      ; host on which t32mciserver is running
INSTANCE=1          ; core instance
```

The practice script (`-s testmp0.cmm`) should have the following instead of `system.cpu.cortexa7`:

```
system.cpu.cortexa7mpcore                           ; MP core
system.config.debugaccessport 0                     ; DAP APB index
system.config.corenumber 2                          ; number of cores
system.config.coredebug.base 0x80110000 0x80112000  ; core addresses
system.config.core 1 1
core.assign 1                                       ; first core selected
```

Trace32, connecting to core 2, should have `-c configmp1.t32`:

```
PBI=MCISERVER
PORT=5001                              ; port on which the mciserver listens
NODE=cva-demo4                         ; host on which mciserver is running
INSTANCE=2                             ; core instance
```

The practice script (`-s testmp1.cmm`) should have the following instead of `system.cpu.cortexa7`:

```
system.cpu.cortexa7mpcore                            ; MP core
system.config.debugaccessport 0                      ; DAP APB index
system.config.corenumber 2                           ; number of cores
system.config.coredebug.base 0x80110000 0x80112000   ; core addresses
system.config.core 2 1
core.assign 2                                        ; second core selected
```

Start `t32mciserver` in a directory in which `libvdgtl.so` is available:

```
<trace32install>/bin/pc_linux64/t32mciserver port=5001 verbose
```

Multiple TAP, debug ports, and ***vdebug*** BFMs in the design are supported. However, only one transactor, specified by `transactorconfig` can be activated and connected at run time.

Support for Simultaneous Connections to Multiple Cores for ARM Development Studio

ARM Development Studio supports simultaneous connections to multiple cores, when a multicore configuration has been selected during the configuration phase.

Related Sections:

- Improving Transactor Performance
- Configuring and Running Lauterbach Trace 32
- Configuring and Running Development Studio
- Configuring and Running Green Hills MULTI

11

Improving Transactor Performance

The performance of the transactor is impacted by various factors at different levels. The number of synchronizations plays a pivotal role; however this number is determined by the debugger and the core being debugged. For instance, the ARM Cortex A cores require nearly three times more operations than Cortex M Cores.

The performance can be measured by the single-step-execution time and the code-execution time.

Measuring Single-Step Execution Time

This section explains how to measure the single-step execution time for various debuggers.

Trace 32

The single-step-execution time is reported by Trace32 after you type the following command on the Trace32 prompt, when the core is connected in the ***up*** mode:

```
B:: tdiag.step.asm
```

XtensaOCD

The single-step-execution time can be measured gdb connected to XOCD after defining a procedure `timeme`, as follows.

```
define timeme
shell echo set \$starttime=$(date +%s.%N) > /tmp/timeme
source /tmp/timeme
$arg0
shell echo print $(date +%s.%N)-\$starttime > /tmp/timeme
source /tmp/timeme
end
gdb> timeme ni
```

Development Studio	A python diagnostic script can be provided to assess the debugger performance. Contact Cadence Customer Support for assistance.

Multi	You can measure single-step execution time at a low level with:
	`MULTI> target time si`
	Single-step execution time at a high level can be measured with:
	`MULTI> starttime; si; wait; endtime`

OpenOCD	The single-step-execution time can be measured gdb connected to OpenOCD after defining a procedure `timeme`, like shown below:
	`define timeme`
	`shell echo set \$starttime=$(date +%s.%N) > /tmp/timeme`
	`source /tmp/timeme`
	`$arg0`
	`shell echo print $(date +%s.%N)-\$starttime > /tmp/timeme`
	`source /tmp/timeme`
	`end`
	`gdb> timeme ni`

Impact of BFM Clock Settings on Single-Step-Execution Time

The most important parameter determining the performance is the **frequency of the clock** driving the BFM.

As the BFM clocks directly determine the interface performance, Cadence recommends to use as high frequency as possible. However, there might be other clocks in the design that might limit the interface, or even render it non-functional. For example, using very fast clocks to drive the JTAG BFM would not help much if the DAP or core clock is slow. This scenario might even lead to overrun errors.

Using a faster clock to drive the BFM might only be possible after elimination of the bottleneck clock—by increasing its relative frequency. Modifying the design clock frequencies might, however, lead to functionality problems. In such cases, you should empirically find out the fastest BFM clock frequency, at which the design is still functional.

Following are specific clock settings:

JTAG Cock Settings	The JTAG clock `tclk` frequency is derived from the BFM clock frequency and the clock settings according to the following formula:
	`tclk = (1/bfmclockperiod)/(2*divider)`
	`divider = INT((1/bfmclockperiod)/(2*jtagclock))`
	When `jtagclock=(1/bfmclockperiod)/2`, the `tclk` runs at half of the BFM clock. The BFM clock should, therefore, run as fast as possible for best performance.
	The integer divider can also be controlled using the `VD_DIVIDER` environment variable. For more information, refer to *Controlling the JTAG Frequency* section in Running and Bringing-Up the Virtual Debug Interface.

APB, AHB, and AXI Clock Settings	The bus clock drives the transactor directly. The transactor speed is directly related to the relative frequency of this clock.
DAP Clock Settings	The DAP clock (`dbgclk_i`) drives the transactor directly. The transactor speed is directly related to the relative frequency of this clock.

Using The Debugger Built-in Batching

The internal batching is another parameter that impacts performance. Some debuggers can be instructed to batch the transactions together, thereby, improving performance.

Trace 32	Batching is enabled by default. It can be disabled in the `cmm` file by setting the following: `system.gtl.prebundle off`
Xtensa OCD	Batching is enabled by default and cannot be changed.
Development Studio	Batching is enabled by default. You can disable it in *Platform Configuration Editor* and *Debug Adapter* on *Probe Configuration* after selecting *Cadence Virtual Debug* by setting `UseBatchAccel` to `False`.

OpenOCD

Batching is enabled by default if the batching is enabled for writes. It can be controlled by:

vdebug batching <*value*>, where value can be: 0, 1, or 2.

- 0: Disables batching
- 1: Enables write operations batching
- 2: Enables both reads and writes to be batched. This value causes the minimum number of synchronizations and provides best performance.

Determining the Code-Execution Time

Trace32

The non-interactive run-time performance is determined by the number of cycles that the emulator can execute, while the debugger is polling for status. The parameters to control this number are specified in the configuration file.

```
system.vt.pollingpause 1ms
```

Xtensa OCD

The following setting enables the emulator-run evaluations to reach the target time of 1 ms or execute 100000 clock cycles, respectively. These numbers need to be adjusted to balance the Palladium run-time versus debugger responsiveness. Larger values allow the Palladium emulator to execute the code faster on the CPU, at the cost of the debugger-reaction time.

```
poll-low='1000' poll-high='100000'
```

Development Studio

The code-execution time can be improved by decreasing the default polling frequency.

1. Open the configured platform `SDF` file as text by selecting the file in *Project Explorer* under *Boards/manufacturer/platform* and right-clicking on *Open With Text Editor*.
2. Under the <*debug*> section under *SWJEnable* add a new line:
   ```
   <config_item
   name="StatusTimeoutRunning">1000</config_item>
   ```
   This setting instructs the Development Studio to check for status once every second, instead of the default 100 times per second.

   > ⓘ You can also add the following line to reduce polling when stopped from 1000 per second to 4:
   > ```
   > <config_item
   > name="StatusTimeoutStopped">250</config_item>
   > ```

3. Save the SDF file (Ctrl+S or *File/Save*).

OpenOCD

The polling numbers need to be adjusted to balance the Palladium run-time versus debugger responsiveness. Larger values allow Palladium to execute the code faster at the cost of the debugger reaction time.

```
vdebug polling 100 1000
```

Related Sections:

- Palladium Performance Numbers
- Protium Performance Numbers
- Configuring and Running Lauterbach Trace 32
- Configuring and Running Xtensa OCD
- Configuring and Running Development Studio
- Configuring and Running OpenOCD

12

Palladium Performance Numbers

This topic specifies the numbers you can refer for determining performance for Palladium.

Performance Numbers for JTAG BFM

The following table lists some single-step performance numbers measured with the maximal JTAG clock frequency when the fastest design clock drives the JTAG BFM.

CPU	Mode	Debugger	Palladium Z1			Palladium Z2		
			fclk	tclk	step	fclk	tclk	step
ARM Cortex A53	2x	Trace32	2.3MHz	575kHz	0.691s	2.6MHz	650kHz	0.628s
		ARM DS			0.755s			0.639s
ARM Cortex A53	1x	Trace32	2.1MHz	1.05MHz	0.598s	2.2MHz	1.1MHz	0.557s
		ARM DS			0.614s			0.550s
ARM Cortex A75	2x	Trace32	1.4MHz	350kHz	1.059s	2.5MHz	625kHz	0.703s
		ARM DS			1.097s			0.841s
ARM Cortex A75	1x	Trace32	1.5MHz	750kHz	0.788s	2.2MHz	1.1MHz	0.682s
		ARM DS			0.990s			0.759s
ARM Cortex M7F	2x	Trace32	2.7MHz	675kHz	0.073s	3.1MHz	775kHz	0.061s
		ARM DS			0.103s			0.081s
		OpenOCD+GDB			0.088s			0.060s
ARM Cortex M7F	1x	Trace32	2.3MHz	1.15MHz	0.056s	2.5MHz	1.25MHz	0.047s
		ARM DS			0.094s			0.072s
		OpenOCD+GDB			0.066s			0.043s
ARM Cortex M4F	2x	Trace32	2.7MHz	675kHz	0.067s	3.1MHz	775kHz	0.063s
		ARM DS			0.062s			0.081s
		OpenOCD+GDB			0.071s			0.061s
ARM Cortex M4F	1x	Trace32	2.2MHz	1.1MHz	0.054s	2.6MHz	1.3MHz	0.048s
		ARM DS			0.060s			0.070s
		OpenOCD+GDB			0.060s			0.043s
ARM Cortex M33F	2x	Trace32	2.5MHz	625kHz	0.077s	3.1MHz	775kHz	0.060s

		ARM DS			0.162s			0.126s
ARM Cortex M33F	1x	Trace32	2.2MHz	1.1MHz	0.055s	2.7MHz	1.35MHz	0.047s
		ARM DS			0.126s			0.115s
ARM Neoverse N1	2x	Trace32	1.5MHz	375kHz	1.143s	2.6MHz	650kHz	0.713s
		ARM DS			1.206s			1.077s
ARM Neoverse N1	1x	Trace32	1.4MHz	700kHz	0.830s	2.2MHz	1.1MHz	0.562s
		ARM DS			0.955s			0.876s
ARM Cortex R5	2x	Trace32	3.0MHz	750kHz	0.418s	2.8MHz	700kHz	0.395s
		ARM DS			0.571s			0.461s
ARM Cortex R5	1x	Trace32	2.1MHz	1.05MHz	0.385s	2.7MHz	1.35MHz	0.341s
		ARM DS			0.457s			0.363s
ARM Cortex R52	2x	Trace32	2.1MHz	525kHz	0.675s	2.8MHz	700kHz	0.397s
		ARM DS			0.470s			0.407s
ARM Cortex R52	1x	Trace32	2.0MHz	1.0MHz	0.552s	2.5MHz	1.25MHz	0.334s
		ARM DS			0.406s			0.302s
Tensilica Xtensa LX7	2x	Trace32	2.8MHz	700kHz	0.192s	2.9MHz	725kHz	0.104s
		XOCD+GDB			0.047s			0.031s
Tensilica Xtensa LX7	1x	Trace32	2.3MHz	1.15MHz	0.189s	2.5MHz	1.25MHz	0.080s
		XOCD+GDB			0.039s			0.018s
Ibex RV32IMC	2x	Trace32	2.8MHz	700kHz	0.077s	2.9MHz	725KHz	0.061s
		OpenOCD+GDB			0.076s			0.053s
Ibex RV32IMC	1x	Trace32	2.4MHz	1.2MHz	0.064s	2.6MHz	1.3MHz	0.046s
		OpenOCD+GDB			0.060s			0.039s

Performance Numbers for APB BFM

The following table lists some single-step performance numbers measured when the fastest design clock drives the APB BFM.

CPU	Mode	Debugger	Palladium Z1			Palladium Z2		
			fclk	pclk_i	step	fclk	pclk_i	step
ARM Cortex A53	2x	Trace32	2.3MHz	1.15MHz	0.050s	2.6MHz	1.3MHz	0.034s
ARM Cortex A53	1x	Trace32	2.1MHz	2.1MHz	0.046s	2.2MHz	2.2MHz	0.031s
ARM Cortex A75	2x	Trace32	1.4MHz	700KHz	0.080s	2.5MHz	1.25MHz	0.050s
ARM Cortex A75	1x	Trace32	1.5MHz	1.5MHz	0.054s	2.2MHz	2.2MHz	0.038s
ARM Neoverse N1	2x	Trace32	1.5MHz	750KHz	0.080s	2.6MHz	1.3MHz	0.043s
ARM Neoverse N1	1x	Trace32	1.4MHz	1.4MHz	0.060s	2.2MHz	2.2MHz	0.035s
ARM Cortex R5	2x	Trace32	3.0MHz	1.5MHz	0.047s	2.8MHz	1.4MHz	0.038s
ARM Cortex R5	1x	Trace32	2.1MHz	2.1MHz	0.042s	2.7MHz	2.7MHz	0.033s
ARM Cortex R52	2x	Trace32	2.1MHz	1.05MHz	0.038s	2.8MHz	1.4MHz	0.025s
ARM Cortex R52	1x	Trace32	2.0MHz	2.0MHz	0.031s	2.5MHz	2.5MHz	0.021s
Tensilica Xtensa LX7	2x	Trace32	2.8MHz	1.4MHz	0.054s	2.9MHz	1.45MHz	0.027s
		XOCD+GDB			0.038s			0.019s
Tensilica Xtensa LX7	1x	Trace32	2.3MHz	2.3MHz	0.053s	2.5MHz	2.5MHz	0.023s
		XOCD+GDB			0.035s			0.012s

Performance Numbers for SWDP BFM

The following table lists some single-step performance numbers measured when the fastest design clock drives the SWDP BFM.

CPU	Mode	Debugger	Palladium Z1			Palladium Z2		
			fclk	pclk_i	step	fclk	pclk_i	step
ARM Cortex A53	2x	Trace32	2.3MHz	1.15MHz	0.161s	2.6MHz	1.3MHz	0.100s
		ARM DS			0.468s			0.381s
ARM Cortex A53	1x	Trace32	2.1MHz	2.1MHz	0.156s	2.2MHz	2.2MHz	0.094s
		ARM DS			0.353s			0.326s
ARM Cortex A75	2x	Trace32	1.4MHz	700kHz	0.192s	2.5MHz	1.25MHz	0.114s
		ARM DS			0.538s			0.390s
ARM Cortex A75	1x	Trace32	1.5MHz	1.5MHz	0.126s	2.2MHz	2.2MHz	0.089s
		ARM DS			0.514s			0.364s
ARM Cortex M4F	2x	Trace32	2.7MHz	1.35MHz	0.014s	3.1MHz	1.55MHz	0.009s
		ARM DS			0.030s			0.040s
ARM Cortex M4F	1x	Trace32	2.2MHz	2.2MHz	0.012s	2.6MHz	2.6MHz	0.008s
		ARM DS			0.028s			0.038s
ARM Neoverse N1	2x	Trace32	1.5MHz	750kHz	0.166s	2.6MHz	1.3MHz	0.098s
		ARM DS			0.436s			0.329s
ARM Neoverse N1	1x	Trace32	1.4MHz	1.4MHz	0.123s	2.2MHz	2.2MHz	0.076s
		ARM DS			0.382s			0.308s
ARM Cortex R5	2x	Trace32	3.0MHz	1.5MHz	0.071s	2.8MHz	1.4MHz	0.060s
		ARM DS			0.157s			0.121s
ARM Cortex R5	1x	Trace32	2.1MHz	2.1MHz	0.063s	2.7MHz	2.7MHz	0.042s
		ARM DS			0.151s			0.117s
ARM Cortex R52	2x	Trace32	2.1MHz	1.05MHz	0.097s	2.8MHz	1.4MHz	0.062s

		ARM DS			0.235s			0.196s
ARM Cortex R52	1x	Trace32	2.0MHz	2.0MHz	0.084s	2.5MHz	2.5MHz	0.053s
		ARM DS			0.196s			0.175s

Performance Numbers for AHB BFM

The following table lists some single-step performance numbers measured when the fastest design clock drives the AHB BFM.

CPU	Mode	Debugger	Palladium Z1			Palladium Z1		
			fclk	pclk_i	step	fclk	pclk_i	step
ARM Cortex A53	2x	Trace32	2.3MHz	1.15MHz	0.048s	2.6MHz	1.3MHz	0.037s
ARM Cortex A53	1x	Trace32	2.1MHz	2.1MHz	0.041s	2.2MHz	2.2MHz	0.032s
ARM Cortex M7F	2x	Trace32	2.7MHz	1.35MHz	0.005s	3.1MHz	1.55MHz	0.003s
ARM Cortex M7F	1x	Trace32	2.3MHz	2.3MHz	0.004s	2.5MHz	2.5MHz	0.003s
ARM Cortex M33F	2x	Trace32	2.5MHz	1.25MHz	0.005s	3.2MHz	1.6MHz	0.003s
ARM Cortex M33F	1x	Trace32	2.2MHz	2.2MHz	0.004s	2.7MHz	2.7MHz	0.003s
ARM Cortex M4F	2x	Trace32	2.7MHz	1.35MHz	0.005s	3.1MHz	1.55MHz	0.003s
ARM Cortex M4F	1x	Trace32	2.2MHz	2.2MHz	0.004s	2.6MHz	2.6MHz	0.002s

Performance Numbers for AXI BFM

The following table lists some single-step performance numbers measured when the fastest design clock drives the AXI BFM.

CPU	Mode	Debugger	Palladium Z1			Palladium Z2		
			fclk	pclk_i	step	fclk	pclk_i	step
ARM Cortex A53	2x	Trace32	2.3MHz	1.15MHz	0.047s	2.6MHz	1.3MHz	0.035s
ARM Cortex A53	1x	Trace32	2.1MHz	2.1MHz	0.040s	2.2MHz	2.2MHz	0.031s

Related Sections:

- Improving Transactor Performance

- Protium Performance Numbers

13

Protium Performance Numbers

This topic specifies the numbers you can refer for determining performance for Protium X1 and X2.

- Performance Numbers for JTAG BFM
- Performance Numbers for APB BFM
- Performance Numbers for SWDP BFM
- Performance Numbers for AHB BFM

Performance Numbers for JTAG BFM

The following table lists some single-step performance numbers measured with the maximal JTAG clock frequency when the fastest design clock drives the JTAG BFM.

CPU	Mode	Debugger	Protium X1			Protium X2		
			fclk	tclk	step	fclk	tclk	step
ARM Cortex A53*	2x	Trace32	12.2MHz	3.05MHz	0.372s	12.1MHz	3.0MHz	0.462s
ARM Cortex A53*	1x	Trace32	10.9MHz	5.45MHz	0.360s	12.1MHz	6.1MHz	0.442s
ARM Cortex A75	2x	Trace32	3.7MHz	925kHz	0.601s	12.1MHz*	3.0MHz	0.597s
ARM Cortex A75	1x	Trace32	3.0MHz	1.5MHz	0.590s	12.1MHz*	6.1MHz	0.507s
ARM Cortex M4F	2x	Trace32	20.1MHz	5.0MHz	0.035s	20.8MHz	5.2MHz	0.036s
ARM Cortex M4F	1x	Trace32	13.6MHz	6.8MHz	0.033s	15.6MHz	7.8MHz	0.036s
ARM Cortex M7F	2x	Trace32	22.0MHz	5.5MHz	0.037s	20.8MHz	5.2MHz	0.038s
ARM Cortex M7F	1x	Trace32	15.5MHz	7.75MHz	0.036s	15.6MHz	7.8MHz	0.037s

*Compiled for only one FPGA with XSRAM card.

Performance Numbers for APB BFM

The following table lists some single-step performance numbers measured when the fastest design clock drives the APB.

CPU	Mode	Debugger	Protium X1			Protium X2		
			fclk	pclk_i	step	fclk	pclk_i	step
ARM Cortex A53*	2x	Trace32	12.2MHz	6.1MHz	0.028s	12.1MHz	6.1MHz	0.028s
ARM Cortex A53*	1x	Trace32	10.9MHz	10.9MHz	0.027s	12.1MHz	12.1MHz	0.026s
ARM Cortex A75	2x	Trace32	3.7MHz	1.85MHz	0.035s	12.1MHz*	6.1MHz	0.031s
ARM Cortex A75	1x	Trace32	3.0MHz	3.0MHz	0.032s	12.1MHz*	12.1MHz	0.030s

*Compiled for only one FPGA with XSRAM card.

Performance Numbers for SWDP BFM

The following table lists some single-step performance numbers measured when the fastest design clock drives the SWDP BFM.

CPU	Mode	Debugger	Protium X1			Protium X2		
			fclk	pclk_i	step	fclk	pclk_i	step
ARM Cortex A75	2x	Trace32	3.7MHz	1.85MHz	0.098s	12.1MHz*	6.1MHz	0.073s
ARM Cortex A75	1x	Trace32	3.0MHz	3.0MHz	0.079s	12.1MHz*	12.1MHz	0.072s
ARM Cortex M4F	2x	Trace32	20.1MHz	10.5MHz	0.008s	20.8MHz	10.4MHz	0.008s
ARM Cortex M4F	1x	Trace32	13.6MHz	13.6MHz	0.008s	15.6MHz	15.6MHz	0.008s

Performance Numbers for AHB BFM

The following table lists some single-step performance numbers measured when the fastest design clock drives the AHB BFM.

CPU	Mode	Debugger	Protium X1			Protium X2		
			fclk	pclk_i	step	fclk	pclk_i	step
ARM Cortex M4F	2x	Trace32	20.1MHz	10.5MHz	0.002s	20.8MHz	10.4MHz	0.002s
ARM Cortex M4F	1x	Trace32	13.6MHz	13.6MHz	0.002s	15.6MHz	15.6MHz	0.002s
ARM Cortex M7F	2x	Trace32	22.0MHz	11.0MHz	0.002s	20.8MHz	10.4MHz	0.002s
ARM Cortex M7F	1x	Trace32	15.5MHz	15.5MHz	0.002s	15.6MHz	15.6MHz	0.002s

Related Sections:

- Improving Transactor Performance
- Palladium Performance Numbers

14

Compile and Debug Error Codes

The following table lists common errors and the possible causes. Most likely, the client or the debugger returns an error. However, some of these errors might occur and be reported by the BFM (when compiled with the `-define DEBUG` option).

Error	Code	Description
`VD_ERR_NONE`	`0x0000`	No error.
`VD_ERR_NOT_IMPL`	`0x0100`	Feature not implemented.
`VD_ERR_USAGE`	`0x0101`	Incorrect usage of the function.
`VD_ERR_PARAM`	`0x0102`	Incorrect parameter type or value passed.
`VD_ERR_NO_MEMORY`	`0x0104`	Program cannot allocate heap memory.
`VD_ERR_CONFIG`	`0x0107`	Incorrect configuration parameter or value.
`VD_ERR_SHM_OPEN`	`0x010a`	Cannot open shared memory. Either it does not exist or permission is not available. Check `/dev/shm`.
`VD_ERR_SHM_MAP`	`0x010b`	Cannot map a shared memory to s system file.
`VD_ERR_SOC_OPEN`	`0x011a`	Cannot open socket on given `host:port`.
`VD_ERR_SOC_OPT`	`0x011b`	Cannot set the socket options. Report the OS.
`VD_ERR_SOC_ADDR`	`0x011c`	Cannot resolve the name to IP address.
`VD_ERR_SOC_CONN`	`0x011d`	Cannot connect the server. Server might be busy.
`VD_ERR_SOC_SEND`	`0x011e`	Error sending data through socket, network issue or client/server process died.
`VD_ERR_SOC_RECV`	`0x011f`	Error receiving data on the socket, network issue or client/server process died.

VD_ERR_LOCKED	0x0202	Device locked.
VD_ERR_NOT_RUN	0x0204	Transactor not running yet.
VD_ERR_NOT_OPEN	0x0205	Transactor not open or connected yet.
VD_ERR_LICENSE	0x0206	Cannot check out the vdebug license.
VD_ERR_VERSION	0x0207	Version mismatch. The version of the vdclient library does not match or accept the version of the vdserver.
VD_ERR_TIME_OUT	0x0301	Time out waiting for response. Can indicate network issue, or design problem, like wiring or stuck signals.
VD_ERR_NO_POWER	0x0302	The design or DAP is not powered up. Check the (c)dbgpwrupack_i input to the BFM, must be 1.
VD_ERR_BUS_ERROR	0x0304	Bus protocol error, like pslverr. Check the waveforms.
VD_ERR_NO_ACCESS	0x0306	No access to an object.
VD_ERR_INV_HANDLE	0x0307	Invalid handle to an object.
VD_ERR_INV_SCOPE	0x0308	Invalid hierarchy scope.

15

Examples of Using the Virtual Debug Interface

Two examples, shipped with the release, demonstrate the usage of the Virtual Debug Interface.

- Example: JTAG Connectivity Test
- Example: Pulpissimo Design with Ibex Core

> Note
> Modify and source the `setup_example.csh` file for software environment setup.

Example: JTAG Connectivity Test

This is a simple example containing the TAP controller and a vdebug JTAG transactor to test the JTAG connectivity and the debugger setup. It can be found in the `example/test_jtag` directory.

The design is shipped with compile and run scripts for the Cadence tools.

- Xcelium: `build_sim` and `run_sim`
- Palladium: `build_xe` and `run_xe`
- Protium: `build_ptm`, `bsum_ptm`, and `run_ptm`

Run scripts are provided for Trace32 and OpenOCD.

The testcase has been put together to demonstrate the JTAG set up with the Cadence Virtual Debug Interface.

The TAP source has been downloaded from `http://www.opencores.org/projects/jtag`.

The debug registers come from `http://www.opencores.org/projects/DebugInterface`.

The design changes include:

- Un-defining the `DGB_CPU_DELAY` in `dbg_cpu_defines.v`

- Un-defining the `TAP_DELAY` in `tap_defines.v`

Compiling the Design for JTAG Connectivity Test Example

The Xcelium compilation needs Xcelium to be installed and the following environment variables to be set:

- `XLMHOME` set to Xcelium installation root
- `PATH` to include Xcelium binaries
- `build_sim` takes following options
    - `work_dir` <*name*> sets compile directory, `c_sim` is a default
    - `tarmac`: compiles in execution trace capabilities
    - `debug`: build a debug version with `-linedebug`

The Palladium compilation needs both Xcelium and VXE / WXE to be installed and the following environment variables to be set:

- `XLMHOME` set to Xcelium installation root
- `XEHOME` set to VXE/ WXE installation root
- `IXCOMHOME` set to IXCOM installation root
- `PATH` to include Xcelium, VXE/ WXE, IXCOM, and HDL-ICE binaries
- `build_xe` takes the following options:
    - `-work_dir` <*name*> sets compile directory, `c_xe` is a default
    - `-tarmac`: compiles in execution trace capabilities
    - `-debug`: build a debug version with additional messages
    - `-1xclk`: uses 1X oversampling

The Protium compilation needs both Xcelium and Protium to be installed and the following environment variables to be set:

- `XLMHOME` set to Xcelium installation root
- `PTMHOME` set to PTM installation root
- `IXCOMHOME` set to IXCOM installation root

- `PATH` to include Xcelium, PTM, IXCOM and HDL-ICE binaries
- `build_ptm` takes the following options:
    - `-work_dir` <*name*> sets compile directory, `c_xe` is a default
    - `-tarmac`: compiles in execution trace capabilities
    - `-debug`: build a debug version with additional messages
    - `-1xclk`: uses 1X oversampling

The path to vdebug installation is specified as VD_REL in build_* script files:

- VD_REL=../../.. (by default)

Running the Design for JTAG Connectivity Test Example

`run_sim` takes following options

- `-work_dir` <*name*> uses compile directory, `c_sim` is a default
- `-tarmac`: enables execution trace
- `-debug`: enables vdebug client logging
- `-gui`: brings up the GUI

`run_xe` takes following options:

- `-work_dir` < *name* > uses compile directory, `c_xe` is a default
- `-tarmac`: enables execution trace
- `-debug`: enables vdebug client logging
- `-gui`: brings up the GUI

`run_ptm` takes following options:

- `-work_dir` < *name* > uses compile directory, `c_ptm` is a default
- `-tarmac`: enables execution trace
- `-debug`: enables vdebug client logging
- `-gui`: brings up the GUI

Connecting the Debugger for JTAG Connectivity Test Example

Take the following steps to connect the debuggers.

Trace32

Modify the `config.t32` to point `SYS` to the Trace32 installation root directory.

Start Trace32:

```
% <install>/bin/pc_linux64/t32m* -s t32.jtag.cmm
```

The Trace32 shows the terminal window with the JTAG properties detected through `system.detect.daisychain` command.

OpenOCD

Start OpenOCD compiled with vdebug support:

```
% openocd -f ocd.jtag.cfg
```

OpenOCD connects and shows the detected TAP properties.

OpenOCD can be disconnected from the vdebug server by pressing `Ctrl-C` in the terminal.

Example: Pulpissimo Design with Ibex Core

The Pulpissimo design with the RISCV Ibex core is used to demonstrate the Cadence Virtual Debug Interface.

The Pulpissimo is one of the platforms available from `http://www.pulp-platform.org`.

It has been downloaded from `https://github.com/pulp-platform/pulpissimo`.

The Ibex core has been downloaded from `https://github.com/lowRISC/ibex`.

The Pulpissimo design changes include:

- Modification of the `pulpissimo.sv` top-level to be a fully synthesizable top level with emulation reset, clock generation, and vdebug BFMs
- Modification to the FC subsystem to instantiate the Ibex core with tracing
- Modification to the `pulp_soc` subsystem to instantiate emulation friendly SRAM for the L2 and L2 primary banks, reduction of the number of hearts, removal of the Pulp TAP

The Ibex core changes include rewriting the trace module to be synthesizable.

Pulpissimo Design

PULPissimo is the microcontroller architecture of the more recent PULP chips, part of the ongoing PULP platform collaboration between ETH Zurich and the University of Bologna. The collaboration started in 2013.

PULPissimo, like PULPino, is a single-core platform. However, it represents a significant step ahead in terms of completeness and complexity with respect to PULPino. The PULPissimo system is used as the main System-on-Chip controller for all recent multi-core PULP chips, taking care of autonomous I/O, advanced data pre-processing, external interrupts, and so on. The PULPissimo architecture includes:

- Either the RI5CY core or the Ibex one as main core
- Autonomous Input/Output subsystem (uDMA)
- New memory subsystem
- Support for Hardware Processing Engines (HWPEs)
- New simple interrupt controller
- New peripherals

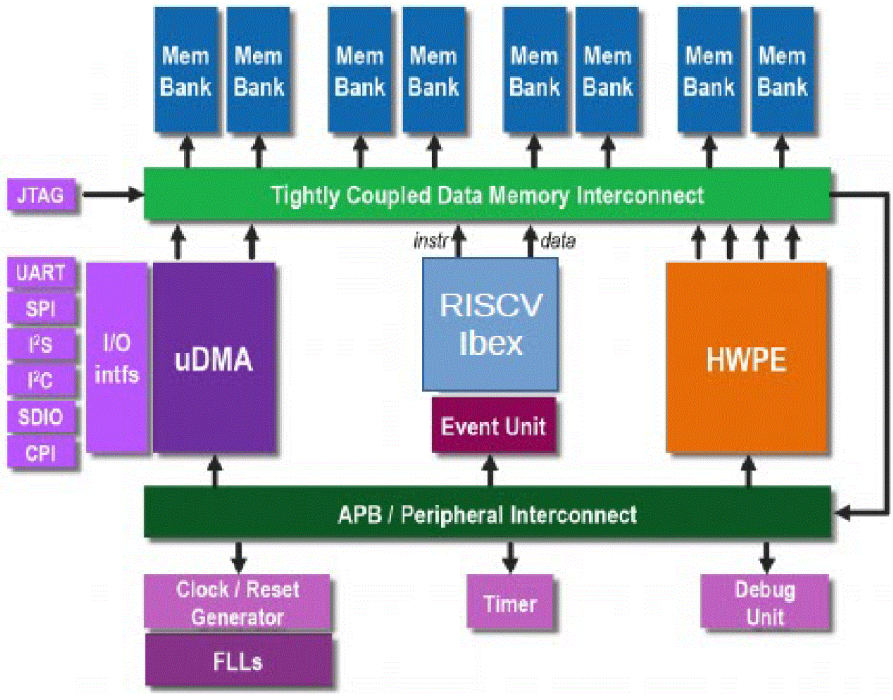
- New SDK

Ibex, formely Zero-riscy, is an in-order, single-issue core with two pipeline stages. It has full support for the base-integer instruction set (RV32I version 2.1) and compressed instructions (RV32C version 2.0). It can be configured to support the multiplication instruction set extension (RV32M version 2.0) and the reduced number of registers extension (RV32E version 1.9). Ibex implements the Machine ISA version 1.11 and has RISC-V External Debug Support version 0.13.2. Ibex has been originally designed at ETH to target ultra-low-power and ultra-low-area constraints. Ibex is now maintained and further developed by the non-profit community interest company lowRISC.

Further information about the core can be found at `http://ieeexplore.ieee.org/document/8106976/` and in the documentation of the IP at the following link:

```
https://ibex-core.readthedocs.io/en/latest/index.html
```

The design architecture is depicted below:



The design is shipped with compile and run scripts for the Cadence tools in the `example/ibex_pulp_j` directory.

- Xcelium: `build_sim` and `run_sim`
- Palladium: `build_xe` and `run_xe`

- Protium: `build_ptm`, `bsum_ptm`, and `run_ptm`
- Run scripts are provided for Trace32 RISCV driver `t32mriscv` and OpenOCD.

Compiling the Design for Pulpissimo Design with Ibex Core Example

The Xcelium compilation needs Xcelium installed and environment variables.

- `XLMHOME` set to Xcelium installation root
- `PATH` to include Xcelium binaries
- `build_sim` takes following options
  - `-work_dir` <*name*>: sets compile directory, `c_sim` is the default
  - `-tarmac`: compiles in execution trace capabilities
  - `-debug`: build a debug version with `-linedebug`

The Palladium compilation needs both Xcelium and VXE / WXE installed and environment variables set:

- `XLMHOME` set to Xcelium installation root
- `XEHOME` set to VXE / WXE installation root
- `IXCOMHOME` set to IXCOM installation root
- `PATH` to include Xcelium, VXE / WXE, IXCOM, and HDLICE binaries
- `build_xe` takes following options:
  - `-work_dir` <*name*>: sets compile directory, `c_xe` is the default
  - `-tarmac`: compiles in execution trace capabilities
  - `-debug`: build a debug version with additional messages
  - `-1xclk`: uses 1X oversampling

The Protium compilation needs both Xcelium and Protium PTM to be installed and environment variables to be set.

- `XLMHOME` set to Xcelium installation root
- `PTMHOME` set to PTM installation root
- `IXCOMHOME` set to IXCOM installation root

- `PATH` to include Xcelium, PTM, IXCOM, and HDLICE binaries
- `build_ptm` takes following options:
    - `-work_dir` <*name*> sets compile directory, `c_xe` is the default
    - `-tarmac`: compiles in execution trace capabilities
    - `-debug`: build a debug version with additional messages
    - `-1xclk`: uses 1X oversampling

The path to vdebug installation is specified as VD_REL in build_* script files:

- VD_REL=../../.. (by default)

Running the Design for Pulpissimo Design with Ibex Core Example

`run_sim` takes following options

- `-work_dir` <*name*> uses compile directory, `c_sim` is the default
- `-tarmac`: enables execution trace
- `-debug`: enables vdebug client logging
- `-gui`: brings up the GUI

`run_xe` takes following options:

- `-work_dir` <*name*> uses compile directory, `c_xe` is the default
- `-tarmac`: enables execution trace
- `-debug`: enables vdebug client logging
- `-gui`: brings up the GUI

`run_ptm` takes following options:

- `-work_dir` <*name*> uses compile directory, `c_ptm` is the default
- `-tarmac`: enables execution trace
- `-debug`: enables vdebug client logging
- `-gui`: brings up the GUI

Connecting the Debugger for Pulpissimo Design with Ibex Core Example

Take the following steps to connect the debuggers.

Trace32

After the design is up and running with the default parameters, start Trace32 in a separate terminal with the provided script.

```
% <install>/bin/pc_linux64/t32mriscv -s t32.jtag.cmm
```

OpenOCD

After the design is up and running with the default parameters, start OpenOCD in a separate terminal with the provided script.

```
% <install>/bin/openocd -f ocd.jtag.cfg
```

> Important
> The script uses polling and backdoor memory access and works only if OpenOCD 0.12 has been patched to include this functionality. Otherwise, it needs to be modified by commenting out the following lines:
>
> ```
> # vdebug polling 100 1000
>
> # vdebug register_target
> ```

The supplied configuration file configures the JTAG chain to connect to the RISCV core through vdebug interface. The OpenOCD contains the gdbserver, which is configured to listen on port 3333.

At this point, the RISCV gdb can connect to OpenOCD, load the code, and debug the software.

The design comes with three software tests under `tests` directory:

- A simple `hello_world` assembly example
- A hello C example compiled with Pulp runtime
- A coremark C example, that measures the core performance.

Every test comes with the GDB init file `.gdbinit`, that gets loaded automatically, when it is allowed in the user environment.

To allow this, write a command to your `~/.gdbinit` file:

```
set auto-load safe-path /
```

Else, the `.gdbinit` file needs to be loaded manually in a terminal:

```
source .gdbinit
```

Switch to one of the directory `tests/`<*test*>.

When starting `gdb` on a different workstation than the OpenOCD, you need to update a line in `.gdbinit`:

```
target remote <host>:3333
```

```
% <path_to_riscv_toolchain>/bin/riscv[32|64]-unknown-elf-gdb <example>.elf
```

The code is loaded and gdb can start debugging the code. When done type `detach` to detach the gdb and `quit` to exit from gdb. The gdb can be started again with a different test.

The OpenOCD can be disconnected from the vdebug server by pressing Ctrl-C in the terminal.